
paradigm

Sharing your work

A reproducible data science workflow using open source tools

Martin Frigaard

This book is for sale at http://leanpub.com/showingyourwork

This version was published on 2019-07-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. What's in this manual?

"*If you can't describe what you are doing as a process, you don't know what you're doing.*" - W. Edwards Deming

This manual is for getting started in data science using R & RStudio, and Git & Github. We will also cover some practical principles of programming, data collection, analysis, and visualizations, and a few computer science topics.

There have been a growing number of excellent publications on a variety of topics in data science (see list below). Rather than write a manual that focused on one of these tools or methods, we wrote a manual for *how these technologies work together*. Because this is a very "high level" manual, we also encourage readers to consult the original publications for more details on each specific topic.

Our goal is to include enough information to get you up and running and at the same time, not overwhelm you. Odds are you've already Googled "*Getting started in [insert open-source software]*," and you know there are a ton of resources available online for anything related to data science, data visualization, machine learning, etc. Figuring out where to start can feel like trying to get a drink of water from a fire hose. Given the massive number of resources, we decided to distill what we consider to be the "common threads" that run through each of these technologies. Along the way, we are going to introduce a few computer science concepts that will help you build a toolkit, get started on a project, take that project to an outcome, and communicate the findings.

Who this manual is for

We've tried to keep this manual accessible to everyone. Whether you're currently using data in your job and want to add these tools to your skillset, or you'd like to start using data to improve what you do. Or maybe you have a knack for counting things and creating pictures. In the next few pages, we're going to introduce you to a workflow (a set of tools) for collecting, manipulating, summarizing, and visualizing data in a reproducible way. We wrote this manual to try and distill these topics into a 'bare-minimum' toolset that you can learn and use quickly (because we know your time is limited).

Whether you're an accountant, scientist, data analyst, journalist, grad student, product manager, or decision-maker, this manual is for you.

What this manual covers

This manual covers getting started with RStudio, Git, and Github. Although we use RStudio daily now, this wasn't always the case. We began our careers in other statistical programs (SPSS, Stata, SAS), and abandoned them for one reason or another. We've continued using R/RStudio because of the sheer number of tasks we can accomplish, and that's what makes us recommend it to you.

Many of the concepts covered in this manual come from our personal experiences. We wanted to make a manual that contained the analytic skills we didn't have after graduating from college but needed to be successful in our jobs. Everything in this manual was included to make you more prepared than we were. We also reached out to our colleagues and included their lessons and insights.

What this manual doesn't cover

We also understand there are alternative approaches to accomplishing the same goal, and we will try to mention these examples wherever possible. Jupyter Lab and Jupyter Notebooks, for example, offer reproducible scientific programming environments that can accomplish many of the same objectives we'll tackle in this book. However, we still think there are reasons you should use RStudio + Github instead, and we will outline these in the following chapters.

Our approach to writing

The US Army is amazingly efficient at training large groups of people on a wide range of tasks in a remarkably short period. Army leadership accomplishes this by addressing four considerations when planning a training exercise:

- (a) *Relative importance*. Which activities contribute most to successful training?

  This manual contains **brief descriptions** of the tools we recommend, **diagrams and figures** outlining how they work, and **examples** for using them.

- (b) *Need*. Which training activities will benefit the most from guidance? Which activities have received little attention in the past or which have previously required improvement?

  We expand on some technologies we feel are harder to grasp (Git and version control), and go over topics college courses overlook or neglect (file naming, project organization).

- (c) *Time*. How much time is available? Which activities can be effectively taught in that time?

Time is the real enemy of any data project. All computational work comes down to keystrokes and neurons. We're trying to narrow the time gap between 1) having a question or task (neurons) and 2) typing commands a computer can execute (and a human can read).

- (d) *Personnel*. What are the known or suspected levels of expertise among individuals receiving training?

  We assume everyone reading this text has zero exposure to the tools we will be covering (R, RStudio, Git, or Github).

Another secret to the Army's training abilities is the Field Manual (FM). Army FMs are amazing–they cover almost any topic you can imagine. For example, watch this video of the drill and ceremony movement called the "counter column"[1].

As you can see, this is a complicated technique. But somehow the Army has been able to teach hundreds of thousands of soldiers for decades. How do they achieve this? By giving soldiers a field manual (FM 22-5) and lots of practice.

The strength of FMs is how they're written: they present the material in everyday language (usually between a 6th-8th-grade reading level), using a lot of diagrams and hand-drawn pictures.

For example, below is a diagram for a performing a counter column from FM 22-5 Army Drill and Ceremony.

[1] https://www.youtube.com/watch?v=EgeZl9UOJ0I

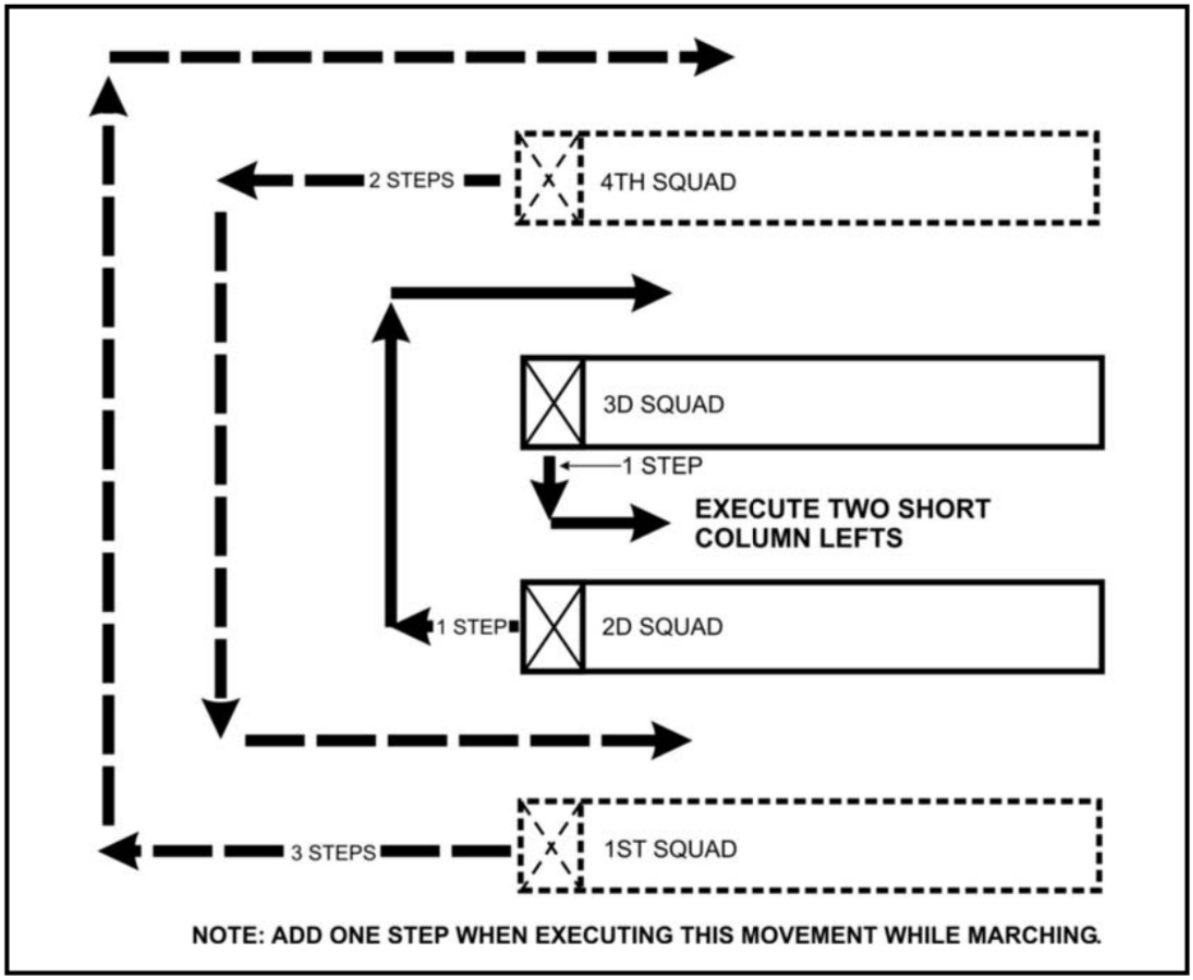


Figure 7-2. Counter-Column March at the Halt.

The image shows how to turn a platoon of soldiers (made up of four squads) around 180 degrees. Describing something like this with words becomes over-complicated quickly.

> "…there are lots of other books that explain what things are called. This book explains what they do." Randall Munroe, Thing Explainer

The quote above comes from an excellent book we recommend everyone reads. The author uses pictures and plain language to describe multiple complicated things (rocket ships, the periodic table, laptops, etc.). The subtitle of the book, "*Complicated stuff in simple words*" is what we're trying to replicate here. Wherever possible, we've dropped unnecessary technical jargon and spelled out any acronyms. We've also created images and diagrams instead of words to explain complicated concepts.

Our goals for you

The two primary goals we hope to accomplish in this manual are to 1) give you a workflow that allows you to collaborate with other people, and 2) communicate your work in a reproducible way to a wide range of audiences. We view collaboration and reproducibility as connected concepts because the better your collaborators can reproduce your work, the better they'll understand your results.

Assumptions we've made

We assume you'll be working on a computer (laptop or desktop), and have the ability to download, locate, and install software applications. Although the examples in this manual use a Mac operating system (macOS), it's possible to recreate everything on a Windows machine.

We also assume you've been using your computer to send/receive emails, write papers, and explore the internet. People encounter a need for data science skills at different times in their lives, so we realize there's a chance a few of you will be familiar with some of the content we're covering. If this is the case, hopefully, we cover it in a novel and painless way that doesn't make reading it feel like a waste of time.

We use the plural 'we' throughout the manual based on the excellent advice[2] from Donald Knuth, Tracy Larabee, and Paul Roberts, "*think of a dialog between author and reader.*." As with most written works, the topics in this manual are the result of many conversations, emails, comment threads, and communications that could not have happened in isolation.

Style guide

The text uses the following style guide:

`this is code.`

```
# this a code chunk
```

> *some quoted text*

click on hyperlinks[3]

plain text for our thoughts

[2] http://www.econ.uiuc.edu/~econ508/Papers/mathwriting.pdf

[3]

2. Part 1: Modern research

We are going to cover the two motivations behind this manual: collaboration and reproducibility. We briefly mention these topics in both academia and the private sector, and explain what we've found at various universities and companies; Our goal is that you see how you can benefit by adopting the best practices from both arenas.

Research in academia

Science has undergone significant changes in the last 10+ years. Personal computers made writing papers and analyzing data accessible to more scientists, and the internet has made publishing faster and easier. Unfortunately, there are some sticky spots where academic research hasn't caught up with modern technology. Jeff Leek[1], biostatistician and Professor at Johns Hopkins Bloomberg School of Public Health, alludes to this in his excellent book, "How to be a modern scientist"[2],

> *The modern academic scientist develops code in the open, publishes data and code open-source, posts preprints of their academic work, still submits to traditional journals, and reviews for those journals, but may also write blog posts or use social media to critique published work in post-publication review fora. These activities can dramatically increase the profile of scientists, particularly junior scientists if done well. But their value for important career milestones such as promotion and tenure or getting grants is still often muted or fuzzy.*

It's been over three years since Dr. Leek published his text, and the good news is that more modern scientific work is done in the open[3], using open source code and software[4], using pre-prints[5], social media[6], and other mediums to communicate[7]).

Unfortunately, a fair amount of academic scientists still aren't using modern methods. According to a 2018 article PLoS[8], only 14% of public health professionals are using reproducible methods. We also know that many scientists believe

[1] http://jtleek.com/
[2] https://leanpub.com/modernscientist
[3] https://www.nature.com/articles/d41586-018-01414-6
[4] https://www.nature.com/articles/nphys3313
[5] https://peerj.com/collections/50-practicaldatascistats/
[6] https://twitter.com/hashtag/datascience?lang=en
[7] https://www.listennotes.com/search/?q=data%20science&scope=podcast
[8] https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0202447

science is in the midst of a reproducibility crisis[9]. There are also hackers trying (and succeeding) at making every scientific journal article free[10], and large universities dropping their multi-million dollar annual subscription contracts with peer-reviewed publishing companies[11] because of their cost. The tools we will cover in this manual can help combat each of these issues and improve the quality of the science you're doing.

Collaboration and reproducibility in academia

Broad visibility has always been difficult for academic scientists. As the quote above alludes to, it's entirely possible a college hiring committee won't see the value in maintaining a blog, developing open-source software packages, or sharing your code online. But you can take some comfort knowing science eventually course corrects, and you're just ahead of the curve.

It's also become clear that the traditional advice of "*publish or perish*" has lead to creating a confusing mountain of papers[12], and encouraged citation metrics which are easily gamed and manipulated[13].

The precise details of how scientific productivity and contribution gets measured in the future remain somewhat uncertain, but we're confident the tools in this manual will provide you with a dependable set of skills that can you can apply across a wide range of fields and professions.

Research in business

This section outlines how companies do research and science, the importance of teamwork, and how having a data-driven decision framework has resulted in better products and services. The general distinction between academia and business is that academics typically use statistics and science to generate knowledge about the world at large, and businesses apply these tools to solve problems related to the vision/mission of the company.

The difference between these two cultures is captured well by the Google engineer Cassie Kozyrkov in her fantastic article, "Never start with a hypothesis"[14],

> "*Statistics is the science of changing your mind under uncertainty, so the first order of business is to figure out what you're going to do unless the data talk you out of it.*"

In business, the results of an experiment or model are often used to drive decisions that will result in either profits or losses for that company, which is why people tend to label these methods "applied data science."

[9] https://www.nature.com/news/1.19970
[10] https://www.sciencealert.com/this-woman-has-illegally-uploaded-millions-of-journal-articles-in-an-attempt-to-open-up-science
[11] https://www.vox.com/the-highlight/2019/6/3/18271538/open-access-elsevier-california-sci-hub-academic-paywalls
[12] https://www.nature.com/news/the-top-100-papers-1.16224
[13] https://academic.oup.com/gigascience/article/8/6/giz053/5506490
[14] https://towardsdatascience.com/hypothesis-testing-decoded-for-movers-and-shakers-bfc2bc34da41

It's important to note that not every problem in business can (or should) be answered using models or data. Many times the existing knowledge of a business's goals, employees, products, and customers can be a better guide for decision-makers.

The largest companies in the world are using data and research to drive their decisions, and it's safe to bet that the companies in the future will rely on data even more. According to Nasdaq[15], Amazon and Alphabet (Google's parent company) spent more than any other company in 2017 Amazon spent over $26 billion, Google spent over $16 billion. But it's not only internet companies investing in research. Volkswagen spent over $15 Billion dollars on research into virtual technology, automation, and lightweight construction in 2017.

Whether it's an internet company like Google, Facebook, or Amazon, or even a retail giant like Walmart, all of these companies have taken some form of raw data and turned it into something valuable for their bottom line.

Collaboration and reproducibility in business

Data has been the raw material used to create business opportunities and products previous generations hardly imagined possible. Google, Netflix, Airbnb, Uber, etc. are all companies who lifeblood is data. They use data to identify customers, and experiment on new features and products, and run simulations on different business decisions.

Time is money in the world of business, so the quicker you can go from identifying the problem to giving possible solutions, the better. Collaboration is a huge advantage here because challenges tend to be somewhat similar for businesses in the same market, or even within departments in the same organizations. Sharing data analytic methods and resources between marketing/sales and customer service allows an organization to reduce the time it takes to develop these from scratch each time.

Data science in business needs to be fast-paced because the landscape is always changing (i.e., new competitors emerge, expanding your products/services into new markets, etc.). Reproducibility is essential for businesses because it's not likely the results of an experiment will remain the "truth" indefinitely, a point articulated by Stuart Buck in his article, "Why Your Company Needs Reproducible Research"[16],

> - "Even if an experiment works on the first, second, or even third tries, those original experiments may "expire," so to speak, if only by the passage of time and the shifting composition or tastes of users. It is statistically—and economically—unwise to assume that an experiment showing a 1% increase in revenue can extrapolate to an entire business unit and remain a 1% increase for all time."*

The data science work in businesses is also typically done in teams–having someone to bounce ideas around with and refine questions is essential to delivering the right information to decision-makers.

The following quote is from Graham J. Williams' text, "The Essentials of Data Science"[17],

[15] https://www.nasdaq.com/article/6-companies-spending-the-most-on-rd-cm1004333
[16] https://towardsdatascience.com/why-your-company-needs-reproducible-research-d4a08f978d39
[17] https://essentials.togaware.com/

> ” *A data scientist brings to a task a deep collection of computer skills using a variety of tools. They also bring particularly strong intuitions about how to tackle complex problems. Tasks are undertaken by resolving the whole into its parts. They explore, visualize, analyze, and model the data to then synthesize new understandings that come together to build our knowledge of the whole. With a desire and hunger for continually learning we find that data scientists are always on the lookout for opportunities to improve how things are done–how to do better what we did yesterday.* “

Leaders (CEOs, managers, etc.) in business need to make decisions quickly because the cost of doing nothing is sometimes higher than the price of making a sub-optimal decision. Many times, presenting your findings to a decision-maker will help your organization throw away the bad ideas quickly so you can move towards a better solution. This manual will provide a skillset and workflow that is adaptable to any company or organization using numbers to help solve problems and make decisions.

Sharing your work

> “*Your work should speak for itself...*” - author unknown

Now that we’ve covered the benefits of having a reproducible collaborative workflow, we can discuss why it’s crucial to make your work discoverable.

There is a sea of information on the internet, and that means everyone is competing for attention. If you want to share all the excellent work you’re doing (writing code, creating visualizations, building models, etc.) it needs to be easily discoverable.

If a future collaborator, prospective employer, or upcoming analysts is going to see what you’ve been up to, that means keeping an ongoing catalog of what you’ve done (*and unless you’re willing to pay for advertisements, that means creating useful content that people read and share*).

Some examples

We will introduce you to a few technologies, methods, and mediums used by successful data scientists who are excellent communicators of their work. These data scientists apply the internet as a tool to engage with broader audiences, create better tools for doing science, document some of their daily struggles/successes, and share more about what it means to conduct research.

