Intro: Why we wrote this

It seems like everywhere we look now, people are using data in beautiful and surprising ways to present their positions or shed light on new topics. We remember the first time we saw the impact data can have on storytelling. Hans Rosling, the Swedish physician/statistician, gave a talk displaying the gapminder dataset. Rosling perfectly his paired enthusiasm with a brilliant display on the screen. As he spoke, over a dozen colorful circles slid across the protection. His Ted profile description is perfect, "In Hans Rosling's hands, data sings."

Today, most primary sources of media use data as part of their evidence base. Check out the interactive data visualizations on the UpShot at the New York Times, the visual journalism data projects at the BBC, or the daily graphs in the Economist.

The massive amounts of data available have spawned new forms of media. Nate Silver's blog covering elections and politics has grown into multiple projects on fivethirtyeight. The Pudding is an example of an online data journalism site that covers non-conventional sources of data. Vox recently won an award for producing a graph that communicates a topic that pundits could've debated endlessly.

Now that we've shown you all this cool stuff, we want to tell you why we wrote this book,

"You've found something cool on the Internet, but you have no idea what it took to make it."

There are a ton of really great resources on the Internet right now for learning data science (see here, here, and here). Many of these courses are fantastic–they can teach you programming languages, website design, database management, statistics, and machine learning. But we sometimes found the sheer volume of these courses can be overwhelming for audiences who are wanting to understand how these technologies fit together.

We decided to take a step back and write a book that describes a data science workflow, or *shows how these tools work together*. We'll show you how R, RStudio, Git, & Github can be used to create elegant yet durable data analysis projects.

We chose to center this book around a particular use case, so there will be code files and tools we'll use that are specific to this project. But we've chosen not to spend too much time on the content of these files (we've documented them you want to look into the details). Instead, We're going to focus more on the "high level" ideas because these are topics you can take with you to your next project. We also encourage you to consult the articles and resource we've recommended throughout each chapter for more materials on each topic.

Our goal for anyone reading this book

We want to show you how to 1) take something neat you found on the Internet, 2) figure out what went into making it, and 3) see if you can reproduce the result.

We plan to include enough information to get you up and running and at the same time, not overwhelm you. If you've already Googled "Getting started in data science," you know there are a *ton* of resources. Figuring out where to start can feel like trying to get a drink of water from a fire hose.

Along the way, we will also cover some practical principles of programming, command-line tools, project file organization, and a few computer science topics.

Who this book is for

We've tried to keep the materials accessible to a broad audience, but we understand there are few useful data analysis texts written for everyone. Data tends to be very specific to the field they come from, and it's hard to find data that gets everyone excited. To try and help address this issue, we use data from multiple sources (Google trends, Twitter, Wikipedia, and Googlesheets).

We focus on the workflow and tools.

The next chapters outline a 'one-stop-shop' toolset that you can learn quickly and readily re-use (because we know your time is limited).

Technical assumptions

The reader we had in mind while writing this book was someone who,

1. Uses a computer every day at work
2. Understands how to navigate a web browser (Chrome, Safari, Firefox, etc.)
3. Has worked with a word processor (like Microsoft Word, Google Docs, or Apple Papers)

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

What this book covers

We will be covering R, RStudio, Git, and Github. We use these tools daily now, but we began our careers in other statistical programs (SPSS, Stata, SAS). We abandoned those tools (we know your pain) because of the sheer number of tasks we can accomplish, and that's what makes us recommend this toolkit to you. We've also reached out to our colleagues and included their lessons and insights.

What this book 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.

How this book is structured

We structured this book somewhat like an Army Field Manual, which means each topic was chosen using the following criteria:

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

    *This book contains **brief descriptions** of the tools we recommend, with **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'll expand on a few tools we felt are harder to grasp (Git and version control). We will also go over topics typical 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. This book is trying to narrow the gap between 1) seeing a data product (neurons) and 2) translating what you see into commands on a computer that can be used to recreate that product.

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

  We assume everyone reading this text has very little exposure to the tools we'll be covering (R, RStudio, Git, or Github). We do expect you are comfortable using a computer.

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

Now, look at the same thing in a figure.

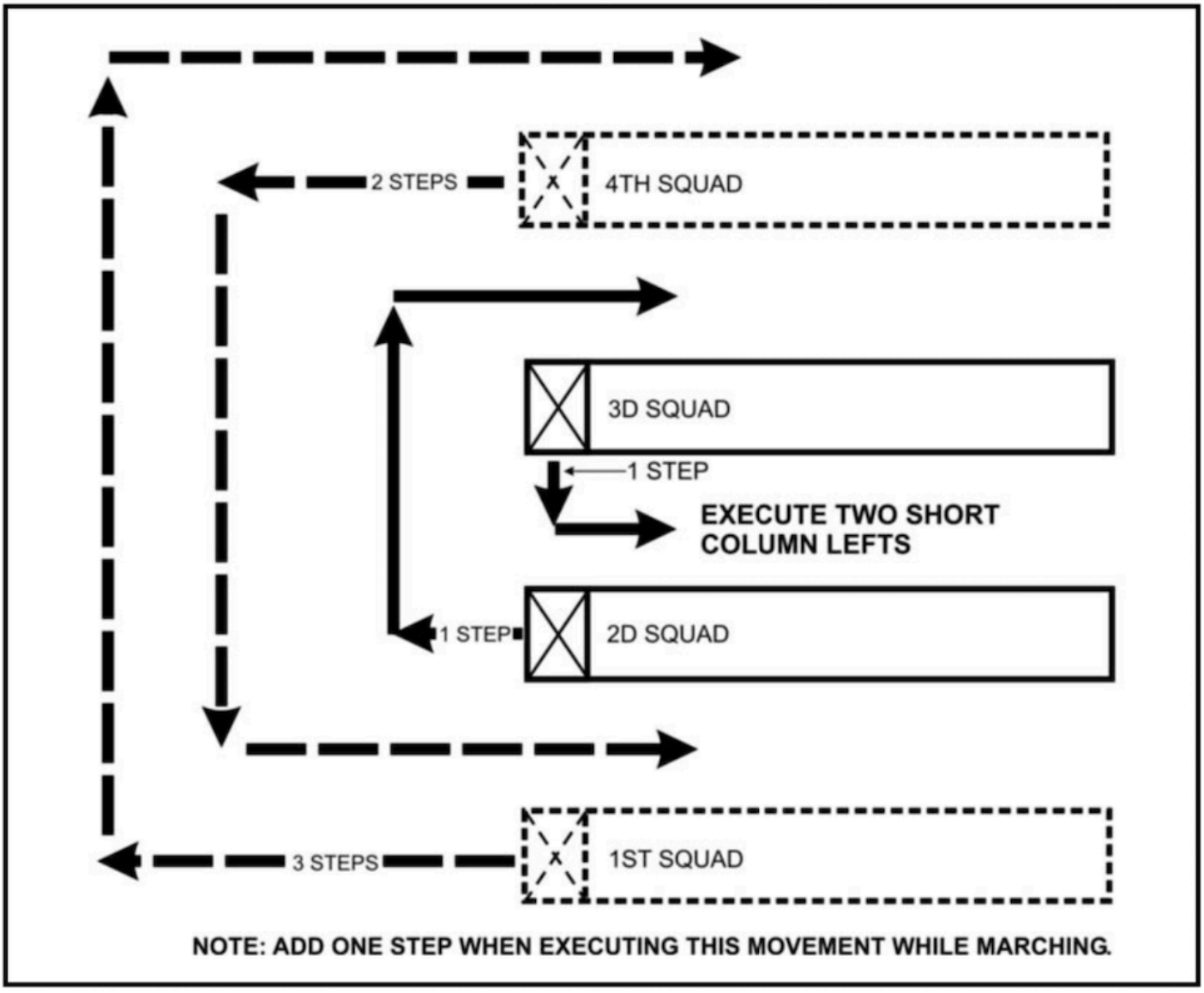


Figure 7-2. Counter-Column March at the Halt.