For example, Lucy D’Agostino McGowan[18] is a post-doc at Johns Hopkins Bloomberg School of Health. She maintains a

[18] https://www.lucymcgowan.com/

blog[19], publishes ebooks[20], has online courses[21], and also attempts to create a real BB-8[22]. Her work is *highly discoverable* and showcases a wide range of skills.

Ricardo Bion is the Data Science manager at Airbnb[23]. He publishes papers[24] on using R in their business setting, gives webinars[25] on how to use modeling to make decisions, and writes articles[26] on workflow practices that contribute to success in their data science teams.

Or take Thomas Lin Pedersen, a former bioinformaticist who now designs software. His graduate research was on tools to analyze hierarchical pangenome data[27], which he turned into a tool[28], made the code free[29], and published[30] his work in a scientific journal. He's also an artist[31].

All of these data scientists have done two things very well: they've collaborated with the data science community (by putting their work online for people to find), and they made their process reproducible for us to adapt and use as we see fit.

Of course, they had to know their subject areas, and have something worth sharing online, but they didn't wait until they were done with their careers and write a book. They started engaging with people while they were working to show how their work gets done.

Communication = thinking with words

In this section, we will briefly go over some suggestions for communicating your work. It's important to remember that every time you're trying to communicate something (a figure, table, important finding, etc.), you're convincing that person they should be listening/reading to what you have to say *instead of anything else they could be paying attention to.* Approaching communication this way puts you in the mind of your audience and keeps you asking, "*why would they want to know this?*"

[19] https://livefreeordichotomize.com/
[20] https://leanpub.com/ggplot2in2
[21] https://leanpub.com/u/lucymcgowan
[22] https://magazine.amstat.org/blog/2017/11/01/lucy-dagostino-mcgowan-and-ryan-jarrett/
[23] https://t.co/EaT2pX2wWm?amp=1
[24] https://peerj.com/preprints/3182/
[25] https://www.rstudio.com/resources/videos/airbnb/
[26] https://medium.com/airbnb-engineering/using-r-packages-and-education-to-scale-data-science-at-airbnb-906faa58e12d
[27] https://vimeo.com/181004000
[28] https://www.data-imaginist.com/panviz/
[29] https://github.com/thomasp85/PanViz
[30] https://www.ncbi.nlm.nih.gov/pubmed/28057677
[31] https://www.instagram.com/thomasp85_/

Avoid technical jargon & acronyms

> "*You must learn to talk clearly. The jargon of scientific terminology which rolls off your tongues is mental garbage.*" - Martin H. Fischer

The most substantial barrier to understanding new disciplines or technologies is getting a handle on their jargon. Because this manual sits at the intersection of computer science, statistics, and web technologies, all the new vocabulary can often seem like learning a foreign language.

Wherever possible, we'll do our best to clear up or define any terms related to computer science, data management system, web technology, or statistics. To maximize the power of the tools in this text, it will help to know a little about their history, so we'll also cover some background.

Science communication takes practice, but it's worth it!

No one is born with an ability to write well–it takes a lot of practice and feedback. The more you communicate with different audiences about your research, the better you'll get at finding an ability to convey its importance.

Communicating results should be one of the most important goals for data scientists. After we left graduate school, we realized how few people had the skills we were taking into the world (and how many people would benefit from them). But small improvements in communication can improve someone's perception of a topic they were previously unaware of, which is what makes math and science so attractive–small investments in understanding can yield substantial returns.

The best science writers capture their audience by weaving science into a compelling narrative. Carl Sagan, Mary Roach, Freeman Dyson, Jared Diamond–all of these authors have a unique talent for making complicated, intricate scientific topics enjoyable by engaging us with the people in the story. By entering graduate school and doing your research, you're now one of the people in the story, contributing to the research topic. It's your job to tell your portion of the story.

Remember, **science is a method**, **not a product**. That process of how you found what you found is the most critical part of your research because it's the part that tells us 1) why we can trust what you published, and 2) how we can try to reproduce your findings.

When we were kids in math class, the teacher would ask us to "show our work." Teachers gave these instructions so they could follow our thought processes through a problem, and see where our thinking was incomplete or mistaken (and probably also to make sure we weren't looking at someone else's paper). If you're regularly showing your work, you'll be able to follow your line of thinking as you progress throughout your career. More importantly, people who find your work will see you're an actual human, with a full range of human experiences, and much more than a CV, résumé, or headshot.

Being good enough

We encourage being 'good enough' at math and science vs. becoming an expert. Expertise takes significant time, energy, resources, and luck to attain. Being 'good enough' means reading about a tool or technology and being capable of distinguishing it from magic. Good enough means seeing a chart or number on the news and knowing how to evaluate its contents. Or imagining something that matters to your business or personal life, and then devising a way to count it.

We think an emphasis on a 'good enough' understanding of math and science makes both topics more practical and useful to everyday life. We hope this approach helps lessen the "us vs. them" mentality that can arise when science makes it's way into the public sphere because most people are using math to some degree every day (they probably don't notice). In our experience, experts don't need encouragement.

We sincerely hope you'll find this information useful and give us feedback at `mfrigaard@paradigmdata.io` or `pspangler@paradigmdata.io.`

Footnotes

- The scientific journal industry is not looking out for your best interests. They have a clearly unethical business model[32], and other prominent universities can't afford their prices[33]. And they won't compensate[34] you for your efforts.
- The metrics previously used to measure success in academic publishing are unreliable and susceptible to being gamed[35]. You don't want to have these be your sole measure of productivity.
- John Ioannidis has led the charge in pointing out some of the ways science publication is flawed. Massive citations to misleading methods and research tools: Matthew effect, quotation error, and citation copying[36].

Here are more perspectives on the reproducibility crisis and publishing practices.

* Opinion: Is science really facing a reproducibility crisis, and do we need it to?[37]

* Publish or Perish: Is Milton's Paradise Lost on Academia?[38]

[32] https://www.theguardian.com/commentisfree/2011/aug/29/academic-publishers-murdoch-socialist
[33] https://www.theguardian.com/science/2012/apr/24/harvard-university-journal-publishers-prices
[34] https://whyevolutionistrue.wordpress.com/2011/09/01/the-racket-of-academic-publishing/
[35] https://academic.oup.com/gigascience/article/8/6/giz053/5506490
[36] https://link.springer.com/article/10.1007/s10654-018-0449-x
[37] https://www.pnas.org/content/115/11/2628
[38] https://areomagazine.com/2018/10/09/publish-or-perish-is-miltons-paradise-lost-on-academia/

3. Part 2: “Have a workflow.”

A workbench is a place to keep and organize tools, and workflow is how you combine these tools to get things done. This chapter will cover the workbench we use and the three guiding principles of the workflow we recommend.

1. Use free open source software
2. Write code
3. Document everything in plain text

Principle 1: Use open-source software

All of the tools in this book are available open-source and available free of charge. Just as a point of reference, the cost of a subscription to SPSS at the time of this writing is $99.00 per user per month. Stata is $595 per year or $1,595 for a perpetual license. There are educational discounts available, but this cost is not offset by much when you take into account the rising price of tuition.

A more important reason we recommend open source tools are the communities that you’ll get access to when you start using them. By entering the universe of open source software, you get to take advantage of seeing problems solved in the open. You’ll also find people like you, grappling with the same issues, and it’s hard to overstate the benefit of this shared camaraderie.

The final reason is philosophical: we all benefit from using open source tools and sharing improvements on them together. The ‘four freedoms’ of open source software[1] captures this sentiment below.

Freedom 0: The freedom to run the program as you wish, for any purpose.

Freedom 1: The freedom to study how the program works, and change it, so it does your computing as you wish.

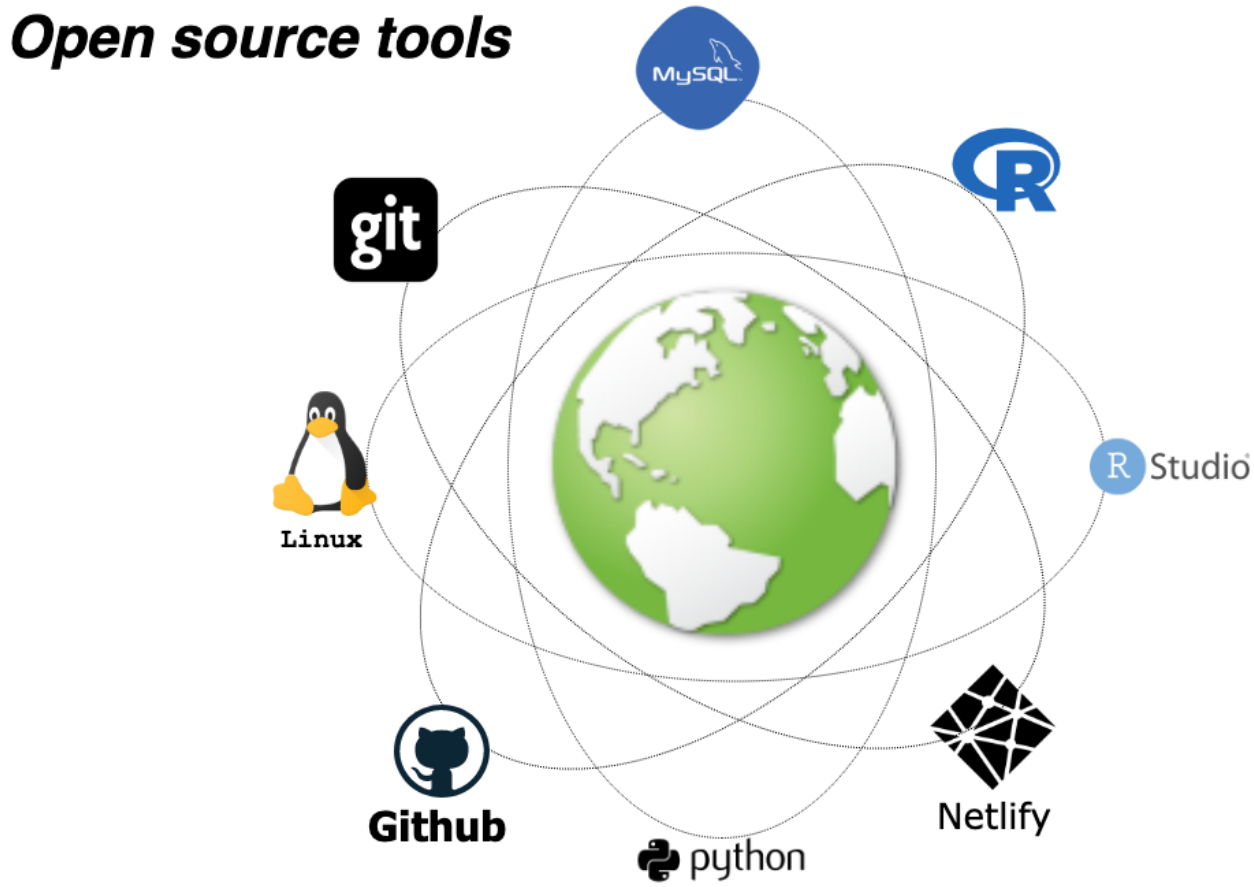
Freedom 2: The freedom to redistribute copies so you can help your neighbor.

Freedom 3: The freedom to distribute copies of your modified versions to others. By doing this, you can give the whole community a chance to benefit from your changes.

We also think it’s onerous to require graduate students (and other scientists) to purchase proprietary software licenses to participate in science.

[1] https://www.gnu.org/philosophy/free-sw.html

We've displayed some examples of open source tools for data management, statistics, and communication in the image below:



Follow the links below to learn more.

- Git[2]
- Github[3]
- Linux[4]
- MySQL[5]
- Netlify[6]

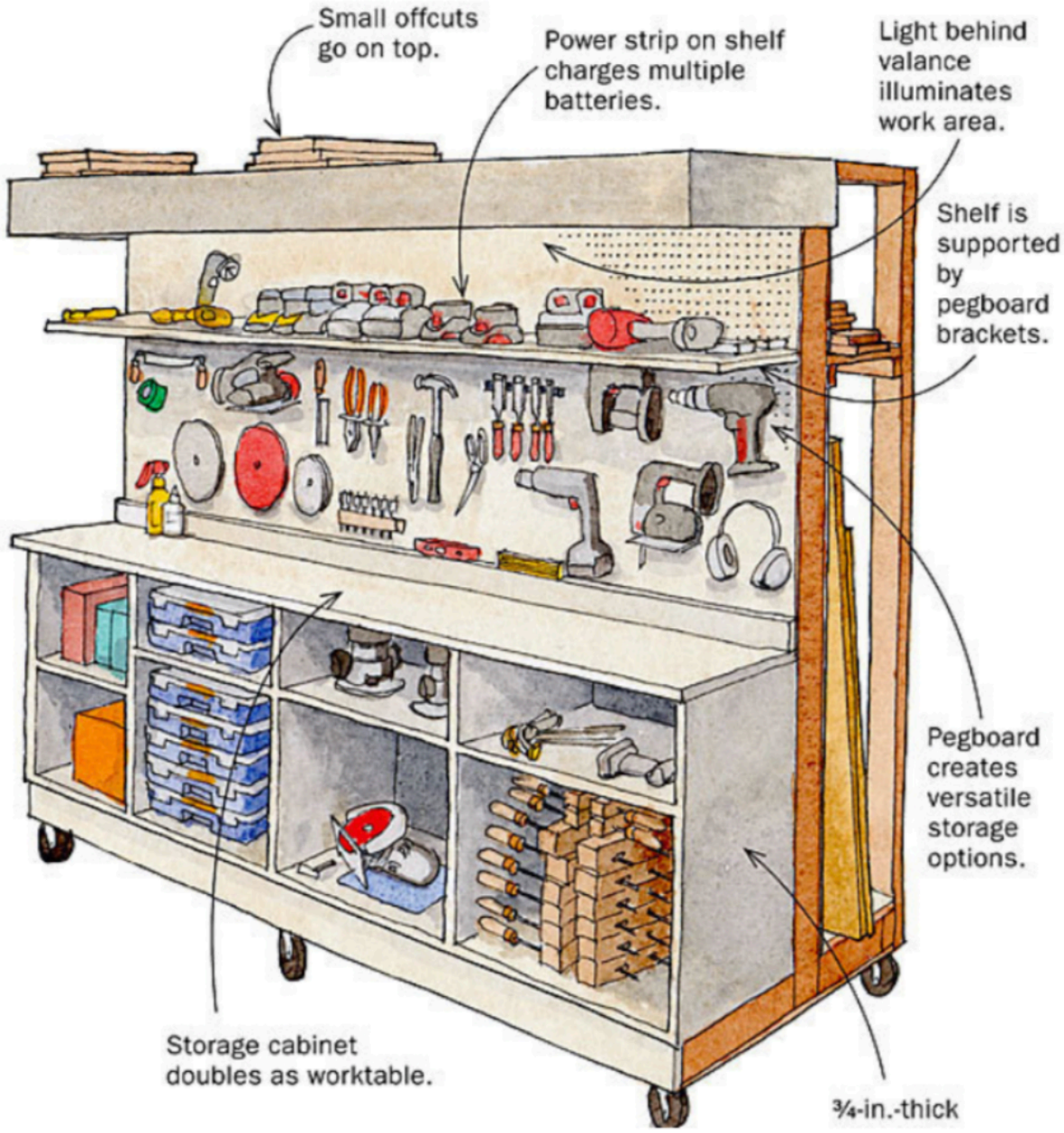
[2] https://git-scm.com/
[3] https://github.com/
[4] https://www.linux.org/
[5] https://www.mysql.com/
[6] https://www.netlify.com/

- Python[7]
- R[8]
- RStudio[9]

The integrated development environment (aka data science workbench)

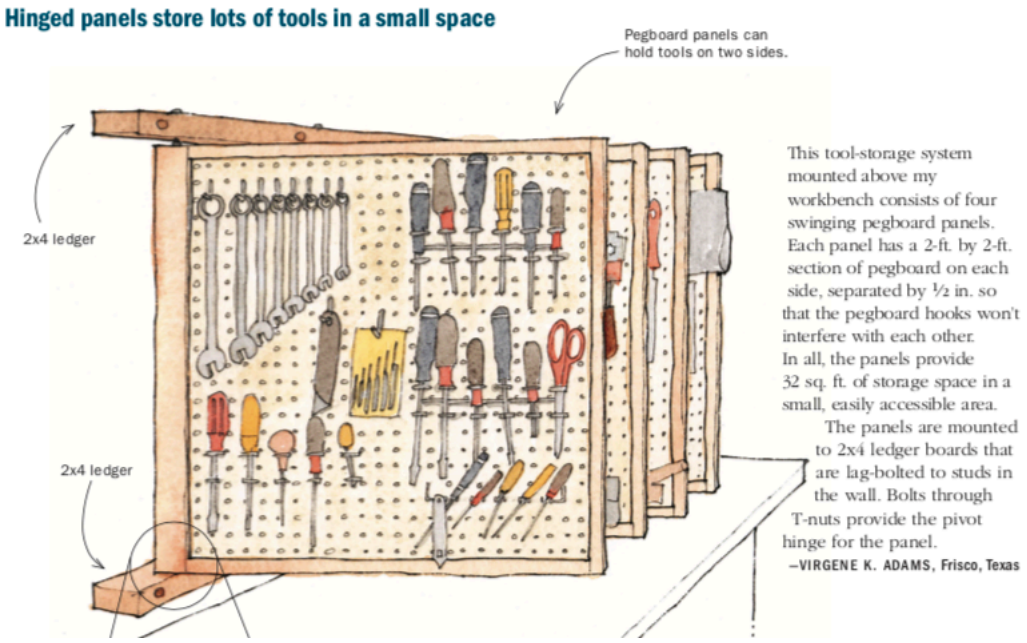
An integrated development environment[10] (IDE) is an application typically used by programmers to build and test software. We find it helpful to think of a woodworkers workbench as an analogy. The image below is an example of a workbench with a simple rolling cart design (for people with minimal garage space).