As you can see, executing a counter column is complicated. But the Army has taught hundreds of thousands of soldiers on this movement for decades. How? They give soldiers a field manual (FM 22–5) to read and dedicated time to practice.

The strength of the FMs is how they present information: they gave the material in everyday language (usually between a 6th–8th-grade reading level) with an emphasis on diagrams, pictures, and simple drawings.

We've found so much data science and statistical information on the internet has a ton of acronyms, jargon, and equations. We've actively avoided using technical verbiage, and focused on using figures and graphics.

> *"…there are lots of other books that explain what things are called. This book explains what they do."*

The quote above comes from Randall Munroe, author of the xkcd comic. In his book "Thing Explainer", Munroe uses pictures and plain language to describe multiple complex systems (rocket ships, the periodic table, laptops, etc.).

The subtitle of "*Thing Explainer*" is *Complicated stuff in simple words*, which is what we're trying to replicate here. Wherever possible, we've dropped unnecessary technical jargon and spelled out any acronyms.

What you'll walk away with

You will have a working project (cool visualizations, lots of code, data) a ton of resources, and a book for reproducing this process again.

Language and style guide

We use the plural 'we' throughout the book based on the excellent advice 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 book are the result of many conversations, emails, comment threads, and communications that could not have happened in isolation. We want to thank everyone who's contributed to these ideas over the years.

The text uses the following style guide:

`this is code.`

```
# this a code chunk
```

> *some quoted text*

click on hyperlinks

plain text for our thoughts

Learn more

- Practical Data Science for Stats is a resource you should bookmark in your browser. The articles in this collection will come up again in future sections, but we found we use these resources so much it's nice to have them somewhere handy.
- The R for data science community and R for Data Science book are excellent resources to help you started.
- **Collaboration and reproducibility** - there's a direct connection between collaboration and reproducibility. The better your collaborators can reproduce your work, the better they'll understand your results.

- This text is also an *opinionated technical manual*, and covers topics left out of typical statistics texts, "*Statisticians have long shied away from teaching process, with the complaint that it might limit the creativity necessary to tackle different analytical problems. However, by not teaching opinionated analysis development, we subject fledgling data to each individually spin their wheels in coming up with process for avoiding common and generalized problems*."
- We recommend RStudio and Github for anyone looking to get started with data science, visualization, reproducible reporting, dashboards development, or website/blog creation. By suggesting these particular tools, we're not saying there aren't other ways or workflows capable of accomplishing the same activities. These are the tools we've found success with, so they're what we recommend.

Part 1: ‘Good enough’ data skills

In 2016, The Carpentries–a non-profit organization that teaches coding and data science skills to researchers–published an article titled, “Good Enough Practices for Scientific Computing”.

The article above has tons of great information, but it was aimed at “*researchers who are working alone or with a handful of collaborators on projects lasting a few days to several months*.” We thought the article’s information was too useful *not* to share with as many people as possible. This book essentially attempts to extend the advice in paper to analysts, journalists, grad students, and non-technical audiences.

Why ‘good enough’?

A constant theme that runs through this book is being able to do “good enough” work. We’re going to introduce you to a lot of technology in the next few pages, and we want you to set reasonable expectations for using these tools.

There are many courses, tutorials, and resources that promise you ‘expert training’ in all of the tools we’re covering. You don’t have time for that–you need enough knowledge to see cool stuff and recreate it. In the beginning, you need to focus on doing good enough work, and then sharing your work. The feedback and input you get from developing things in the open are what builds your skillset. Expertise is something you can attain with experience (if that’s your goal), but in many cases, being good enough will get lots of work done.

How to share your work

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

There is a sea of information on the Internet, and that means everyone is competing for everyone else’s attention. You want to share whatever you’re doing (writing code, building graphs, creating apps, etc.) so it’s discoverable. This way, if a future collaborator, prospective employer, or up-and-coming analyst is looking around for cool stuff on the Internet, they’ll see what you’ve been doing.

Make cool sh!t and share it

We’ll introduce you to a few data scientists and journalists who are excellent communicators of their work. These examples use the Internet as a tool to engage with broader audiences, create better tools for doing science, document some of their daily struggles/successes.

Our first example, Lucy D’Agostino McGowan is a post-doc at Johns Hopkins Bloomberg School of Health. She maintains a blog, publishes ebooks, has online courses, and also attempts to create a real BB–8.

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

Or take Amber Thomas, the Senior Journalist-Engineer at the Pudding. She writes tutorials, maintains a blog, and spends her time “hanging upside down in aerial silks.”

All of these people have done two things very well:

1. **Created good work**: All created projects across multiple mediums, websites, and platforms
2. **Shared with as many people as possible:** these examples made their projects discoverable and collaborated with the data science community (by putting their work online for people to find)

Of course, they also had to know their subject areas, and have something worth sharing. But they didn't wait until their work was perfect, or until they were done with their careers to share and get feedback. They started engaging with people while they were working to show how their work gets done.

'Good enough' communication

It's important to remember that whenever you're trying to communicate something, you're convincing your audience that they should be paying attention to you instead of everything else. Today, there's a lot more 'everything else.'

Approaching communication this way puts you in the mind of your audience and keeps you asking, "*why would they want to know this?*" Try to keep the value of what you're saying visible to your audience.

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 book 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.

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, the better you'll get at finding an ability to convey why what you have to say is essential.

When we were kids in math class, most teachers required us to show how we got the answers ("show your work"). Teachers told us 'show our work' so they could follow our thought processes through a problem, and see where our thinking was incomplete or mistaken.

If you're regularly sharing your work, people can follow your line of thinking as you progress throughout your project. More importantly, people who find your work will give you feedback and improve your ideas.

Get 'good enough,' then go for more if you need it

What is 'good enough'? We think being 'good enough' means reading about a tool or technology and being capable of distinguishing it from magic. Good enough also means seeing a chart or graph on the Internet and knowing how to evaluate its contents. Or imagining something that matters in your business or personal life, and then devising a way to count it.

Feedback

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

Footnotes

- check out the Data Carpentry lessons for getting closer to expertise
- The author of the paper above, Greg Wilson, has an excellent blog on teaching technology and all things awesome.

Part 2: “Have a workflow.”

Working with data **workbench** is a place to keep and organize tools, and a **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 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.