[7] https://www.python.org/
[8] https://www.r-project.org/
[9] https://www.rstudio.com/
[10] https://en.wikipedia.org/wiki/Integrated_development_environment



source: https://www.finewoodworking.com/2008/12/11/lighted-storage-cart-for-tools-and-lumber

As we can see, this workbench is efficiently designed to keep essential tools for the job within arms reach, and it uses storage space efficiently. IDE design follows these same principles. However, we also know some models are better than others. For example, consider the design of a different workbench below.



source: https://www.finewoodworking.com/2010/11/12/free-plan-space-saving-tool-rack

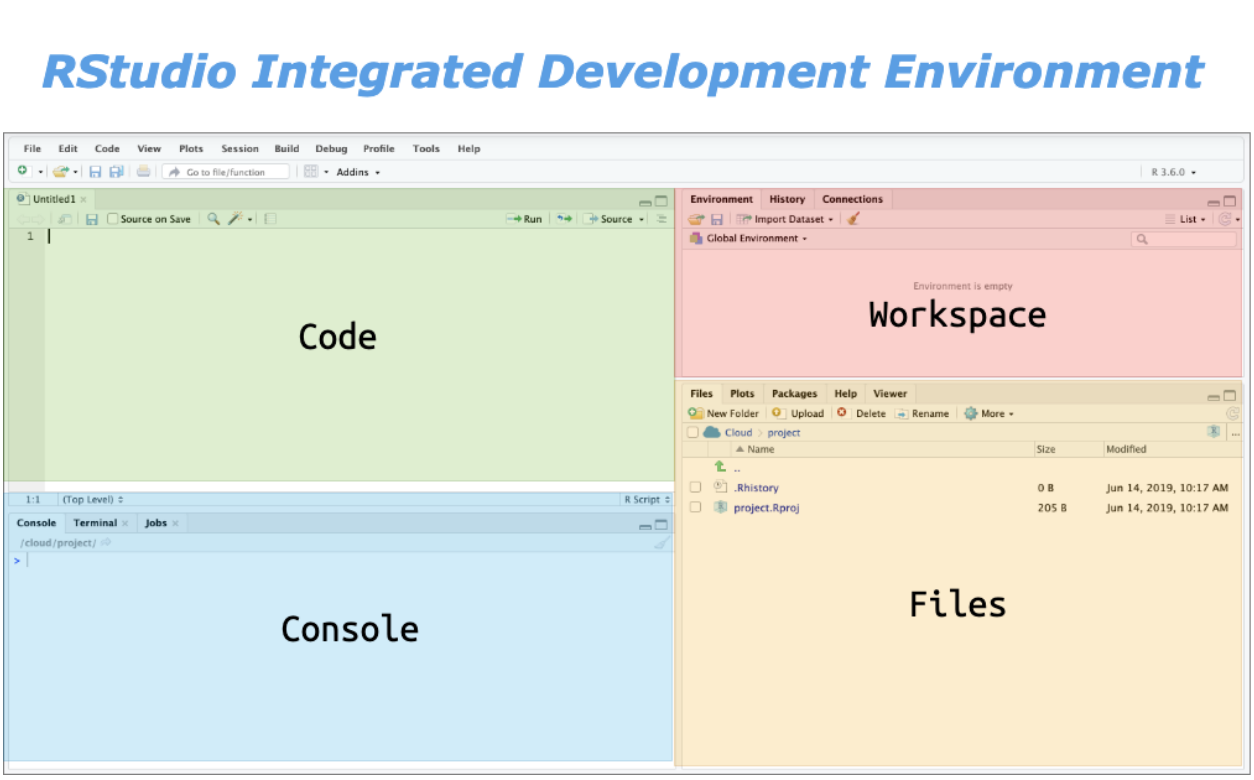
By making the panel positions adjustable, the workbench allows for easier access to more tools. Depending on the job and tools required, the woodworker can customize the panel arrangement as they see fit (or as the website describes it, "*the simple pin that allows the rack's various faces to swing left/right for access to either side.*")

These examples illustrate how differences in the design of a workbench can have a meaningful impact on levels of productivity. Well-designed workbenches give us access to more tools without making these tools more difficult to find.

Our workbench

We recommend choosing a workbench that minimizes the number of additional applications you'll need to have open to get work done. We've found we can use RStudio[11] for ~90% of our daily work (* they're not paying us to say this*). RStudio gives us access to all the tools we need in the same place.

[11] https://www.rstudio.com/products/RStudio/



RStudio has four primary panes, each serving a specific function.

- the **Code** or **Source** pane is where we can document both human-readable and computer-readable text
- the **Workspace** holds the data, functions, and other data analysis objects
- the **Console** displays the results from our written code (and allows us to enter commands directly)
- the **Files** gives us access to the happenings outside the RStudio environment (imported raw data, exported results, etc.)

Just like any workbench, we need to fill RStudio with the proper tools. RStudio plays well with many different tools, but for now, we are just going to focus on R and Git.

What is R?

R is a free statistical modeling software[12] application and language. If you are using the desktop application, follow the links below to install R.

Installing R & RStudio

1. First, you'll need to download and install R from CRAN[13].

[12] https://www.r-project.org/
[13] https://cran.r-project.org/

2. Second, download and install RStudio[14], the integrated development environment (IDE) for R

Working in the browser

1. An alternative to downloading and installing R and RStudio is using RStudio.Cloud[15] which operates entirely in your browser. You'll need to sign up for RStudio.cloud for free using your Google account or email address, but we recommend using a Github account. You can create a Github account here[16].

Open-source software bonus: As mentioned previously, you'll also find a massive network of support on Stackoverflow[17], RStudio Community[18], and Google Groups[19].

What is Git?

Git[20] is a version control system (VCS). VCSs are used to track changes to projects with code. You can read more about Git in their online text here[21].

What is Github?

Github[22] is the web-based hosting service for Git. You should set up a free an account with Github here[23].

What do Git/Github do?

We will cover more on Git/Github in later sections, but for now, know these tools will allow you to keep track of changes to your project over time.

Note: *You should explore different IDE's on your own– you'll see there are many options, both paid and unpaid. We're confident you'll see RStudio is well suited to handle more than most of the things you'll want to accomplish.*

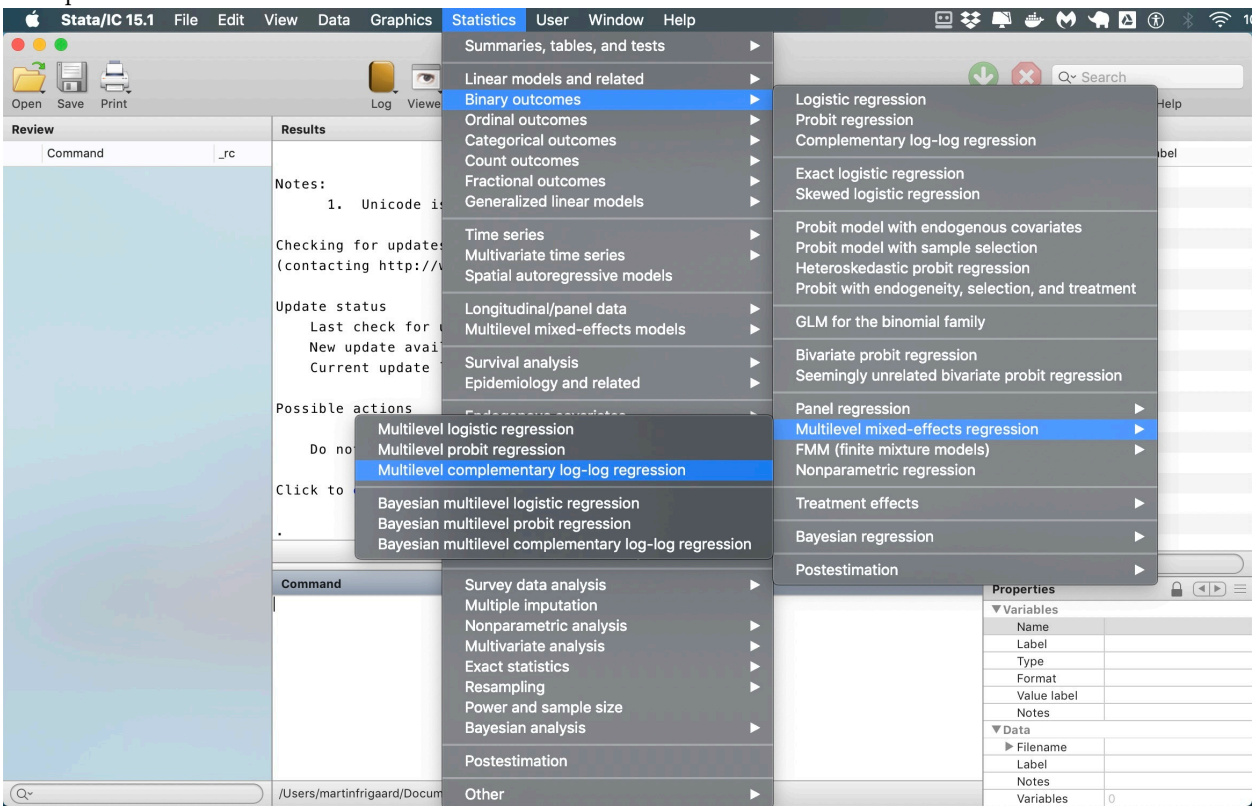
---

[14] https://www.rstudio.com/products/rstudio/download/
[15] https://rstudio.cloud/
[16] https://github.com/join
[17] https://stackoverflow.com/questions/tagged/r
[18] https://community.rstudio.com/
[19] https://groups.google.com/forum/#!forum/r-help-archive
[20] https://git-scm.com/
[21] https://git-scm.com/book/en/v2
[22] https://github.com/
[23] https://github.com/join?source=header

Principle 2: Write code

Usually, people interact with their computers using point-and-click graphical user interfaces[24] or GUIs (pronounced 'gooey'). GUIs are quick and easy to learn because their design environment usually mimics an actual physical space (i.e., desktops, folders, or documents). GUIs are a mostly positive development because designing software with a more user-centered design[25] is one of the main reasons technology adoption has been on the rise for the past 20+ years.

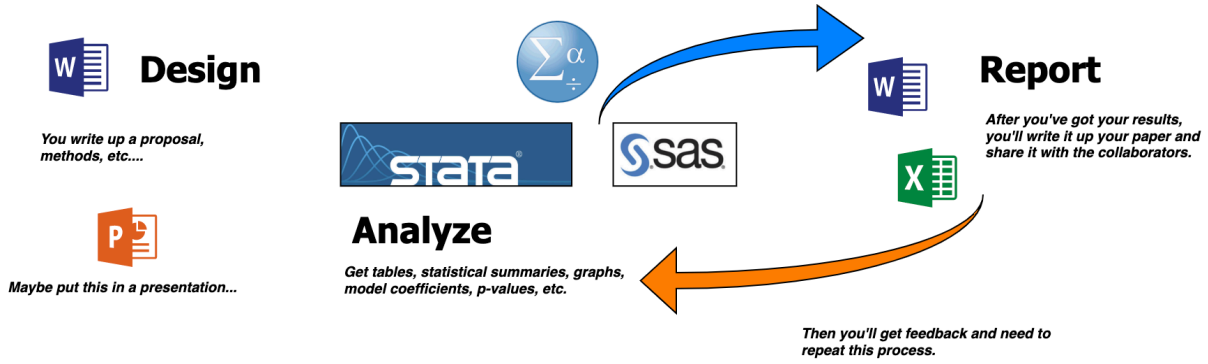
User-centered-designed software includes most of the point-and-click operating systems and applications. These programs give the user the ability to click through a predetermined list of options and procedures using their mouse or track-pad.



However, we think there are times when we should resist the temptation to abstract away some of life's complexity, and data science is one of them. Using applications like these encourage a copy + paste workflow like the one below.

[24] https://en.wikipedia.org/wiki/Graphical_user_interface

[25] https://en.wikipedia.org/wiki/User-centered_design



As you can see from the image when you receive feedback or input to your project, you'll be required to go back through the same elaborate procedure (with each time taking just as long as the first).

We recommend an alternative to the copy + paste workflow based on the activities of a modern scientist from Jeff Leek we outlined in Chapter 1:

1. Develop code in the open
2. Publish data and code open source
3. Post preprints of your work
4. Submit and review for traditional journals
5. Blog or use social media to critique published work

Two things should stand out from the list above: First, modern science is mostly writing. Second, some of that writing is code (i.e., programming). It's, for this reason, we recommend adopting a workflow based on "*writing code*" wherever possible. We are aware that 'writing code' means being able to type, which might be daunting for people who struggle on a keyboard. We recommend practicing this skill (there are plenty of great apps out there to help!) because typing is an unavoidable necessity for using a computer.

Software isn't a solution

An extension of the "*writing code*" philosophy is to view data science software applications as tools for gaining a deeper understanding of the world (but not oversimplifying or obfuscating it).

As we stated in chapter 1, the scientific method is a process. Software is a tool to help move that process along faster. Tools that improve our understanding shorten the distance between questions and answers, but doesn't leave out any crucial details.

For this reason, we recommend avoiding point-and-click environments in most proprietary software applications (SPSS, Stata, SAS, etc.). It's hard to keep track of everything you click on (or the order of you clicked on them in) inside these environments, and this makes it hard to know what step you're currently doing.

What do we mean by 'a workflow'?

A workflow is a set of steps that can be used repeatedly to answer any question you might encounter in your work.

The quote below is from an interview with Andrew Gelman, a statistician from Cornell, who is an author on the excellent blog Statistical Modeling, Causal Inference, and Social Science[26].

> **Question**: "I'm wondering how you, as an educator and statistician, would like to see statistical and data literacy change in general for a general population?"
>
> **Answer**: "... I've come to realize that a lot of people don't even know what they did. People don't have a workflow, they have a bunch of numbers, and they start screwing around with the numbers and putting calculations in different places on their spreadsheet, and then at the end, they pull a number out and write it down and type it into their report." - Andrew Gelman[27]

As the quote above illustrates, how you got an answer is just as relevant as the answer you got. The tools we provide in this text give you a start-to-finish chain of documentation from question to solutions.

Recap: data science jobs need a particular set of tools, and a workbench to organize these tools. Data science jobs also have a lot of moving parts, and in order manage that complexity, you'll need a workflow that gives you the ability to 1) write down what you want to do and how you want to do it, and 2) write code that translates those thoughts and intentions into something a computer can execute. These two points bring us to our next topic: documentation. As you'll discover, plain text files are a great way to accomplish these tasks.

Principle 3: Document everything in plain text

In The Pragmatic Programmer[28], authors Hunt and Thomas advise *'Keep[ing] Knowledge in Plain Text'*. This sentiment has been repeated here[29], here[30], and here[31].

[26] https://statmodeling.stat.columbia.edu/
[27] https://soundcloud.com/dataframed/election-forecasting-polling
[28] https://www.amazon.com/Pragmatic-Programmer-Journeyman-Master/dp/020161622X
[29] https://simplystatistics.org/2017/06/13/the-future-of-education-is-plain-text/
[30] https://richardlent.github.io/post/the-plain-text-workflow/
[31] http://plain-text.co/index.html#introduction

We recommend you keep track of your changes, notes, and any pertinent documentation about your project in plain text `README` files. The reasons for this will become more apparent as we move through the example, but I wanted to outline a few here:

- plain text lasts forever (files written 40 years ago are still readable today)
- plain text can be *converted* to any other kind of document
- plain text is text searchable (ctrl+F or cmd+F allows us to find keywords or phrases)

These all sound great, but you might still be wondering what makes a file 'plain text,' so we'll define this below. This chapter will also cover why you might want to consider switching over to a plain text editor if you're currently using something like, Google Docs, Apple Papers, or Microsoft Word.

Wait–why would I change what I'm doing if it works?

We get it–change is difficult, and if you have a working ecosystem of software that keeps you productive, don't abandon it. However, you should be aware of these technologies and recognize that people using them will be adapting *their* workflows to collaborate with you.

We covered the problems with a copy+paste workflow previously, but there are additional reasons to avoid this toolset:

1. It's not reproducible
2. It's not logical or necessarily honest to separate computation from the analysis or presentation
3. It's error-prone

What *isn't* plain text

Non-plain text files are usually called binary (i.e., files with binary-level compatibility) need special software to run on your computer. The language below is a handy way to think about these files:

> "Binary files are *computer-readable but not human-readable*[32]"

What *is* plain text

So if binary files aren't plain text, what is a plain text file? The language from the Wikipedia[33] description is helpful here:

[32] https://www.webopedia.com/TERM/B/binary_file.html
[33] https://en.wikipedia.org/wiki/Text_file

When opened in a text editor, plain text files display computer and human-readable content.

And here is the most crucial distinction–**human-readable vs. computer-readable**. I'll be sure to point out which files are binary and which are plain text as we go through the example, but generally speaking, a plain text file can be opened using a text editor. Examples of text editors include Atom[34], Sublime Text[35], and Notepad++[36]

Markdown & Rmarkdown

A common type of plain text file is a markdown file, or `.md` file. Markdown has a straightforward syntax that's easy for both humans and computers to read, and it allows for some formatting options to aid with communication (see Markdown Syntax Documentation[37] on John Gruber's site).

RStudio has an extension of markdown, RMarkdown[38]. Using RMarkdown in RStudio allows for a genuinely reproducible workflow because you're able to write your thoughts, code, display results, and then share everything in multiple outputs.

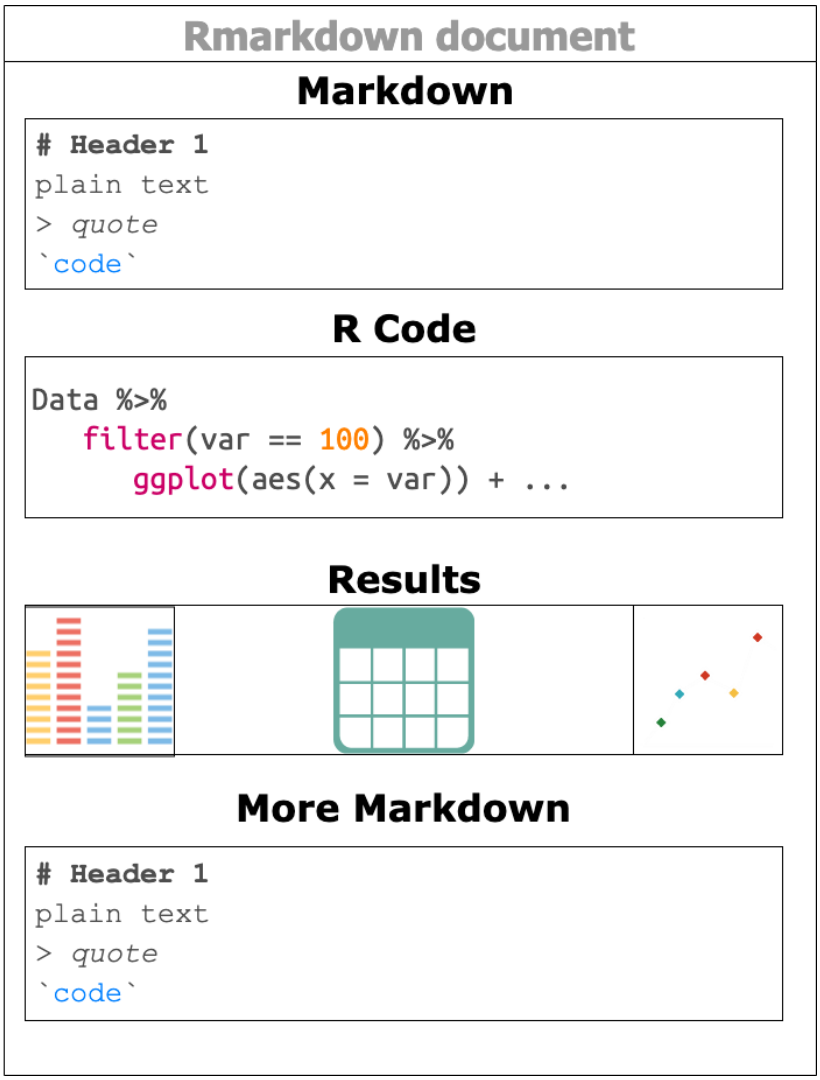
[34] https://atom.io/
[35] https://www.sublimetext.com/
[36] https://notepad-plus-plus.org/
[37] https://daringfireball.net/projects/markdown/syntax
[38] https://rmarkdown.rstudio.com/

Words for humans to read (prose, expectations, predictions...)

Code that translates our thoughts into something the computer can read

Imported data, summary tables, graphs, model results, etc.

Output to interpret, explain, makes sense of, generally ponder over, etc.



I recommend reading up on R and RMarkdown because of how many different outputs this combination can be used to produce (.pdf, .docx, and .html). Consult the R Markdown: The Definitive Guide[39] for more information. The image below is an output from an `.Rmd` document in RStudio.

[39] https://bookdown.org/yihui/rmarkdown/

Rmarkdown Document

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

Load the `tidyverse`.

```
## ● __  _     __    .    ○          ●  .
##  / /_(_)__/ /_ ___  _____ _______ ___
## / __/ / _  / // / |/ / -_) __(_-</ -_)
## \__/_/\_,_/\_, /|___/\__/_/ /___/\__/
##      ●  . /___/      ○      .      ●
```

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

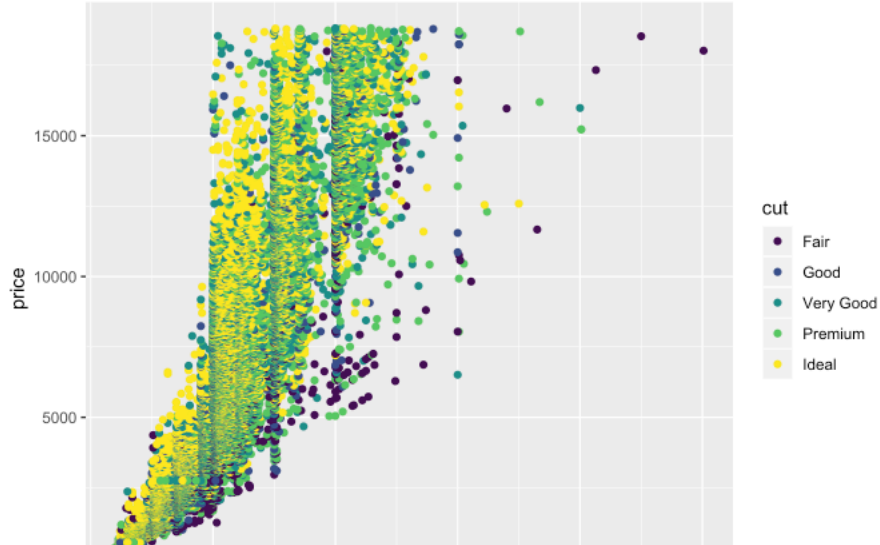
```
ggplot2::diamonds %>% glimpse(78)
```

```
## Observations: 53,940
## Variables: 10
## $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23,…
## $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very Good, …
## $ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, J, J,…
## $ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI1, VS…
## $ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, 59.4,…
## $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54, 62,…
## $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, 340,…
## $ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, 4.00,…
## $ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, 4.05,…
## $ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, 2.39,…
```

Including Plots

You can also embed plots, for example:

```
diamonds %>%
    ggplot(aes(x = carat,
               y = price,
               color = cut)) + geom_point()
```



Python vs. RStudio

Python is a great language, and it can do a broader range of computational tasks than R. I would never tell a researcher or scientist that Python is something they shouldn't learn (the benefits of being multilingual extend beyond just spoken languages, too).

We recommend R/RStudio because we wrote this book for people who have a data file and specific questions (or general curiosity). Thus, the entry point for our audience into data science is *with data they need to analyze*, and this is what R was made to do.

Additional reasons for using R/RStudio

Below are a few more reasons you should consider using R/RStudio in case you're still on the fence.

You can focus on your work

> '*The only factor becoming scarce in a world of abundance is human attention*" – Kevin Kelly in Wired

We recommend R/RStudio because of the time saved by switching between software applications. For example, when I was in graduate school, I had to have *a minimum of five applications open* to do my daily work of data analysis (MS Word to write, MS Excel to create tables, Stata for statistics, the browser for internet research, and Adobe for reading .PDFs).

Five different GUIs, each with their design characteristics, and each costing me valuable neurons every time I had to switch between them (read more about attentional residue in the footnotes). With R/RStudio, I cut this number to two (RStudio and the browser).

RStudio gives you a better mental model for data analysis

The third reason is the design of the IDE itself. RStudio is a complementary cognitive artifact, something described in this article from David Krakauer[40],

> "*[complementary cognitive artifacts are] certainly amplifiers, but in many cases, they're much, much more. They're also teachers and coaches…Expert users of the abacus are not users of the physical abacus—they use*

[40] http://nautil.us/blog/will-ai-harm-us-better-to-ask-how-well-reckon-with-our-hybrid-nature

> *a mental model in their brain. And expert users of slide rules can cast the ruler aside having internalized its mechanics. Cartographers memorize maps, and Edwin Hutchins has shown us how expert navigators form near symbiotic relationships with their analog instruments.*"

These are in contrast to competitive cognitive artifacts, which is what a GUI does.

> "*In the case of competitive artifacts, when we are deprived of their use, we are no better than when we started. They're not coaches and teachers—they are serfs.*"

RStudio does not remove the complexity of doing data analysis, writing blog posts, building applications, debugging code, etc. Instead, it creates an environment where you can do each of these tasks without having them abstracted away from you into drop-down menus, dialogue boxes, and point-and-click options.

There have been considerable efforts from the scientists at RStudio to create an environment and ecosystem of tools (called `packages`) to make data analysis less painful (and even fun). We're confident you'll find it helps you think about the inputs and outputs of your work in productive and creative ways.

FOOTNOTES

- The Ford Foundation report, "Roads and Bridges"[41], outlines some other reason you should be using open-source software.
- Read these articles on attentional residue and multitasking (then try to stop doing it).
  - Why is it so hard to do my work? The challenge of attention residue when switching between work tasks - ScienceDirect[42]
  - Information, Attention, and Decision Making[43]
  - Causes, effects, and practicalities of everyday multitasking[44]
- see Baumer et al.[45] for an in-depth summary of why you should abandon a copy + paste workflow
- This text is an *opinionated technical manual* anyone looking to get started with data science, visualization, reproducible reporting, dashboards development, or website/blog creation. We primarily recommend performing these tasks with R & RStudio (via RStudio.Cloud), and Git/Github. We're not saying there aren't other means or tools capable of accomplishing the same activities; these are the tools we've found success with, so they're what we recommend.

[41] https://www.fordfoundation.org/about/library/reports-and-studies/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure/
[42] https://www.sciencedirect.com/science/article/pii/S0749597809000399
[43] https://aom.org/uploadedFiles/Publications/AMJ/June_2015_FTE.pdf
[44] https://www.sciencedirect.com/science/article/pii/S0273229714000513
[45] https://arxiv.org/abs/1402.1894

4. Part 3: Project files and organization

Now that we've recommended a workbench (RStudio) and a set of tools (R, Git, Github), we'll use an example project to show how combining these tools create a durable and adaptive workflow. We want to get started with an example early because having a job to do allows us to cover project organization.

Our statistical coursework rarely covered how to set up a project setup (we often marvel at how much time we wasted trying to find our files). The way we set our projects up–how we organize files and folders–will directly contribute to our ability to be productive. You've probably discovered it's hard to get things done in a messy office? Well, it will be hard to do data science if we don't organize our files in a logical way that helps us get things done.

Example project: motivation for getting data

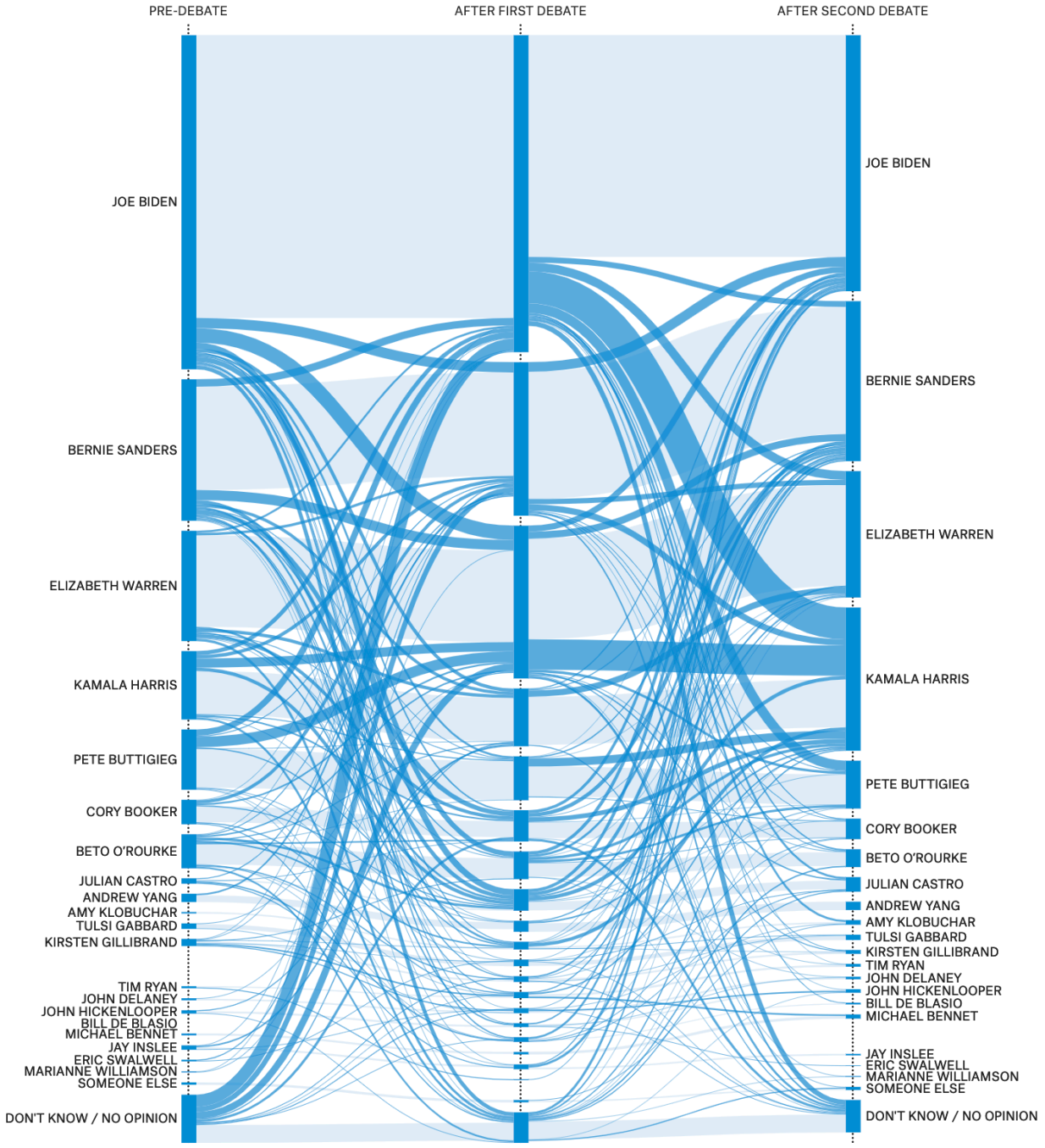
We read something on the internet, got curious, and decided we wanted to dig a little deeper.

The scenario I've described above might seem vague, but we want to show how powerful these tools can be, and we've found some of the most exciting data projects are born of basic curiosity.

In this case, let's imagine we read something about the first round of the 2019 Democratic Presidential Debates[1], but we missed all the news coverage.

We happened to read an article from the data journalism website `fivethirtyeight`[2], and it displayed an image showing how voters had changed their minds after seeing the candidates.

[1] https://en.wikipedia.org/wiki/2020_Democratic_Party_presidential_debates_and_forums
[2] https://projects.fivethirtyeight.com/democratic-debate-poll/



source: https://projects.fivethirtyeight.com/democratic-debate-poll/

Wanting to be informed citizens (and knowing how to collect and analyze data), we decide to investigate how each candidate performed using various sources of data.

Data journalism = "social science in real time"

Journalists are a bit like statisticians in the sense that both get to "*play in everyone's backyard*"[3]. Data journalism explicitly combines analysis and communication skills, which makes it a great place to look for tools and methods we can adapt to various projects.

Data journalists like Aleszu Bajak[4], Andrew Flowers[5], and Andrew Ba Tran[6] have been hugely influential in introducing R as a tool in the newsroom. The best data journalism pieces combine the rigor of numbers and math with an ability to **write something people will read**.

More importantly, journalists are trained to view the world differently than typical scientists or analysts. As the NBC investigative reporter Andy Lehren[7] describes in the text Digital Investigative Journalism[8],

> *"Journalists can approach data differently than those more trained in computer sciences. Take, for instance, matching databases. Traditional IT managers compare data sets that were designed to talk with each other. Journalists may wonder if the payroll list of school teachers includes registered sex offenders."*

More brains are better than one when it comes to looking at data, and that's usually because of the different types of questions that come out of these brains.

Modern data

To demonstrate how powerful R/RStudio can be, we are going to combine data from four different sources. Each source represents a different way to access data in using R + RStudio.

1) The `gtrendsR`[9] package for R gives us access to Google search terms and trends. We're going to use this to import data on Google searches for the candidates before and after the night of the debates.

2) `rtweet`[10] package in R can be used to download Twitter data but takes a few steps to get set up. Fortunately, we've written a tutorial here[11] and the package has excellent documentation (see here[12] and here[13]).

[3] https://www.nytimes.com/2000/07/28/us/john-tukey-85-statistician-coined-the-word-software.html
[4] https://twitter.com/aleszubajak
[5] https://twitter.com/andrewflowers
[6] https://twitter.com/abtran
[7] https://twitter.com/lehrennbc
[8] https://www.palgrave.com/gp/book/9783319972824
[9] https://github.com/PMassicotte/gtrendsR
[10] https://rtweet.info/
[11] http://www.storybench.org/get-twitter-data-rtweet-r/
[12] https://rtweet.info/articles/auth.html
[13] https://rtweet.info/articles/intro.html

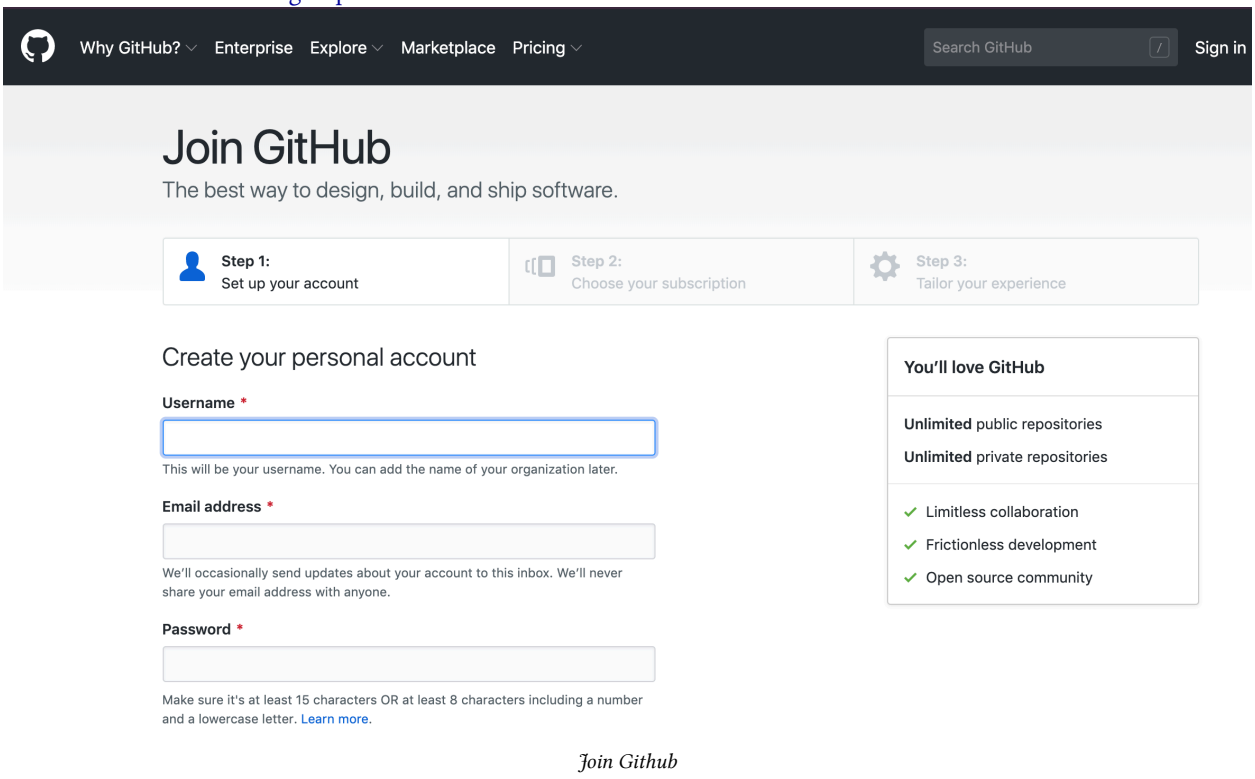
3) There is a Wikipedia[14] page dedicated to the debates. We'll be scraping the tables with airtime for a candidate using the `xml2`[15] and `rvest`[16] packages.