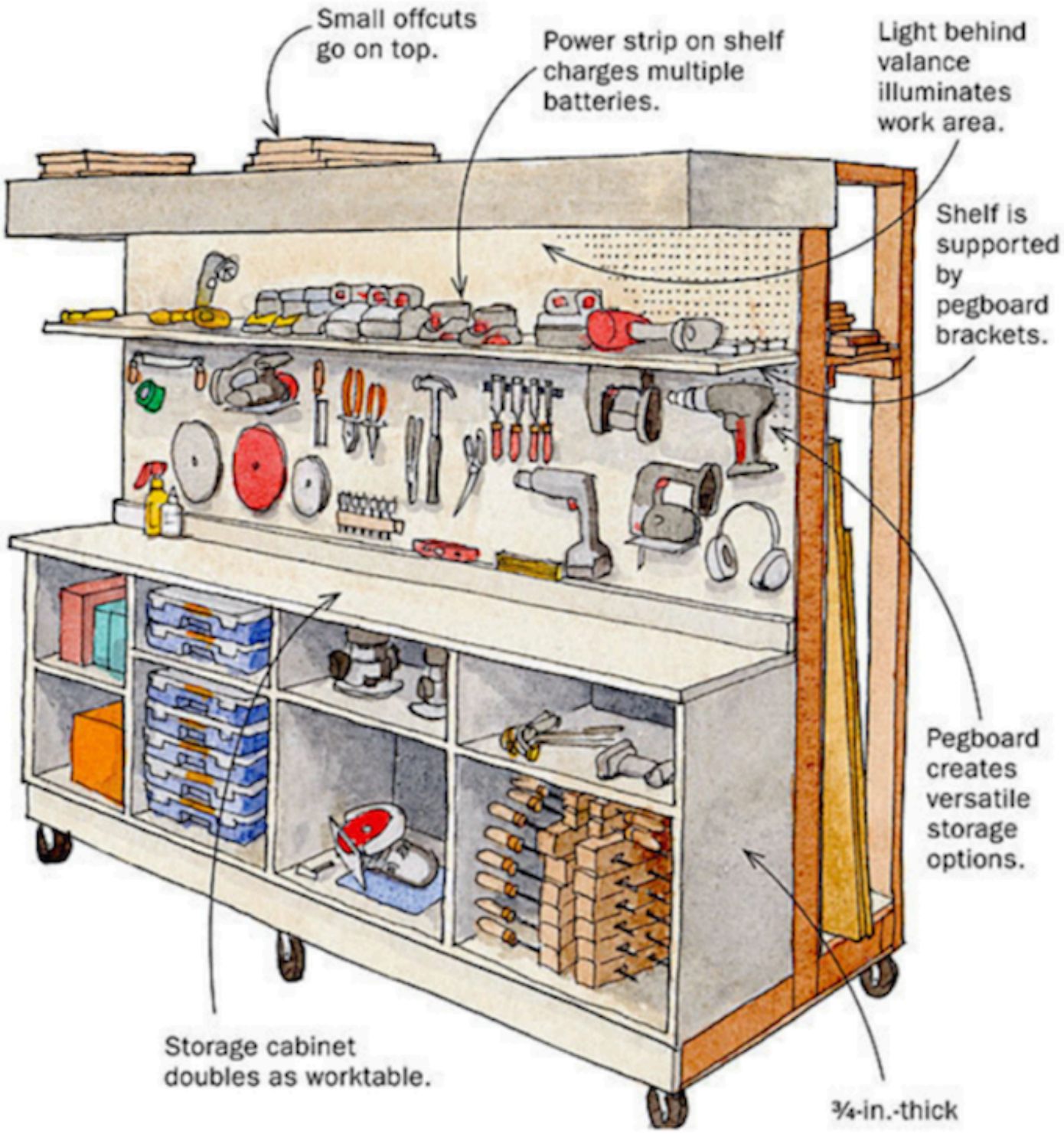
We’ve displayed some examples of open source tools for data management, statistics, and communication in the image below:

Follow the links below to learn more.

- Git
- Github
- Linux
- MySQL
- Netlify
- Python
- R
- RStudio

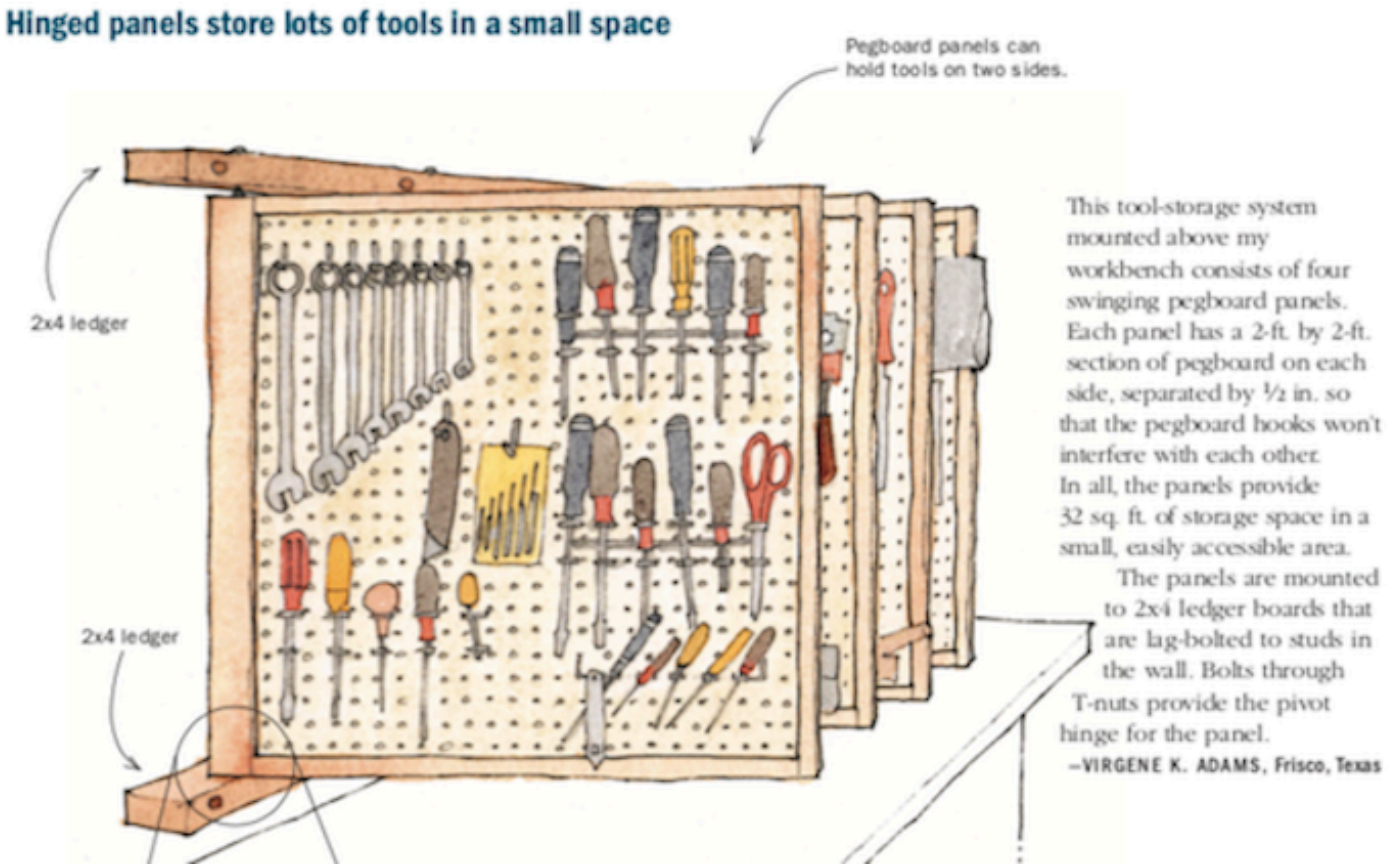
The integrated development environment (aka the workbench)

An integrated development environment (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).