4) Finally, we also have some data from voters on how they felt about each democratic candidate going into the debates stored in a Google Sheet[17] that we've accessed using the `googlesheets`[18] package in R (*you will need to copy this sheet into your Google drive to get this data set*). Another option is to use the `datapasta`[19] package and copy + paste these data into R.

Step 1: Github

In this example, we will be using an RStudio.Cloud environment to perform the analyses. All of these steps can be accomplished using the RStudio IDE on your local desktop, too.

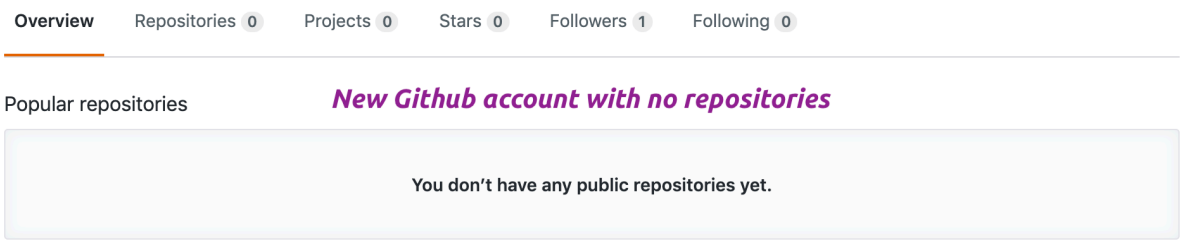
Head over to Github and sign up[20] for a free account.



Join Github

[14] https://en.wikipedia.org/wiki/2020_Democratic_Party_presidential_debates_and_forums
[15] https://cran.r-project.org/web/packages/xml2/index.html
[16] https://rvest.tidyverse.org/
[17] http://bit.ly/2YEVASu
[18] https://cran.r-project.org/web/packages/googlesheets/vignettes/basic-usage.html
[19] https://cran.r-project.org/web/packages/datapasta/README.html
[20] https://github.com/join

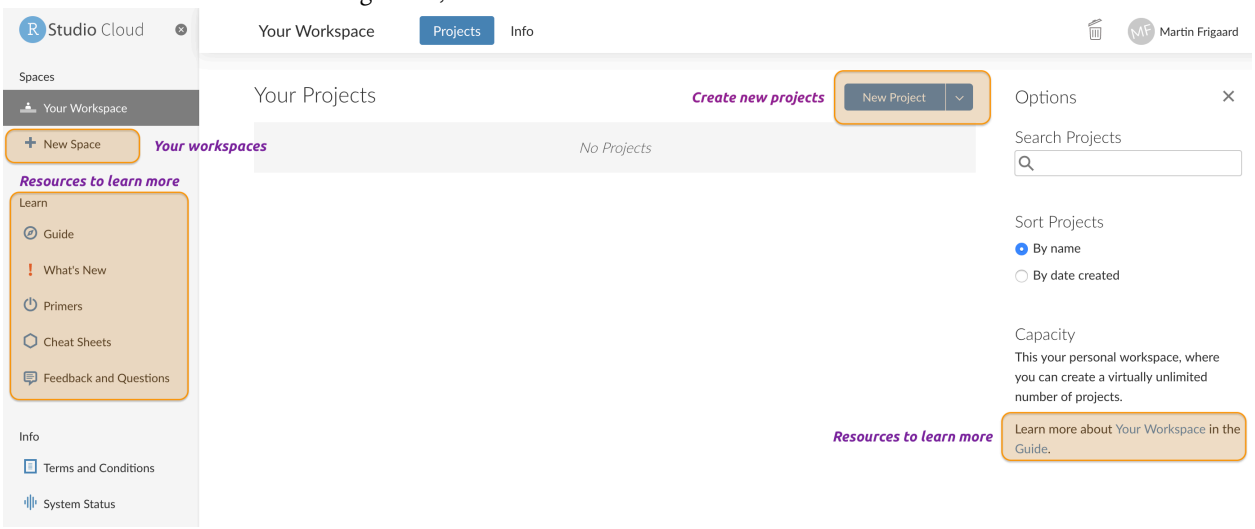
After you've completed the necessary forms (*remember you only need a free account!*), you should see a page with a message telling you "**You don't have any public repositories yet**".



Github profile overview

Step 2: RStudio.Cloud

We will eventually create our repositories, but for now, let's head over and use our Github account to sign into RStudio.Cloud[21]. After we're all signed in, we will see the screen below:



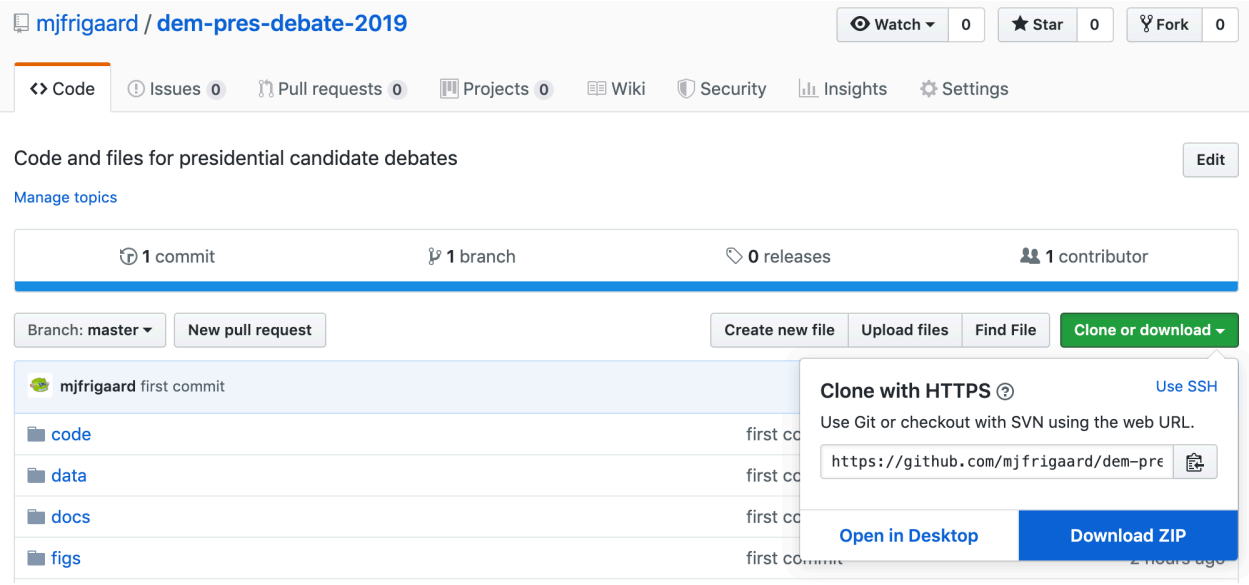
RStudio.Cloud environment

We've outlined the various resources, projects, and workspaces in the image above (we will go over each in more detail in a later section). For now, we are going to download a repository from Github and open it in RStudio.Cloud.

Step 3: Download a repository from Github

Most of the repos on Github are free for us to download and use. We can do this by clicking on the green **Clone or download** button and click **Download ZIP**.

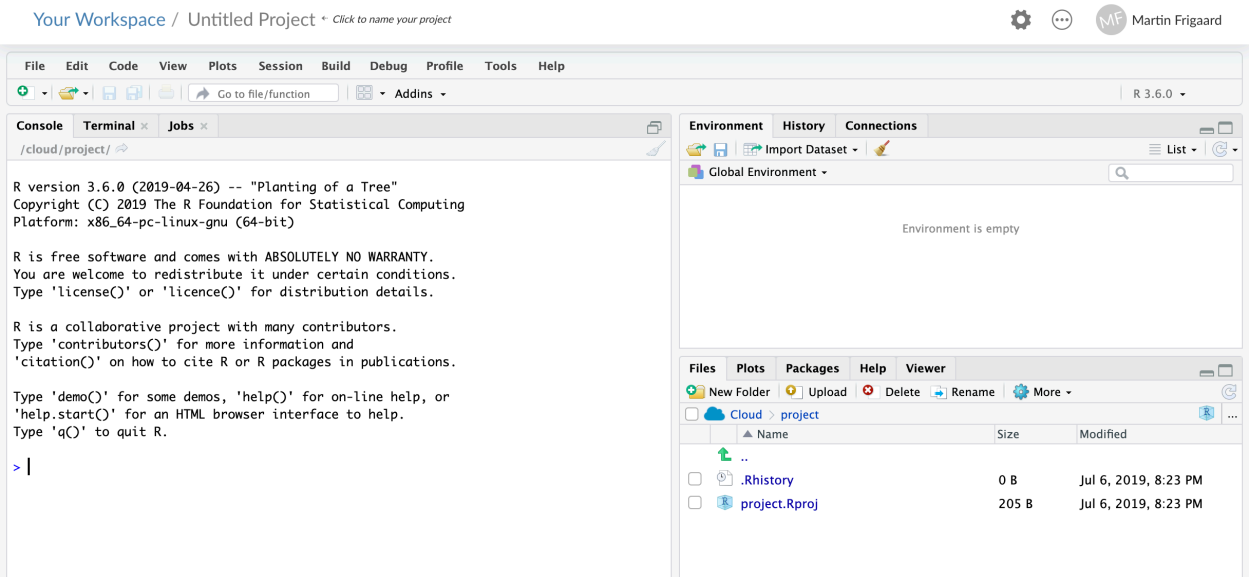
[21] https://rstudio.cloud/



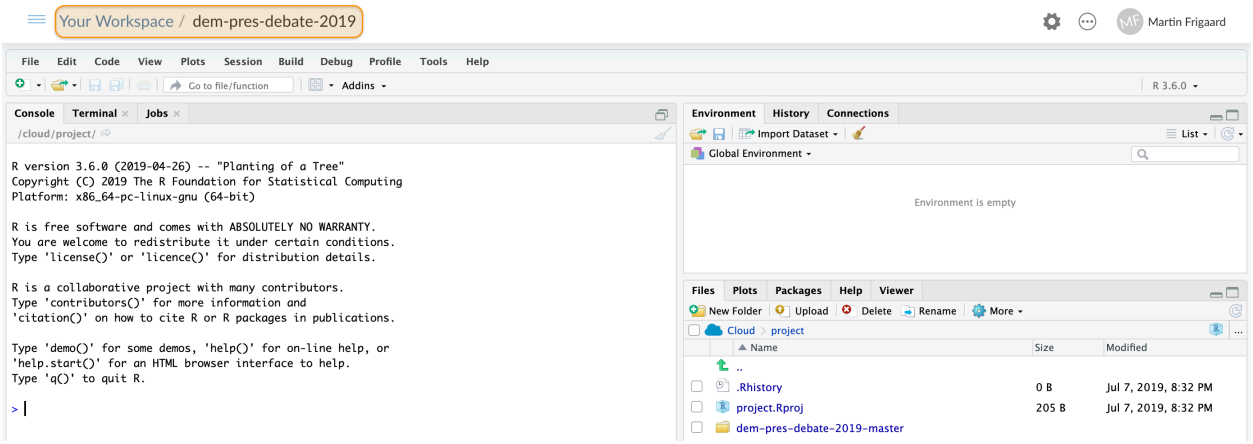
Pick a location on your computer to put your project and download the zipped Github folder.

Step 4: Upload files into RStudio.Cloud

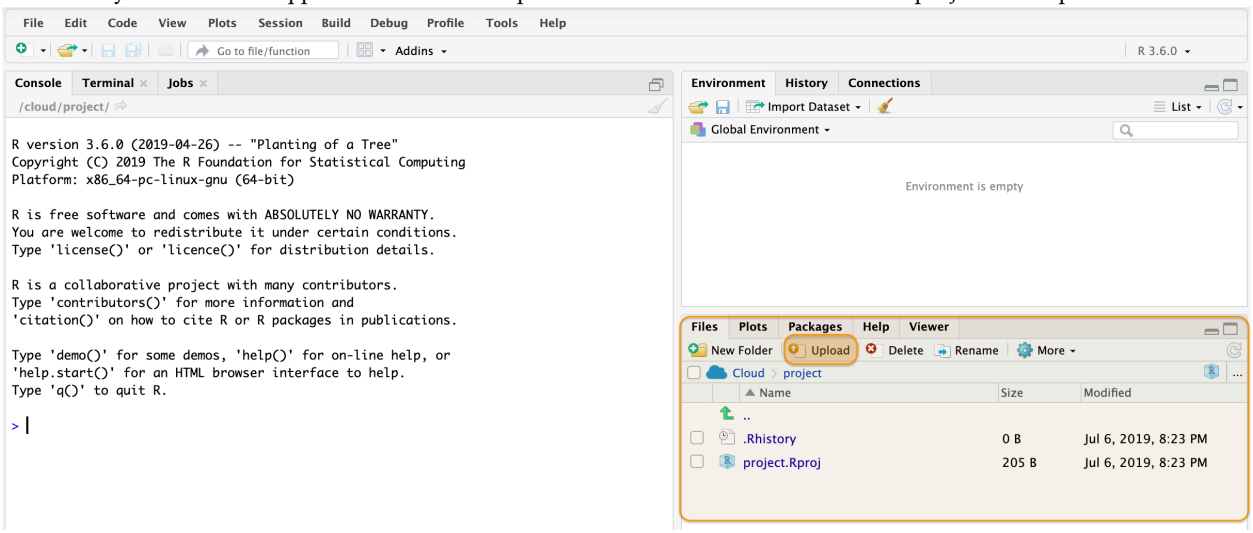
Back in the RStudio.Cloud browser, we're going to click on the **New Project** button. It should display the RStudio IDE in the browser like the image below:



We are going to change the name of this **Untitled** project to **dem-pres-debate-2019**. The results should look like the image below:



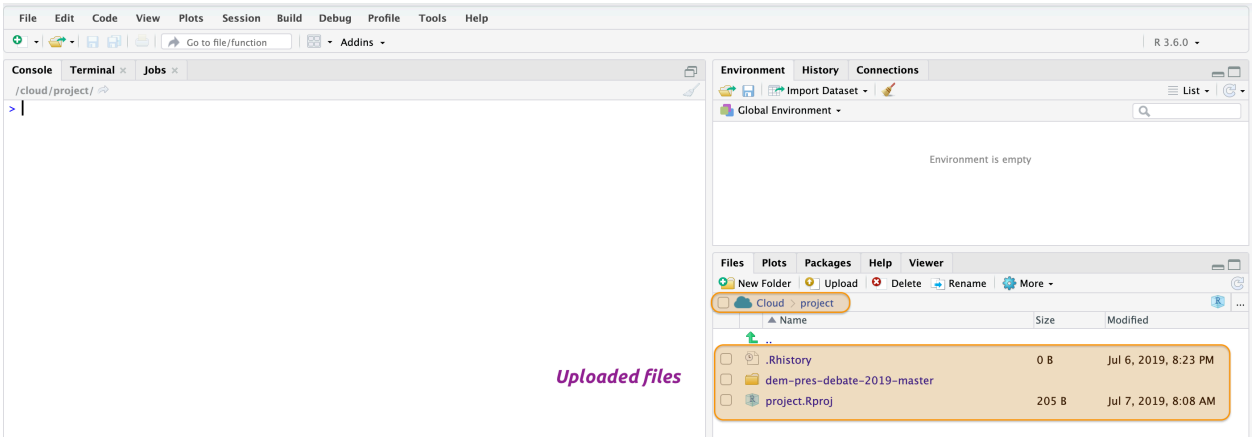
Look at the **Files** pane in the lower right corner and click on the **Upload** button, then click on **Choose files** and locate the recently downloaded zipped Github folder. Upload this file into the RStudio.Cloud project workspace.



upload window

Accessing files in RStudio.Cloud

Uploading these files might take some time, but when everything is in RStudio.Cloud, we'll see the `dem-pres-debate-2019-master` folder in the **Files** pane.



uploaded files

The unzipped the file we uploaded and created a folder called `dem-pres-debate-2019-master`. Unfortunately, it put this folder *inside* the `cloud/project` folder. We wanted to upload the *contents* of the `dem-pres-debate-2019-master` file into the `cloud/project` folder (and not the folder itself).

```
Cloud/project/ # here
              ├── dem-pres-debate-2019-master # move these contents...
              └── project.Rproj
```

We are going to use this opportunity to introduce a few command line tools. To do this, we'll be working from the **Terminal** pane in RStudio.Cloud. The next session will be a quick "crash course" in operating system terms, their differences, a few **Terminal** commands, why we still have these command line tools, and how to use them effectively.

The Command line: Unix and Windows

In 2007, Apple released its Leopard[22] operating system that was the first to adhere to the Single Unix Specification[23]. I only introduce this bit of history to help keep the terminology straight. macOS and Linux are both Unix systems, so they have a similar underlying architecture (and philosophy). Most Linux commands also work on macOS.

Windows has a command line tool called Powershell, but this is not the same as the Unix shells discussed above. The differences between these tools reflect the differences in design between the two operating systems. However, if you're a Windows 10 user, you can install a bash shell command-line tool[24].

[22] https://en.wikipedia.org/wiki/MacOS_version_history#Version_10.5:_%22Leopard%22
[23] https://en.wikipedia.org/wiki/Single_UNIX_Specification
[24] https://www.windowscentral.com/how-install-bash-shell-command-line-windows-10

Command line interfaces

The command line interface[25] (CLI) was the predecessor to a GUI, and there is a reason these tools haven't gone away. CLI is a text-based screen where users interact with their computer's programs, files, and operating system using a combination of commands and parameters. This basic design might make the CLI sound inferior to a trackpad or touchscreen, but after a few examples of what's possible from on the command-line and you'll see the power of using these tools.

What am I getting out of this?

That is a fair question–being able to use the command line gives us more 'under-the-hood' access to any computer. We can use the command line to navigate our computer's files, install new programs or libraries, and track changes to files. It might seem clunky and ancient, but people keep this technology around because of it's 1) specificity and 2) modularity (also the two features that make Unix programs so powerful). What do we mean by this?

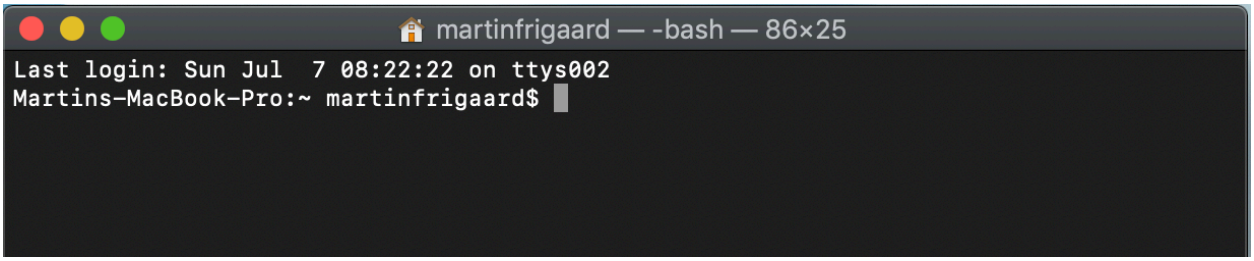
- Specificity[26] means each Unix command or tool does one thing very well (or DOTADIW[27])
- Modularity[28] is the ability to mix and match these tools together with 'pipes,' a kind of grammatical glue that allows users to expand these tools in seemingly endless combinations

Having these skills have also make us more comfortable when we've had to interact with remote machines or foreign operating systems (Linux, per se). We will work through an example to demonstrate some of these features.

The Terminal (mac0S)

Below is an image of what the terminal application looks like on macOS. On Macs, the Terminal application runs a bash shell[29], which is why you can see the `bash -- 86x25` on the top of the window. Bash is a commonly used shell, but there are other options too (see Zsh[30], tcsh[31], and sh[32]). *Fun fact: `bash` is a pun for the `sh` shell: `b`ourne-`a`gain `sh`ell.*

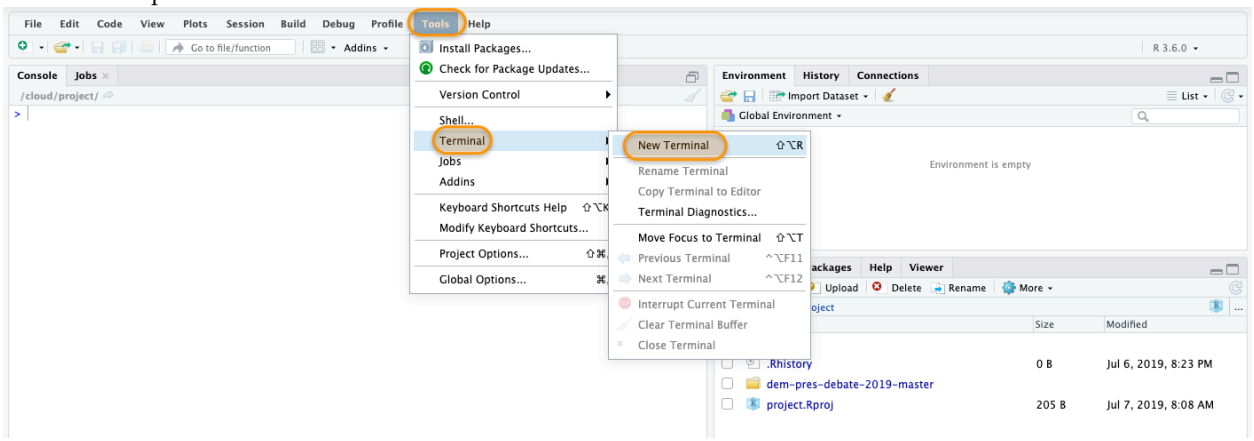
[25] https://en.wikipedia.org/wiki/Command-line_interface
[26] https://www.dictionary.com/browse/specific
[27] https://en.wikipedia.org/wiki/Unix_philosophy#Do_One_Thing_and_Do_It_Well
[28] https://en.wikipedia.org/wiki/Modularity#Table_1:_The_use_of_modularity_by_discipline[34]
[29] https://en.wikipedia.org/wiki/Bash_(Unix_shell)
[30] http://zsh.sourceforge.net/
[31] https://en.wikipedia.org/wiki/Tcsh
[32] https://en.wikipedia.org/wiki/Bourne_shell



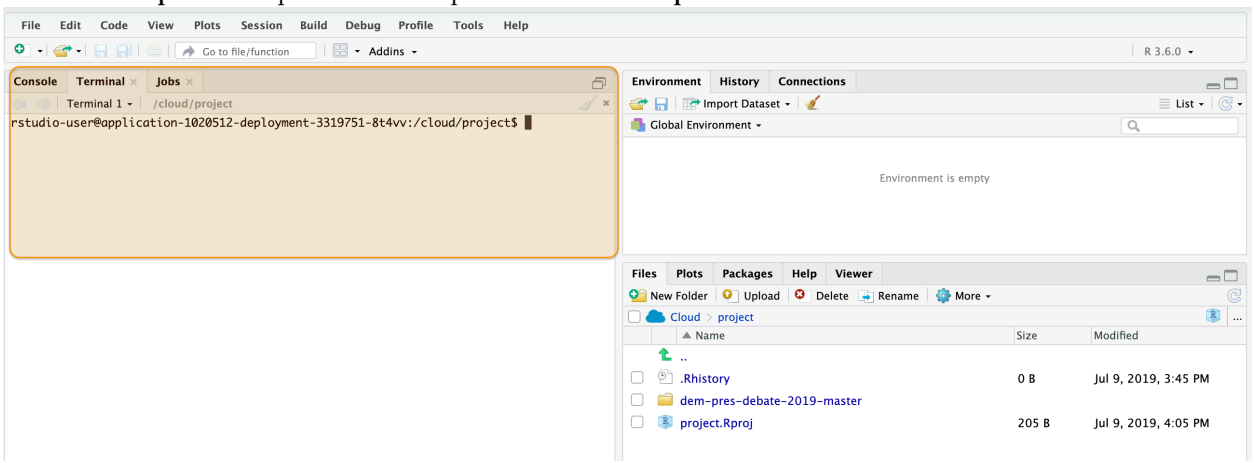
The Terminal is a command line interface emulator application for Mac users. Terminal is available as an application under the **Applications** > **Utilities** > **Terminal**.

The Terminal (RStudio)

The Terminal pane is also available in RStudio under **Tools** > **Terminal** > **New Terminal**.



The **Terminal pane** will open in the same pane as the **Console pane**.



Now we will get some practice organizing our data science project using the command line.

Good enough command line tools

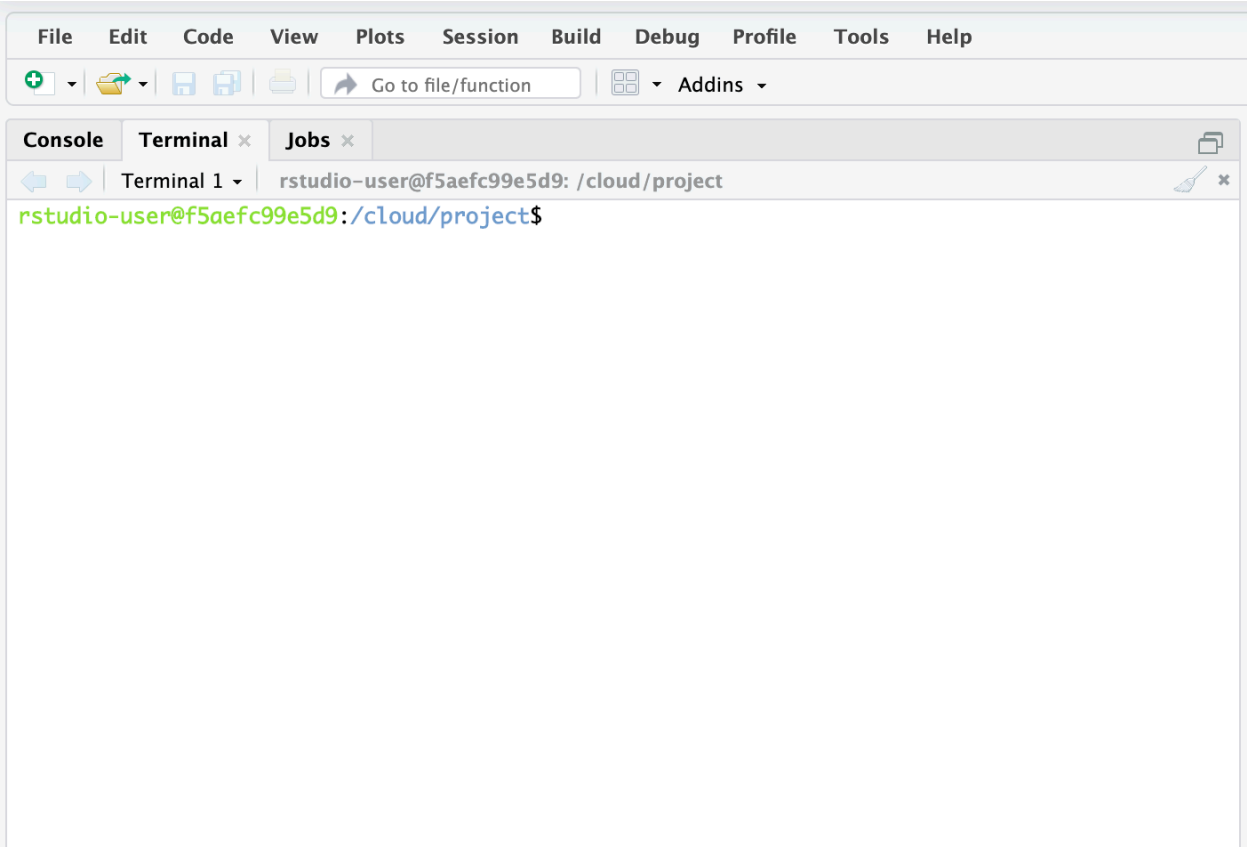
FAIR WARNING–command line interfaces can be frustrating. Computers don't behave in ways that are easy to understand (that's why GUIs exist). Switching from a GUI to a CLI seems like a step backward at first, but the initial headaches pay off because of the gains we'll have in control, flexibility, automation, and reproducibility.

Here is a quick list of commonly used Terminal commands.

- **`pwd`** - `p`rint `w`orking `d`irectory
- **`cd`** - `c`hange `d`irectories
- **`cp`** - `c`o`p`y files from one directory to another
- **`ls`** - `l`i`s`t all files
- **`ls -la`** - `l`i`s`t all files, including hidden ones
- **`mkdir`** - `m`a`k`e `dir`ectory
- **`rmdir`** - `r`e`m`ove a `dir`ectory
- **`cat`** - display a text file in Terminal screen
- **`echo`** - outputs text as arguments, prints to Terminal screen, file, or in a pipeline
- **`touch`** - create a few files
- **`grep`** - "`g`lobally search a `r`egular `e`xpression and `p`rint"
- **`>>`** and **`>`** - redirect output of program to a file (don't display on Terminal screen)
- **`sudo`** and **`sudo -s`** (BE CAREFUL!!) performing commands as **`root`** user can carry some heavy consequences.

Command line skill #1: who is using what?

After downloading the files from Github, we've uploaded the zipped folder into the `Cloud/project`. In the RStudio.Cloud **Terminal** pane, we should see something like this:



Cloud terminal prompt

This looks like gobbledygook at first, but command line interfaces have a recognizable pattern if we know what we're looking for:

- First, we can almost always expect some kind of `user@machine` identifier to tell us who we're signed is as and on what machine
- Second, there's usually some way of displaying the **`home`** directory. In this case, it's the stuff between the colon (`:`) and the dollar sign `$` (**`/cloud/project`**)

Let's check a few things to help figure out what's going on.

```
rstudio-user@f5aefc99e5d9:/cloud/project$ whoami
rstudio-user
```

We are the `rstudio-user` on this machine `f5aefc99e5d9`. The same information on a local MacBook laptop might look like this:

```
Martins-MacBook-Pro:~ martinfrigaard$ whoami
martinfrigaard
```

In this case, the machine information would be `Martins-MacBook-Pro` and the location to be the **`home`** directory `~` (the top level folder) for the user `martinfrigaard`.

Command line skill #2: where am I?

In the RStudio.Cloud **Terminal** pane, enter the print working directory (`pwd`) command:

I've omitted everything preceeding the prompt (`$`) for easier printing

```
$ pwd
/cloud/project
```

`pwd` tells us where we are, otherwise known as the current working directory. Imagine the current working directory as the spot we're standing, and file path `/cloud/project` as the way back to our **`root`** folder.

To get a sense of our surroundings, lets list the files in `/cloud/project` using `ls`

```
$ ls
dem-pres-debate-2019-master  project.Rproj
```

We can see the folder (`dem-pres-debate-2019-master`) and the RStudio project file (`project.Rproj`). On a side note, it's always a good idea to pay attention to file extensions (`.Rproj`, `.R`, `.md`, etc.), because different files interact with the **Terminal** in different ways.

Command line skill #3: moving around

Now that we know where we are, and what files and folders are in here with us, we can start to stretch our legs and move around. Let's start by changing directories `cd` to the `dem-pres-debate-2019-master` folder, then check with `pwd`.

```
$ cd dem-pres-debate-2019-master
$ pwd
/cloud/project/dem-pres-debate-2019-master
```

Now we can check the files in this new directory with `ls`

```
$ ls
01-import.Rmd     README.Rmd  data
02-wrangle.Rmd    README.md   dem-pres-debate-2019.Rproj
03-visualize.Rmd  code        figs
```

The output from `ls` shows me there are four sub-folders in the `dem-pres-debate-2019-master` folder, two `.Rmd` files, one `.md`, and one `.Rproj` file.

Now that we've moved into this folder and looked around, let's climb back out of it. We can always move up one folder by executing the `cd ..` command.

```
$ cd ..
$ pwd
cloud/project
```

Let's move back into `dem-pres-debate-2019-master` using `cd` again, but this time we will move up one folder using `cd /cloud/project`.

```
$ cd /cloud/project
$ ls
dem-pres-debate-2019-master  project.Rproj
```

We can also check the files in `dem-pres-debate-2019-master` using `ls` and the folder name.

```
$ ls dem-pres-debate-2019-master
01-import.Rmd     README.Rmd  data
02-wrangle.Rmd    README.md   dem-pres-debate-2019.Rproj
03-visualize.Rmd  code        figs
```

This tells **Terminal** to list the files in the folder at the end of the file path.

Absolute vs. relative file paths

An **absolute file path** starts at the root directory (~ or \) and follows along the path, folder by folder, until it lands in the last folder or file.

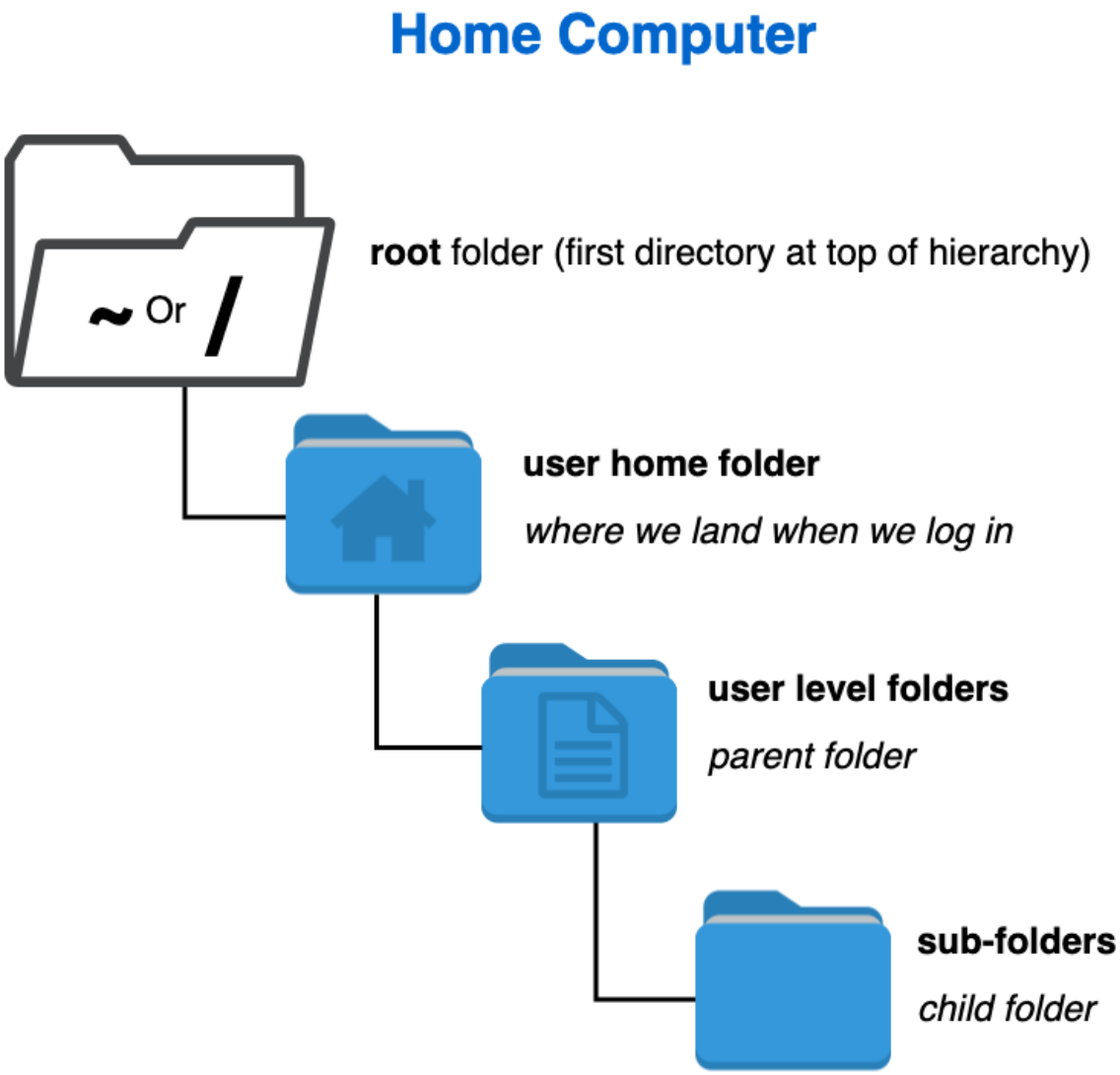
`/start/from/absolutely/where/i/tell/us`

A **relative file path** starts at a folder but leaves the rest 'relative' to wherever that folder is located.