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

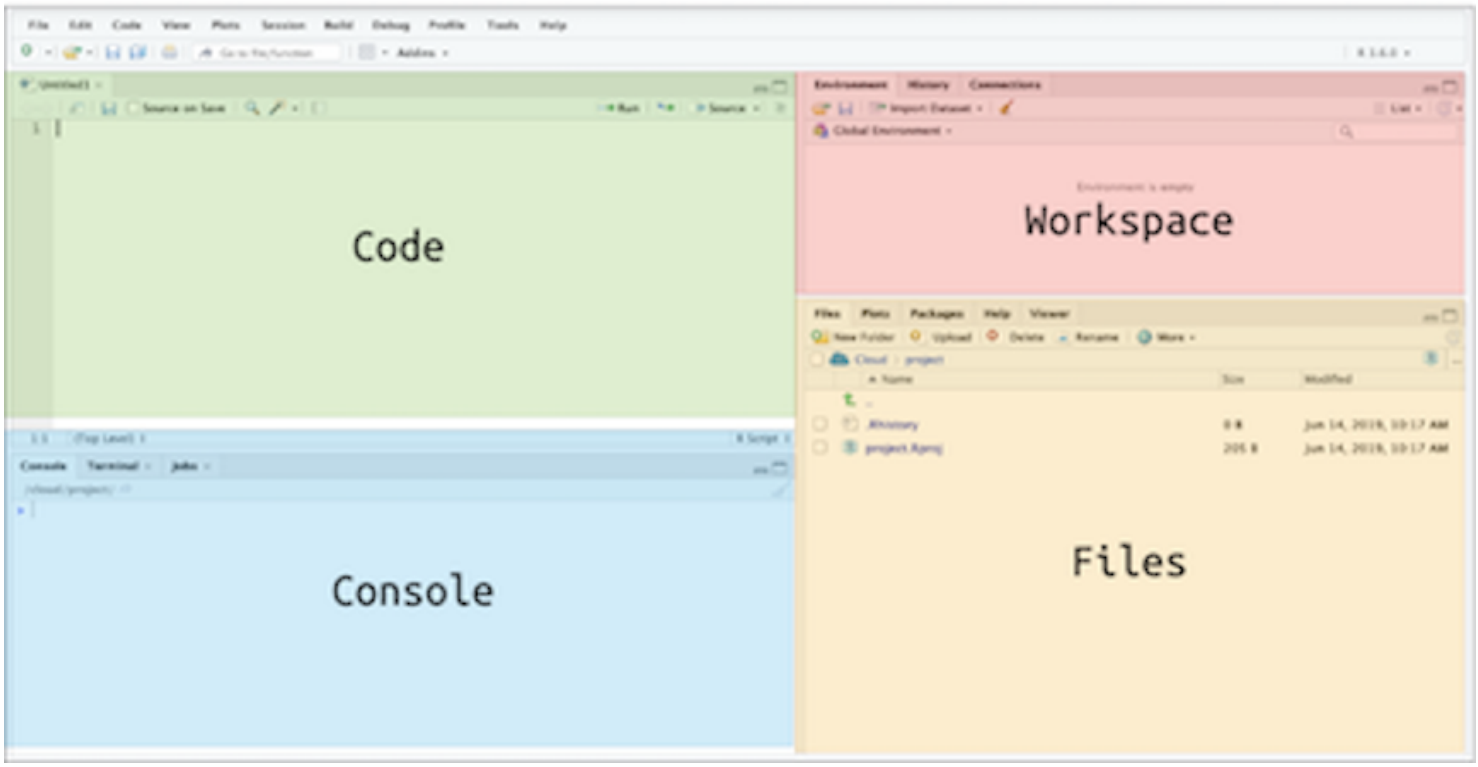
By making the panel positions adjustable, the workbench allows for easier access to more tools. Depending on the job, woodworkers can customize the panel arrangements with "*the simple pin that allows the rack's various faces to swing left/right for access to either side.*."

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 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.

RStudio Integrated Development Environment



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 tools we need for the job (and RStudio plays well with many open-source software tools!). For now, we are just going to focus on R and Git.

What is R?

R is a free statistical modeling software 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.
2. Second, download and install RStudio, the integrated development environment (IDE) for R

Use RStudio in the browser

An alternative to downloading and installing R and RStudio is using RStudio.Cloud 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.

What is Git?

Git 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.

What is Github?

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

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.

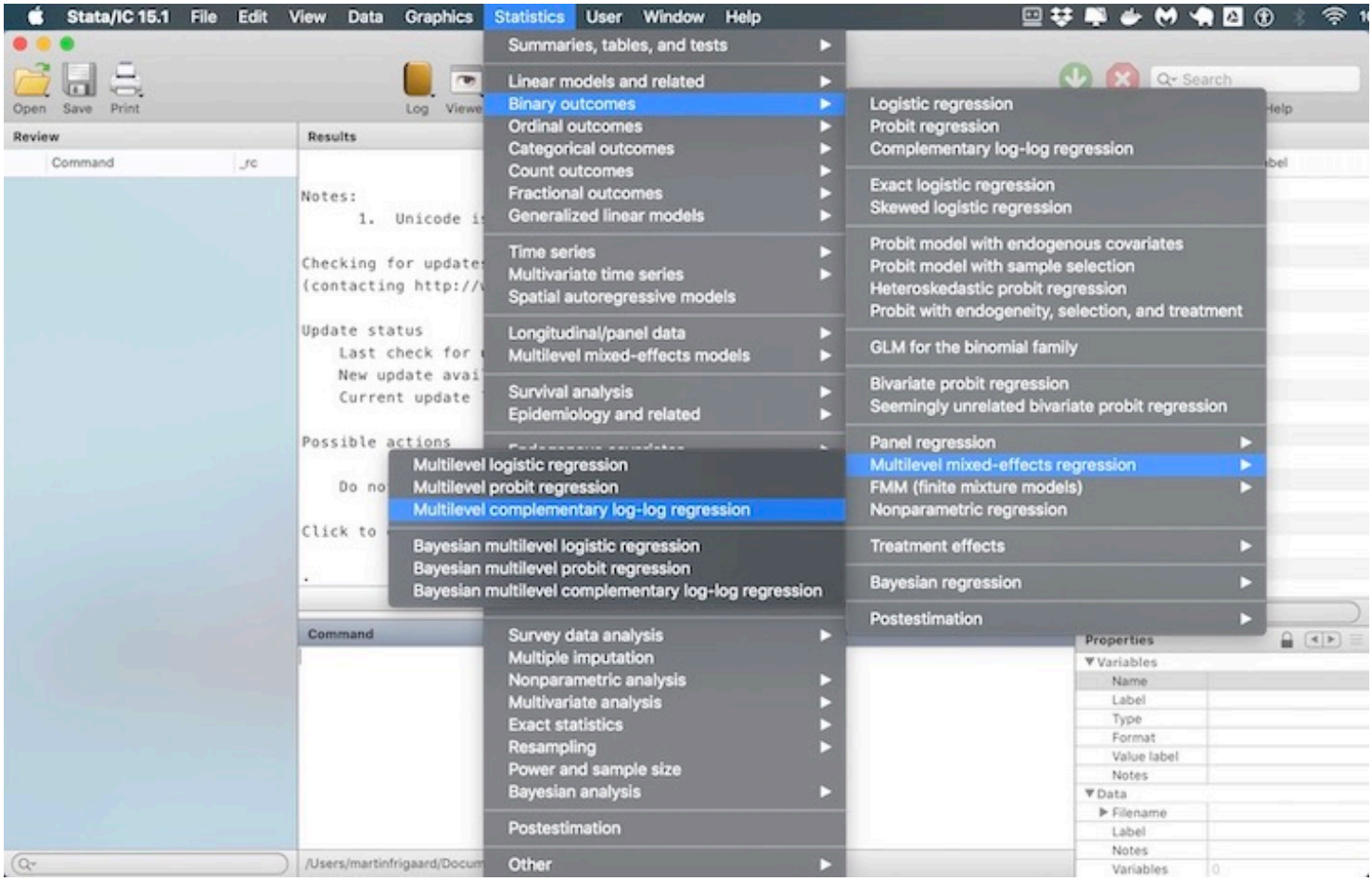
Note: *You should explore different IDE's on your own– you'll see there are many options, both paid and unpaid. We're confident you'll see RStudio is well suited to handle more than most of the things you'll want to accomplish.*