`start/from/wherever/we/put/me`

Folder trees

Below is an example folder tree structure on a macOS.



The **`root`** folder is the "upper most" location of this machine's folders and files. In macOS, `root` is represented with a tilde (~). In Windows, the `root` folder is located with the forward slash (`/`). If we have the right priviliges, we can log in as the `root` user, and the prompt will change from `$` to `#` (be careful here!)

When we log into a computer, we start in a **`home`** folder (usually with a shorter version of that user's full name they used to set up their operating system). The `home` folder is the typical "starting point" for that `user`'s folders and files. If we are working on macOS, this is the folder with a little house on it.

Depending on the operating system, this location starts off with some standard default folders (`Desktop`, `Documents`,

`Downloads`, and `Applications`)

Special case: Windows machines

On a Windows machines, the file path to `dem-pres-debate-2019-master` might look like this:

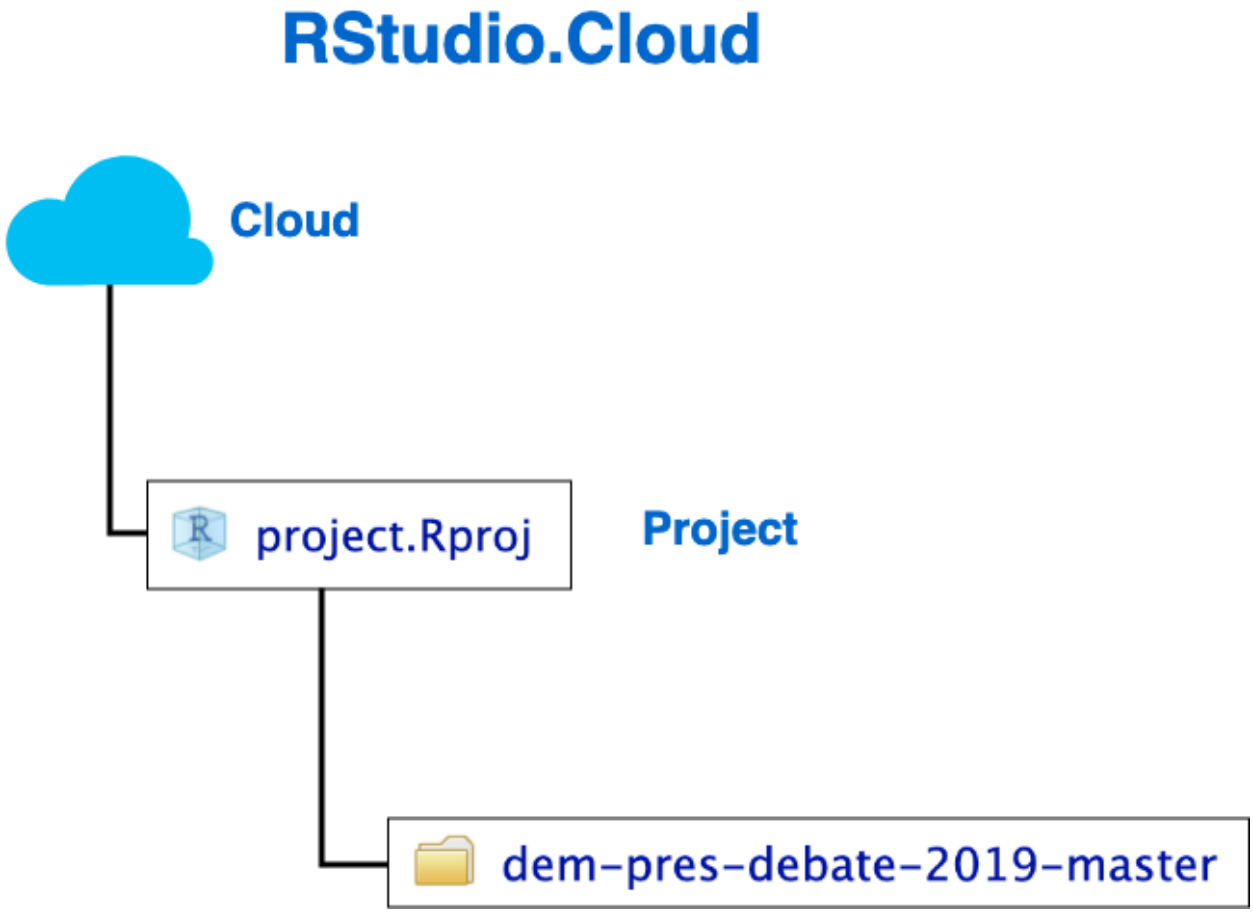
```
C:\Users\martinfrigaard\Documents\dem-pres-debate-2019-master
```

But we would need to write it like this:

```
C:\\Users\\martinfrigaard\\Documents\\dem-pres-debate-2019-master
```

This is because in R, the `\` is called an escape character, so in order to navigate through folders we will have to use two backslashes `\\`.

Below is the folder tree on RStudio.Cloud:



Now, this image might be about as clear as mud, but it'll make more sense when we start moving things around.

Command line skill #4: moving things around

We're working in RStudio.Cloud, but the GUI representation of our folder structure won't be much different if we were working on our local laptop.

Remember, we want to move the contents of `dem-pres-debate-2019-master` into `cloud/project`. The command for moving files from one place to another is `mv`, but we are going to add an two options, `-v` and `*`. There are many other options for using `mv`, read about them here[33].

The sequence of commands we'll enter in the RStudio.Cloud Terminal are below:

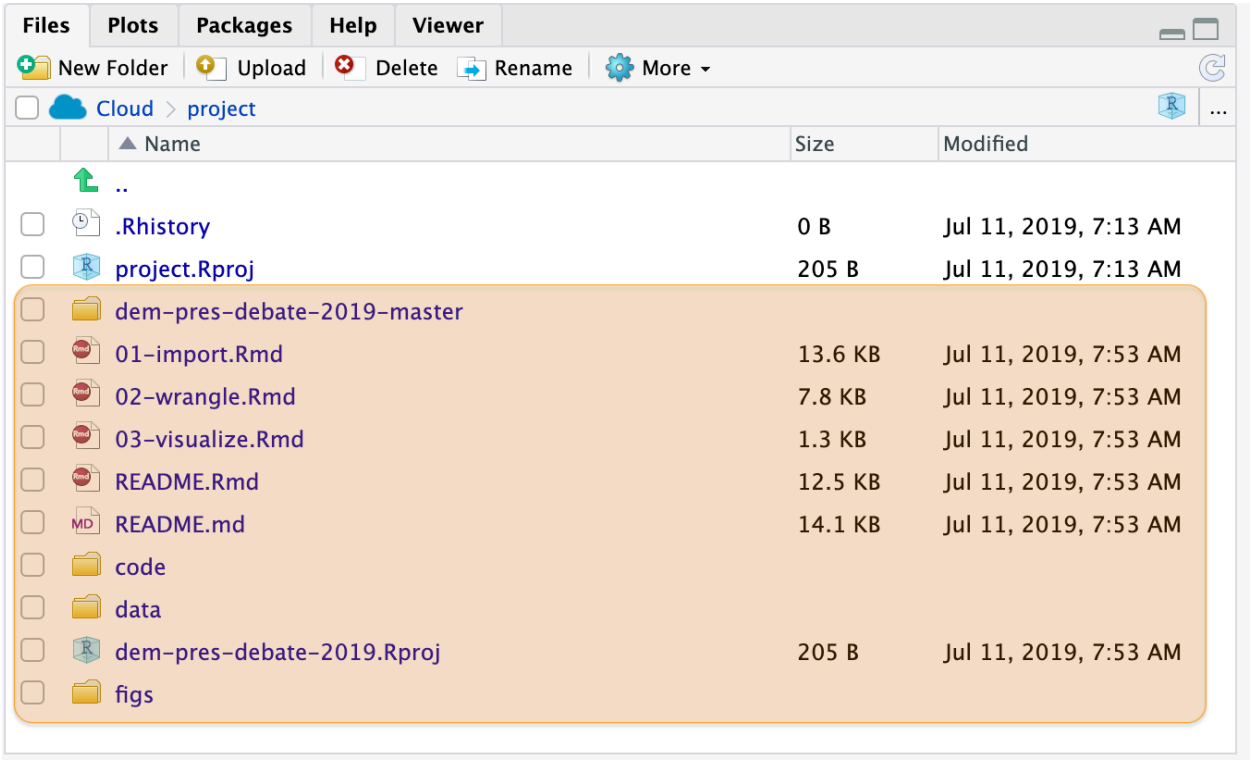
```
$ mv -v dem-pres-debate-2019-master/* /cloud/project
```

You will see the following changes in **Terminal**:

```
'dem-pres-debate-2019-master/01-import.Rmd' -> '/cloud/project/01-import.Rmd'
'dem-pres-debate-2019-master/02-wrangle.Rmd' -> '/cloud/project/02-wrangle.Rmd'
'dem-pres-debate-2019-master/03-visualize.Rmd' -> '/cloud/project/03-visualize.Rmd'
'dem-pres-debate-2019-master/README.Rmd' -> '/cloud/project/README.Rmd'
'dem-pres-debate-2019-master/README.md' -> '/cloud/project/README.md'
'dem-pres-debate-2019-master/code' -> '/cloud/project/code'
'dem-pres-debate-2019-master/data' -> '/cloud/project/data'
'dem-pres-debate-2019-master/dem-pres-debate-2019.Rproj' -> '/cloud/project/dem-pres-debate-20\
19.Rproj'
'dem-pres-debate-2019-master/figs' -> '/cloud/project/figs'
```

And the following changes in the **Files** pane:

[33] https://www.gnu.org/software/coreutils/manual/html_node/mv-invocation.html#mv-invocation



mv-ed files

This tells us all of the files have been moved. But we will want to get rid of the old folder, `dem-pres-debate-2019-master`.

Command line skill #5: Deleting things

To delete a folder, we can either use `rmdir` or `rm -Ri`.

```
$ rm dem-pres-debate-2019-master -Ri
rm: descend into directory 'dem-pres-debate-2019-master'?
```

This command is helpful because the `i` option tells **Terminal** to check wiht us before doing anthing. Go ahead and enter `n` and try using `rmdir` to delete the `dem-pres-debate-2019-master` folder.

```
$ rmdir dem-pres-debate-2019-master
rmdir: failed to remove 'dem-pres-debate-2019-master': Directory not empty
```

Terminal does it's best to save us from ourrselves, but thats not always possible. As Doug Gwyn said,

> "*Unix was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things.*"

Well what does `rmdir` actually do then? We can figure this out with `rmdir --help`

To delete a folder with files, in it, we have to add the “ option

```
$  rmdir --help
```

This command will print some useful information about the `rmdir` command:

```
$  rmdir --help
Usage: rmdir [OPTION]... DIRECTORY...
Remove the DIRECTORY(ies), if they are empty.
# else omitted...
```

Now we know this is not the right tool for the job (the folder isn’t empty), so we will use `rm -Ri dem-pres-debate-2019-master`. Each folder and file will prompt a question that needs a response before **Terminal** can delete anything.

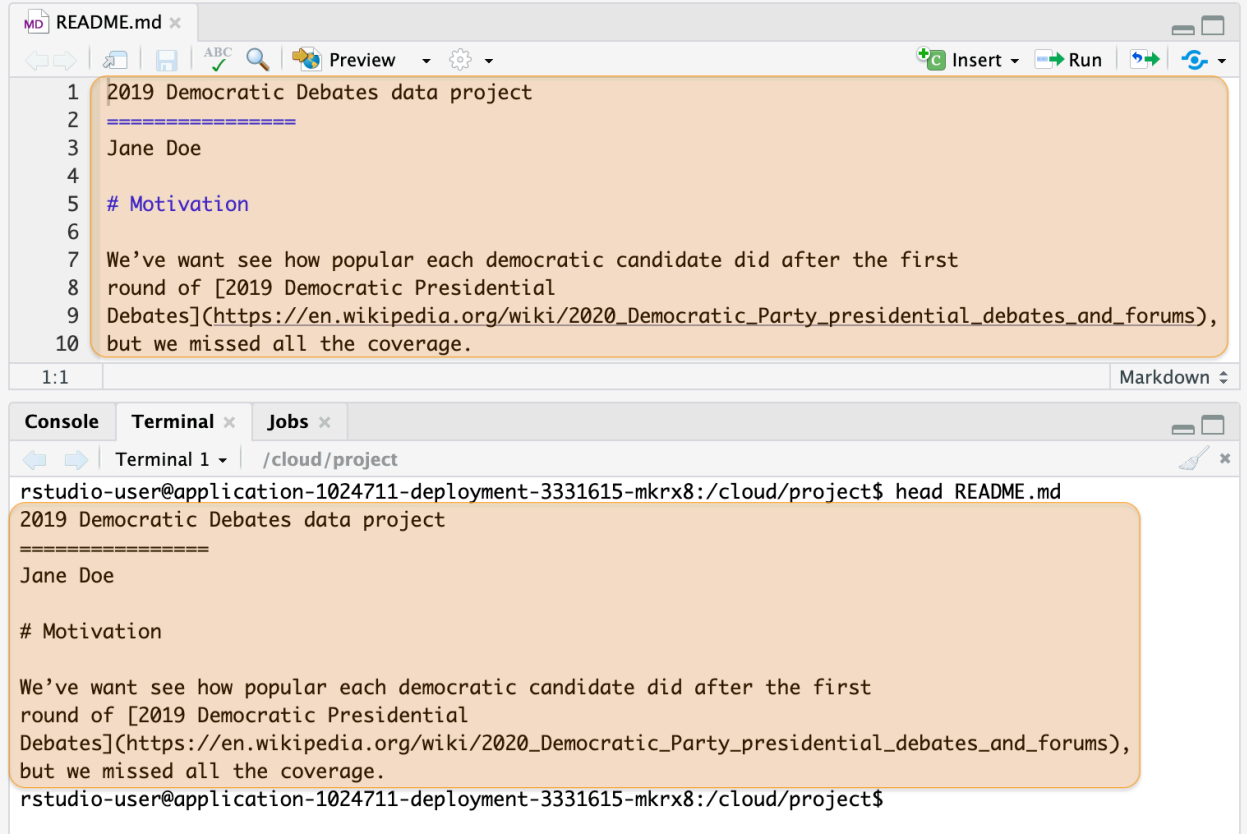
The **Terminal** pane should have the following contents when we’re finished:

```
$ rm -Ri dem-pres-debate-2019-master
rm: descend into directory 'dem-pres-debate-2019-master'? y
rm: remove regular file 'dem-pres-debate-2019-master/.DS_Store'? y
rm: remove regular file 'dem-pres-debate-2019-master/.gitignore'? y
rm: remove directory 'dem-pres-debate-2019-master'? y
```

Command line skill #6: Printing things

Terminal works very well with plain text format. For example, I can use `head` and the name of a file I want to see.

```
$ head README.md
```



As we can see, this is the first few lines of the `README.md`. Markdown is a plain text format, so it will print clearly to the **Terminal** window. In addition to `head`, we can also use the `tail` command to view the bottom of the `README.md` file.

What if we want to see all the contents in `README.md`? Well, before doing this we want to see how big the file is, and we can do that using `wc` (which stands for "word count").

```
$ wc README.md
# 462  1739 14415 README.md
```

The three numbers above are the number of lines (`462`), the number of words (`1739`), and the number of characters (`14415`).

This is telling us that `README.md` might be hard to read on the Terminal window. Fortunately, that's where the `less` command comes in.

```
$ less README.md
```

`less` will display the contents of `README.md`, but in a way that allows us to scroll through the file using the arrow keys. After we're done viewing the file, we can exit `less` using `q`.

Another option to print is `cat`, but this will print the entire contents to the **Terminal** window, so use `wc` first to see if that's the best choice.

Command line skill #7: Create things

Sometimes we might need to create a new file and add some text to it. This skill is handy if we don't have to open any new applications.

The `touch` command will create a new file (`CHANGELOG.txt`), and `echo` will put the "some thoughts" on this file (which we can verify with `cat`).

```
$ touch CHANGELOG.txt
$ echo "some thoughts" > CHANGELOG.txt
$ cat CHANGELOG.txt
some thoughts
```

The **>** symbol tells **Terminal** to send `echo "some thoughts"` to `CHANGELOG.txt`. When we use `car`, we see these commands put "some thoughts" into the top lines of the new file, `CHANGELOG.txt`.