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

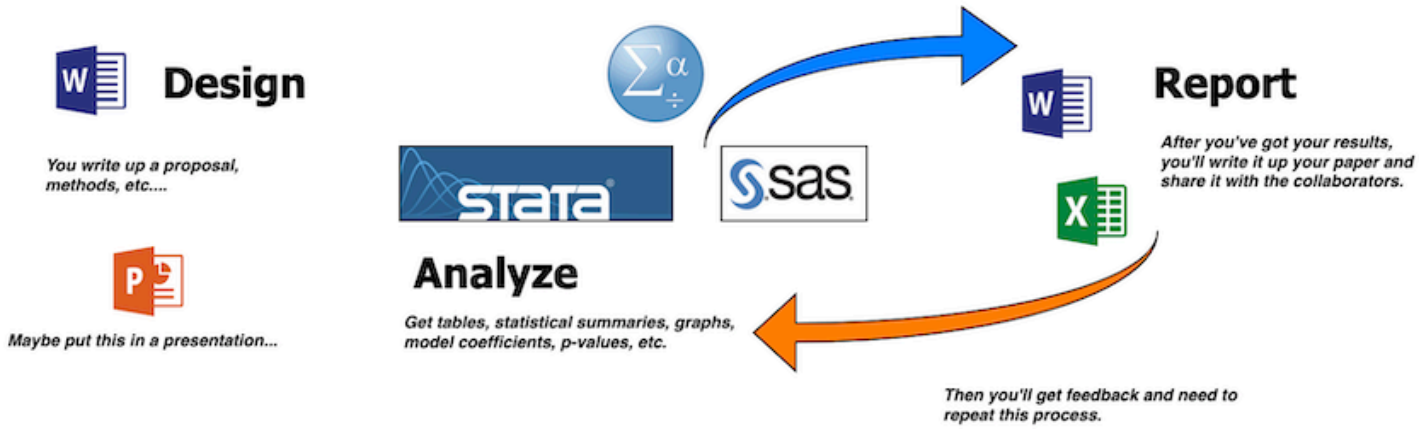
Principle 2: Write code

Usually, people interact with their computers using point-and-click graphical user interfaces 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 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.



As you can see from the image, you'll be required to go back through the same elaborate workflow whenever you receive feedback or input to your project. Each time will take 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 as a tool, not a solution

We think of data science software as the tools that help us gain a deeper understanding of the world. We don't think of software as an alternative to thinking or expect a software tool to do our thinking for us. We also prefer tools that we can reuse on future projects.

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 don't recommend relying too heavily on point-and-click proprietary software applications (SPSS, Stata, SAS, etc.). These GUI's make it hard to keep track of what you've clicked on, the order of you clicked on them in, and can oversimplify or obfuscate what's going on.

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.

> ***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*

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.

Data science jobs need a particular set of tools, and a workbench to organize these tools. To manage your data science projects, you'll need a workflow that gives you the ability to 1) document your intentions and, 2) write code that can translate your plans into something a computer can execute. These two points bring us to our next topic: plain text. 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, authors Hunt and Thomas advise *'Keep[ing] Knowledge in Plain Text'*. This sentiment has been repeated here, here, and here.

We recommend you keep your files, notes, and any pertinent documentation about your project in plain text 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 searchable (`ctrl+F` or `cmd+F` allows us to find keywords or phrases)

We'll also cover why you might want switch over to a plain text editor if you're currently using 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

If we've sold you on using this flexible and adaptable tool, but you're still wondering what makes a file 'plain text,' we'll cover that next. But first, we need to talk about what *isn't* a plain text file.

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"*

What *is* plain text

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

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

The last bit is the most crucial distinction–**human-readable vs. computer-readable**. In this manual, we'll point out which files are binary and which are plain text. Generally speaking, plain text files can be opened using a text editor or viewed with a command-line tool. Examples of text editors include Atom, Sublime Text, and Notepad++.

Markdown & Rmarkdown

Markdown files (`.md`) are common type of plain text files. Markdown is a 'lightweight markup language,' which means it's easy for humans to read, and computers can convert it to HyperText Markup Language (HTML). Markdown allows for some formatting options to aid with communication (see below)

```
<!-- comments -->

normal text

*italic*

**bold**

> quote

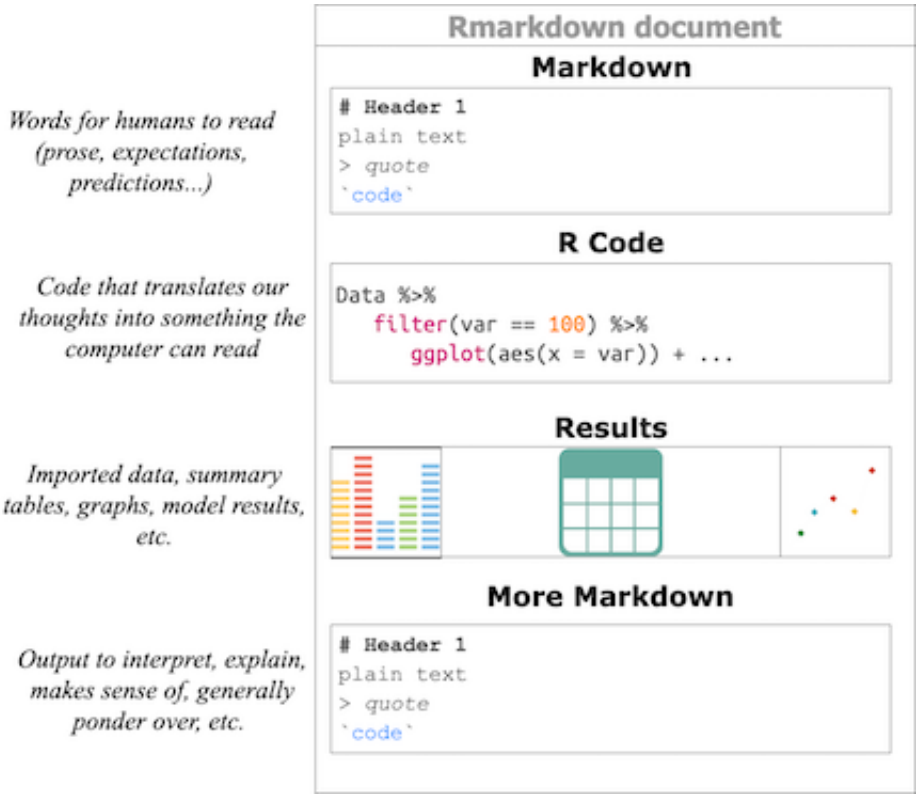
`code`

# h1
## h2
### h3
#### h4
##### h5
###### h6
```

To learn more, see Markdown Syntax Documentation on John Gruber's site).

We recommend learning markdown before any other programming language because it's the lingua franca for asking questions. Stackoverflow, RStudio community, Reddit, Github, and many other sites use markdown to post questions and answers. We recommend experimenting with StackEdit, a browser-based markdown editor that gives you the ability to write in markdown and see the syntax rendered as HTML.

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



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 for more information. The image below is an output from an `.Rmd` document in RStudio.

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)
```

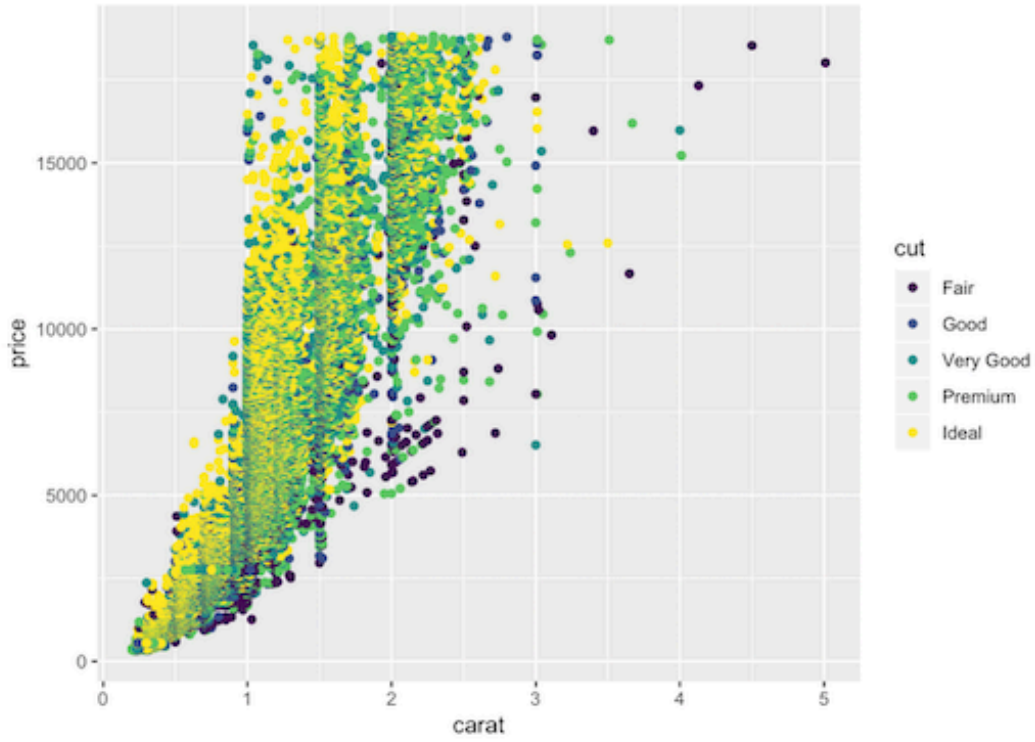
```
## Observations: 53,940
## Variables: 10
## $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23,…
## $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very Good
```

```
## $ 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 & Jupyter vs. R & RStudio

Python is a neat language and a great tool to combine with R.

It's also helpful to know a little Python even if you're primarily working in R, because 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'd have *a minimum of five applications open* to do data analysis. I would be using Word to write, Stata for statistics, Excel to create tables, the browser for internet research, and Adobe Acrobat for reading PDFs. That means I needed to learn five different GUIs, each with their design characteristics.

Each software application cost me valuable neurons whenever 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,

> *"[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 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, dialog 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", 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):
  1) Why is it so hard to do my work? The challenge of attention residue when switching between work tasks
  2) Information, Attention, and Decision Making
  3) Causes, effects, and practicalities of everyday multitasking
- See Baumer et al. for an in-depth summary of why you should abandon a copy + paste workflow

Part 3: Setting up your data science project

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

In the last chapter, we recommended a workbench (RStudio) and a set of tools (R, Git, Github). Now 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 never covered the details of setting up a project (and 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.

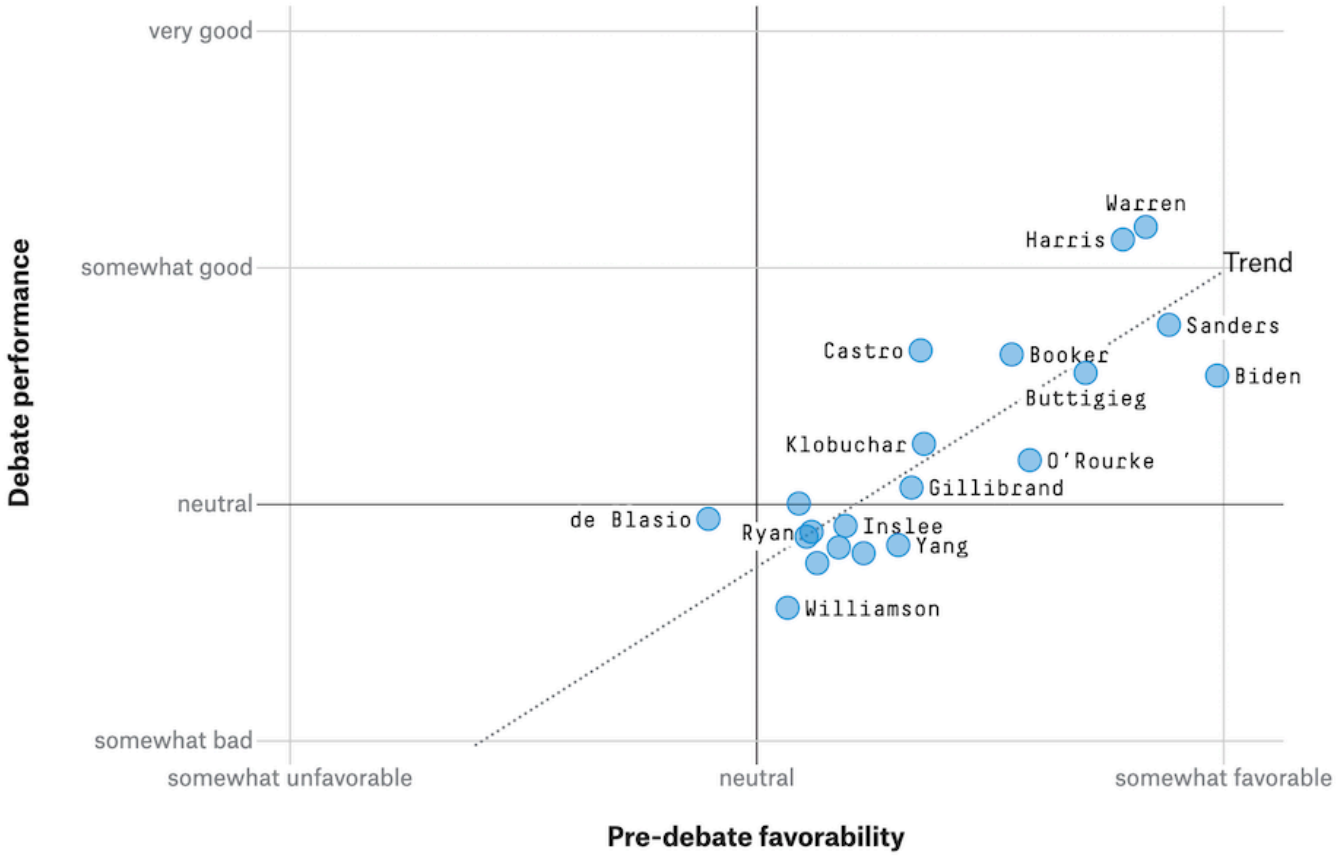
Example: FiveThirtyEight's 2019 Presidential debate project

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

The scenario we've described above might seem vague, but we want to show the power of these tools. We've also found some of the most exciting data projects are born from basic curiosity.

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

We stumbled across an article on the data journalism website `fivethirtyeight`, and it displayed an image showing the relationship between how voters felt about the candidates before the debates, and how the candidates did in the debate.



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

Journalists are a bit like statisticians in the sense that both get to "play in everyone's backyard". Data journalists explicitly combine analysis and communication skills. Marrying these two skills makes data journalism an extraordinary place to look for tools and methods to adapt to different projects.

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

Another reason journalism is an excellent resource for sharing your work is that journalists are trained to view the world differently than typical scientists or analysts. As the NBC investigative reporter Andy Lehren describes in the text Digital Investigative Journalism,

> *"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."*

Journalists can communicate *why something matters*, which is a great skill to hone. Explaining a data project to someone with zero domain expertise (or data science knowledge) is a great way to practice your communication skills. It's also a great way to make sure you're thinking about an audience for the project.

A collection of modern data sources

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` package for R gives us access to Google search terms and trends. We're going to import data from Google searches before and after the night of the debates.

2) `rtweet` 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 and the package has excellent documentation (see here and here).