The `CHANGELOG.txt` file is for writing notes about changes to our project, but we should add a date to make sure they're listed chronologically. Unix has a `date` variable we can access using `$(date)` (which essentially 'attaches' the output from the command date with "some thoughts"), so we will repeat the process above, but include today's date with `$(date)`.

```
$ echo $(date) "some thoughts" > CHANGELOG.txt
$ cat CHANGELOG.txt
```

In Unix systems, we can always access today's date with the `date` or `cal`.

Command line skill #8: Combine things

The commands above are great for creating new files and adding new text, but what if `CHANGELOG.txt` already exists and we wanted to add more thoughts to it? We can do this by changing the **>** symbol to **>>**.

```
$ echo $(date) "more thoughts" >> CHANGELOG.txt
$ cat CHANGELOG.txt
# Thu Jul 11 13:45:57 UTC 2019 some thoughts
# Thu Jul 11 13:49:42 UTC 2019 more thoughts
```

>> tells **Terminal** to append the output from `echo` to `CHANGELOG.txt` on a new line.

Another powerful tool in the Unix toolkit is the pipe (`|`). This can be used to 'direct' outputs from one command to another. For example, if I wanted to see how many R script files are in `code` folder, I could use the following:

```
$ ls code | grep ".R" | less
```

We will leave the `grep` command for you to investigate with `--help` to figure out what's happening here.

Other command line stuff: homebrew

The bash shell on macOS comes with a whole host of packages we can install with homebrew[34], the "The missing package manager for macOS (or Linux)".

(You won't be able to do this on RStudio Terminal, but there are other options we will list below)

After installing homebrew, we recommend installing the `tree`[35] package.

```
$ # install tree with homebrew
$ brew install tree
$ # get a folder tree for this project
$ tree
```

The `tree` command gives us output like the folder tree below.

```
├── 01-import.Rmd
├── 02-wrangle.Rmd
├── 03-visualize.Rmd
├── README.Rmd
├── README.md
├── code
├── data
│   ├── processed
│   └── raw
└── dem-pres-debate-2019.Rproj
```

Folder trees come in handy for documenting the project files (and any changes to them).

Command line recap

We've covered eight command line tools, and we hope you can see how these can be combined to create very efficient workflows and procedures. By tethering commands together, we can move inputs and outputs around with a lot of flexibility.

[34] https://brew.sh/

[35] https://brewinstall.org/Install-tree-on-Mac-with-Brew/

Organizing your project files

As we saw above, the `tree` output gave us a printout of the project folder in a hierarchy (i.e. a tree with branches).

The thing to notice is the separation of files into folders titled, `data`, `docs`, and `src` or `code`. These folder names were not chosen at random–there is a way to organize a data science project. We recommend starting with the structure outlined by Greg Wilson et al. in the paper, "Good Enough Practices for Scientific Computing"[36]. If you already have a organization scheme, we still recommend reading at least this section[37] of the paper–it's full of great information and links to other resources.

Getting more help

This section has been a concise introduction to command line tools, but hopefully, we've demystified some of the terminologies for you. The reason these technologies still exist is that they are powerful. Probably, you're starting to see the differences between these tools and the standard GUI software installed on most machines. Vince Buffalo[38], sums up the difference very well,

> "*the Unix shell does not care if commands are mistyped or if they will destroy files; the Unix shell is not designed to prevent you from doing unsafe things.*"

The command line can seem intimidating because of its power and ability to destroy the world, but there are extensive resources available for safely using it and adding it to your wheelhouse.

- The Unix Workbench[39]
- Data Science at the Command Line[40]
- Software Carpentry Unix Workshop[41]

More on file organization, collaborating, and version control

Fortunately, many articles have come out in the last few years with excellent, practical advice. I recommend reading these before getting started (you'd be surprised at the cacophony of files a single project can produce). We've listed a few 'must reads' below:

[36] https://swcarpentry.github.io/good-enough-practices-in-scientific-computing/#project-organization
[37] (https://swcarpentry.github.io/good-enough-practices-in-scientific-computing/#project-organization)
[38] http://vincebuffalo.org/blog/
[39] https://seankross.com/the-unix-workbench/
[40] https://www.datascienceatthecommandline.com/
[41] https://swcarpentry.github.io/shell-novice/

- working with data in spread sheets[42],
- sharing data with collaborators[43],
- how to name your files[44], and
- the importance of using version control[45].

Terminals vs. Shells: Sometimes you'll hear the term "shell" thrown around when researching command line tools. Strictly speaking, the Terminal application is not a shell, but rather it *gives the user access to the shell*. Other terminal emulator options exist, depending on your operating system and age of your machine. Terminal.app is the default application installed on macOS, but you can download other options (see iTerm2[46]). For example, the GNOME[47] is a desktop environment based on Linux which also has a Terminal emulator, but this gives users access to the Unix shell.

[42] https://www.tandfonline.com/doi/full/10.1080/00031305.2017.1375989
[43] https://www.tandfonline.com/doi/full/10.1080/00031305.2017.1375987
[44] https://speakerdeck.com/jennybc/how-to-name-files
[45] https://www.nature.com/news/democratic-databases-science-on-github-1.20719
[46] https://www.iterm2.com/
[47] https://en.wikipedia.org/wiki/GNOME

5. Part 4: keeping track of the changes to your work

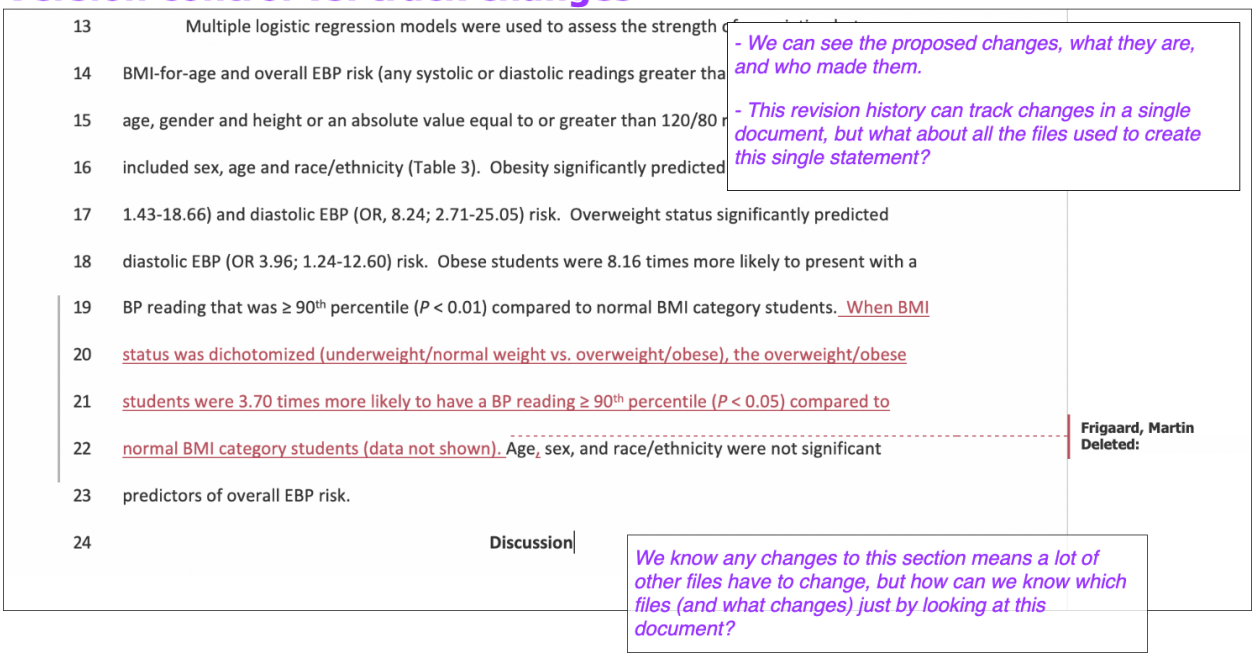
In the previous sections, we've covered R and RStudio and the command line. If you're unfamiliar with these topics, please start there. This section will include tracking changes, plain text files, version control with Git in RStudio.

Tracing your steps

'Showing your' work also means showing how your work has changed over time. In the example project, the `doc` folder contains a file titled, `2012-10-62-ican-manuscript-revision-v02.docx`. The .docx is an earlier version of the manuscript, and there are some suggested changes to the results section below.

The good thing about using tracked changes in .docx files is that we can see 1) what the differences are, 2) who suggested them, 3) the time/date of the proposed change, and 4) any comments about the change.

Version control vs. track changes



Unfortunately, this only applies to a single document. When you're writing by committee (which is quite often in science), you know asking someone to change a single sentence can result in changes to dozens of files. Fortunately, this change is suggesting a deletion, so this is unlikely to result in generating additional analyses, tables, write-ups, etc.

Git

Git is a version control system[1] (VCS), which is somewhat like the **Tracked Changes** in Microsoft Word or the **Version History** in Google Docs, but extended to every file in a project. Git will help you keep track of your documents, datasets, code, images, and anything else you tell it to keep an eye on.

Plain text + Git

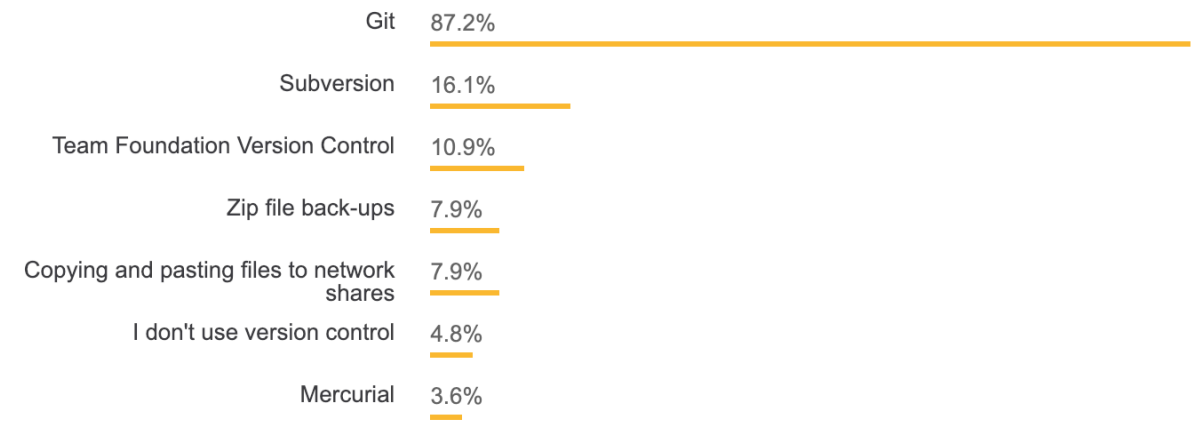
Git prefers plain text files because, until recently, software engineers and app developers were using programs like Git to track their source code (which they write in plain text)–another reason we recommend keeping your documentation and notes in a plain text file.

Why use Git?

You will eventually ask yourself, *why am I subjecting myself to this–is there another way?*

We've included these sections to remind you that you're making a sound choice.

Git has become the most common version control system used by programmers[2].



74,298 responses; select all that apply

Git is the dominant choice for version control for developers today, with almost 90% of developers checking in their code via Git.

[1] https://en.wikipedia.org/wiki/Version_control
[2] https://insights.stackoverflow.com/survey/2018#work-version-control

source: StackOverflow Developer Survey Results[3]

Git is a useful way to think about making changes

Git is also a helpful way of thinking about the changes to your project. The terminology of Git is strange at first, but if you use Git long enough, you'll be thinking about your code in terms of 'commits,' 'pushes,' 'forks,' and 'repos.'

As someone who analyzes data regularly, these concepts are also countable, which means you can quantify change and work in more interesting ways.

Installing Git

1. Download and install Git.[4]
2. Create a Github[5] account.

Configuring Git with `git config`

Git needs a little configuration before we can start using it and linking it to Github. There are three levels of configuration within Git, `system`, `user`, and `project.`

1) For **system** level configuration use:

```
git config --system
```

2) For **user** level configuration, use:

```
git config --global
```

3) For **Project** level configuration use:

```
git config
```

I'll set my Git `user.name` and `user.email` with `git config --global` so these I've configured these for all projects on my computer.

```
$ git config --global user.name "Martin Frigaard"
$ git config --global user.email "mjfrigaard@gmail.com"
```

I can check what I've configured with `git config --list`.

[3] https://insights.stackoverflow.com/survey/2018#overview
[4] http://git-scm.org/
[5] https://github.com/

```
$ git config --list
```

At the bottom of the output, I can see the changes.

```
user.name=Martin Frigaard
user.email=mjfrigaard@gmail.com
```

These are also stored in a `.gitconfig` file I can view using:

```
$ cat .gitconfig
[user]
    name = Martin Frigaard
    email = mjfrigaard@gmail.com
```

Synchronize RStudio and Git/Github

Jenny Bryan[6] has created the online resource Happy Git and GitHub for the useR[7] has all in the information you will need for connecting RStudio and Git/Github. I echo a lot of this information below (with copious screenshots).

Go to *Tools* > *Global Options* > ...

- 1. Click on *Git/SVN*
- 1. Then *Create RSA Key...*
- 3, 4, and 5. In the dialog box, enter a passphrase (and store it in a safe place), then click *Create*.

The result should look something like this:

[6] https://jennybryan.org/
[7] http://happygitwithr.com/

```
whoeveryouare ~ $ ssh-keygen -t rsa -b 2891 -C "USEFUL-COMMENT"
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/username/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/username/.ssh/id_rsa.
Your public key has been saved in /Users/username/.ssh/id_rsa.pub.
The key fingerprint is:
SHA483:g!bB3r!sHg!bB3r!sHg!bB3r!sH USEFUL-COMMENT
The key's randomart image is:
+---[RSA 2891]----+
|      o+   . .   |
|     .=.o . +    |
|     ..= + +     |
|      .+* E      |
|     .= So =     |
|    .  +. = +    |
|   o.. = ..* .   |
|  o ++=.o =o.    |
| ..o.++o.=+.     |
+----[SHA483]-----+
```

Now I need to go back to Terminal and store this SSH from RStudio.

Adding a key SSH in Terminal

In the Terminal, I enter the following commands.

```
$ eval "$(ssh-agent -s)"
```

The response tells me I'm an Agent.

```
Agent pid 007
```

Now I want to add the *SSH RSA* to the keychain. There are three elements in this command: the `ssh-add`, the `-K`, and `~/.ssh/id_rsa`.

- The `ssh-add` is the command to add the *SSH RSA*
- The `-K` stores the passphrase I generated, and
- `~/.ssh/id_rsa` is the location of the *SSH RSA*.

```
$ ssh-add -K ~/.ssh/id_rsa
```

Enter the passphrase and then have it tell me the identity has been added.

```
Enter passphrase for /Users/username/.ssh/id_rsa:
Identity added: /Users/username/.ssh/id_rsa (username@Martins-MacBook-Pro.local)
```

Create the ~/.ssh/config file

On macOS-Sierra 10.12.2 or higher requires a `config` file. I can do this using the Terminal commands above.

First, I move into the ~/.ssh/ directory.

```
$ cd ~/.ssh/
```

Then I create this `config` file with `touch`

```
$ touch config
# verify
$ ls
config        id_rsa        id_rsa.pub
```

I use `echo` to add the following text to the `config` file.

```
Host *
 AddKeysToAgent yes
 UseKeychain yes
```

Recall the >> will send the text to the `config` file.

```
# add text
$ echo "Host *
>  AddKeysToAgent yes
>  UseKeychain yes" >> config
```

Finally, I can check `config` with `cat`

```
# verify
$ cat config
Host *
 AddKeysToAgent yes
 UseKeychain yes
```

Great! Now I am all set up to use Git with RStudio. In the next section, we will move the contents of a local folder to Github.

Writing = thinking with words

At the very minimum, a researcher should be able to communicate 1) the problem they found, 2) their proposed solution, 3) what they did, and 4) what they observed. Each of these steps has too much detail for pointing and clicking.

Every area of research has it's own jargon and acronyms, and it can be challenging to explain complicated concepts in plain language. But we've found doing this is worth the effort because the process allows you to sort through your thoughts on a topic, develop new analogies, and make new connections.

I highly recommend the article The Science of Science Writing[8] for anyone looking to improve their write technical writing abilities. I received some excellent advice along the way about prose, "*favor brevity above all else, but never sacrifice precision or clarity just to rush to the point.*"

[8] https://www.americanscientist.org/blog/the-long-view/the-science-of-scientific-writing

6. Part 5: putting it all together (and getting it online)

In this last chapter, we are going to make some changes to the files in our project, create new tables and figures, and push everything online in a way that people can find and discover our work.

`s1-ln134428491673680677-1939656818Hwf-794334153IdV137612480813442849PDF_HI0001.pdf`

Project file changes

Changes to code

Changes to docs

Changes to results

Documenting the changes with Git

Git terminology

Below are some commonly used terms/commands associated with Git and Github.

commits - commits are the staple in Git/Github the workflow. Commits are what Git uses to track the changes you've made to files or folders, so they can be considered nouns ("I'm creating a commit with these changes") or verbs (I am going to commit these changes to my project").

To quote David Demaree,

- "Semantically, each commit represents a complete snapshot of the state of your project at a given moment in time; its unique identifier serves to distinguish that state from the way the files in your project looked at any other moment in time."*

repository - this refers to the files and folders in your project and all the changes you make while working on them. On your local computer, a repository can exist in a folder you initialize a repository in (see below). On Github, a repository has the following structure: `https://github.com/<username>/<repository_name>`.

init - the command `git init` is used to initialize a new git repository (it tells Git to start tracking changes in this directory).

status - whenever you wonder what you've done, what is happening, or if you're just generally confused, you can check the status of a git repository with `git status` (use this liberally).

clone - this command copies all files and changes into a new working directory from a remote, initializes (`init`) a new Git repository, and it adds a remote called `origin`.

diff - this is how Git shows differences between files. Read more about how changes are formatted/displayed here.[1]

[1] https://www.git-tower.com/learn/git/ebook/en/command-line/advanced-topics/diffs

7. Conclusion

This book is **done** and ready for *the world to see*, hooray!