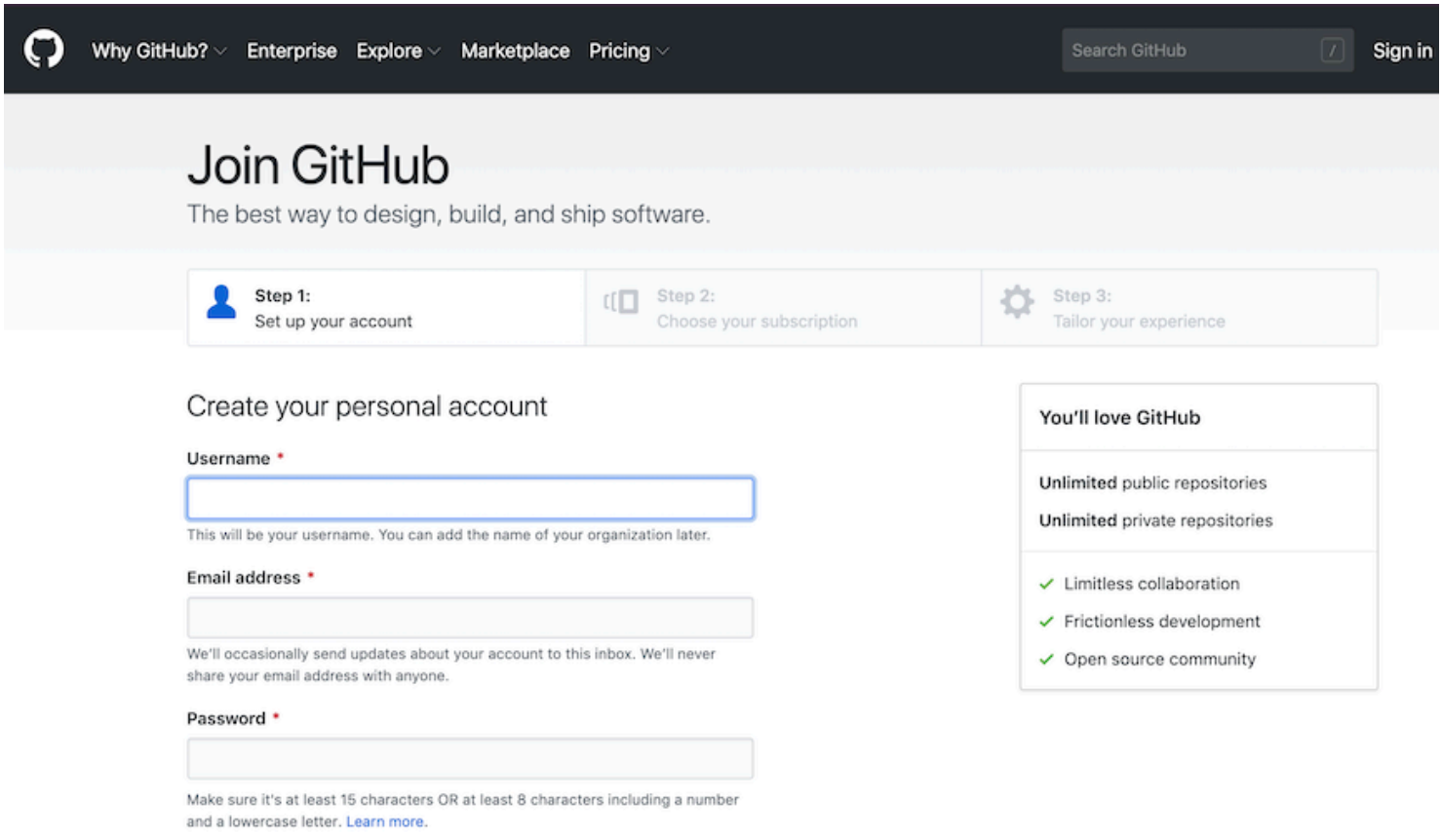
3) There is a Wikipedia page dedicated to the debates. We'll be scraping the tables with airtime for a candidate using the `xml2` and `rvest` packages.

4) Finally, we also have some data from voters on how they felt about each democratic candidate going into the debates. These data are in a Google Sheet, and we've used the `datapasta` package and copy + paste these data into R. Another option is the `googlesheets4` package in R (*you will need to copy this sheet into your Google drive to get this data set*).

Step 1: Github

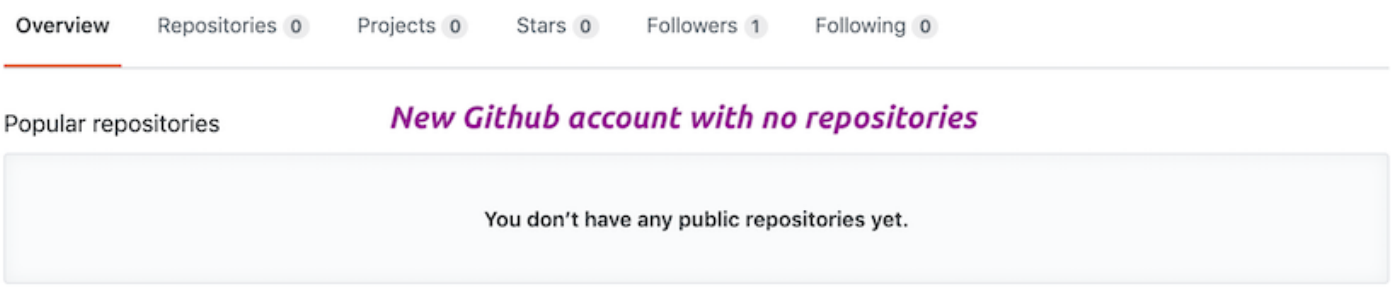
In this example, we will be using an RStudio.Cloud environment to perform the analyses. All of these steps can be accomplished using the RStudio IDE on your local desktop, too.

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



Join Github

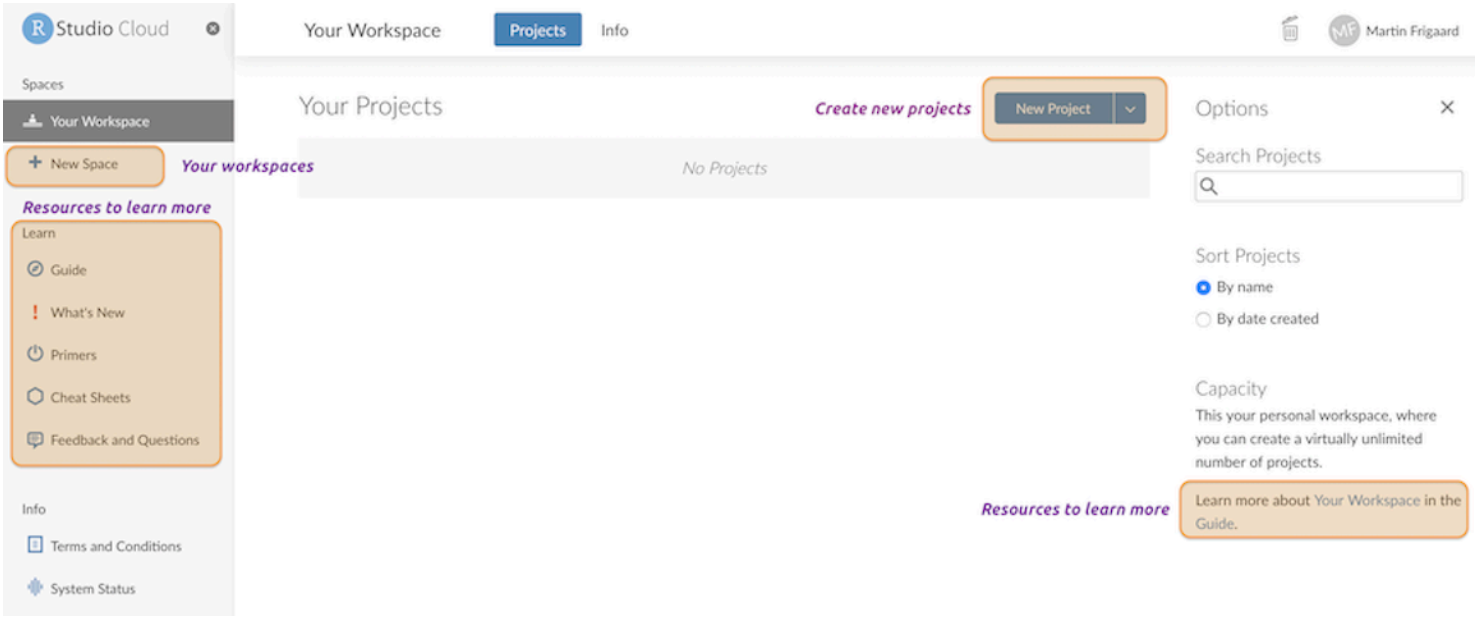
After you've completed the necessary forms (*remember you only need a free account!*), you should see a page with a message telling you "**You don't have any public repositories yet**".



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 in to RStudio.Cloud. After we're all signed in, we will see the screen below:

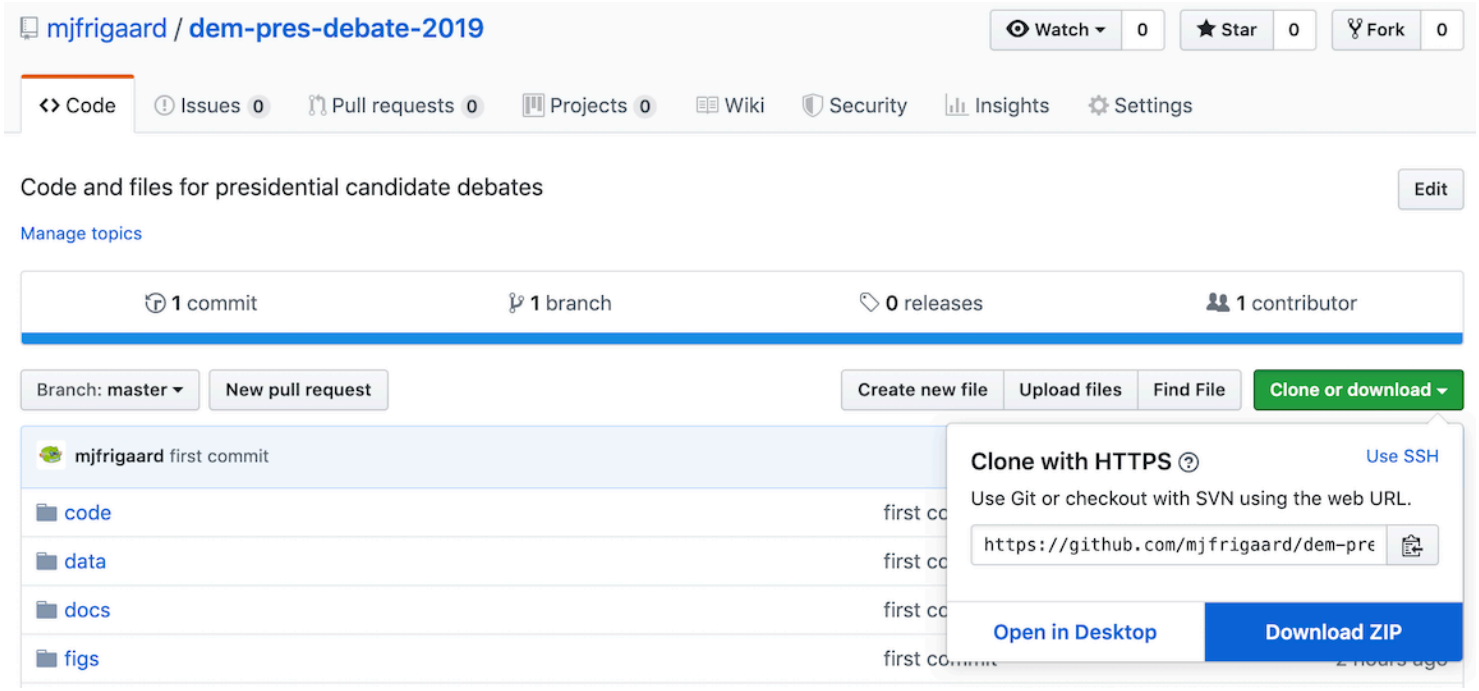


RStudio.Cloud environment

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

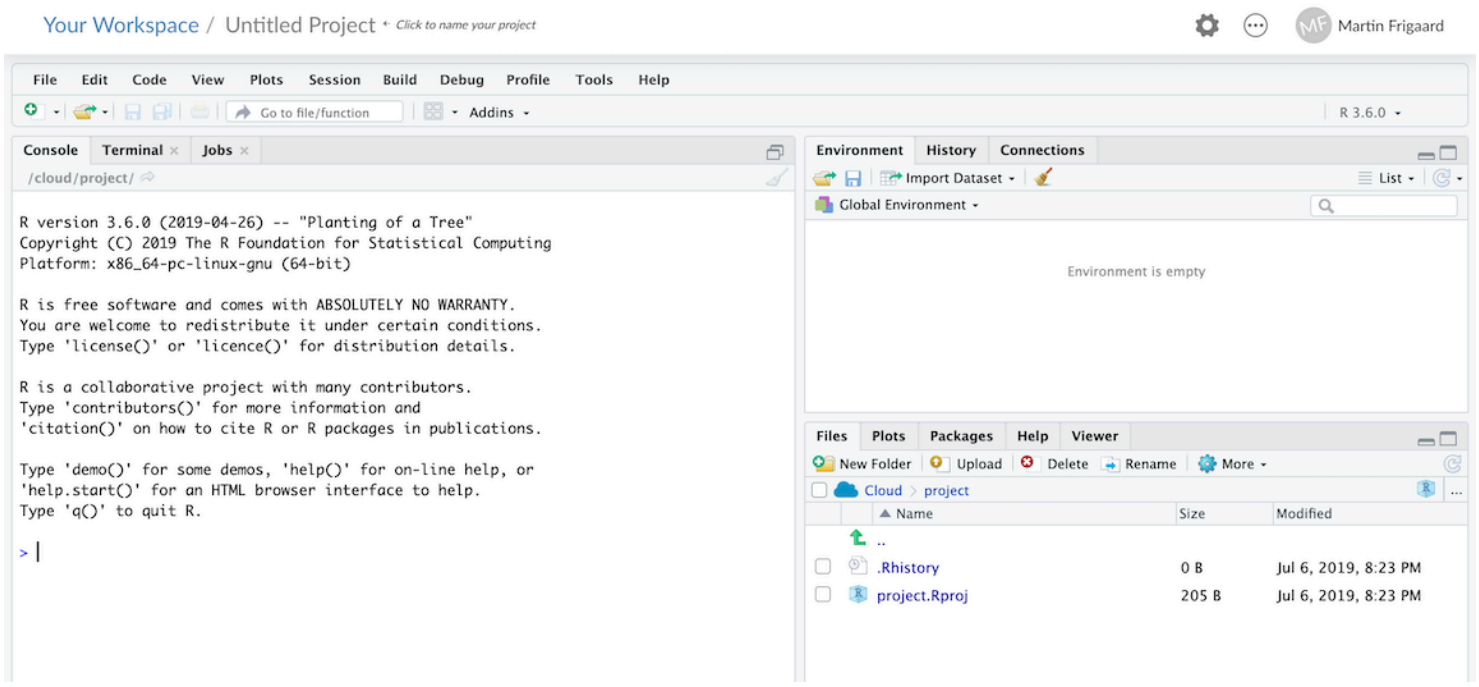
Most of the repos on Github are free for us to download and use. We can do this by clicking on the green **Clone or download** button and click **Download ZIP**.



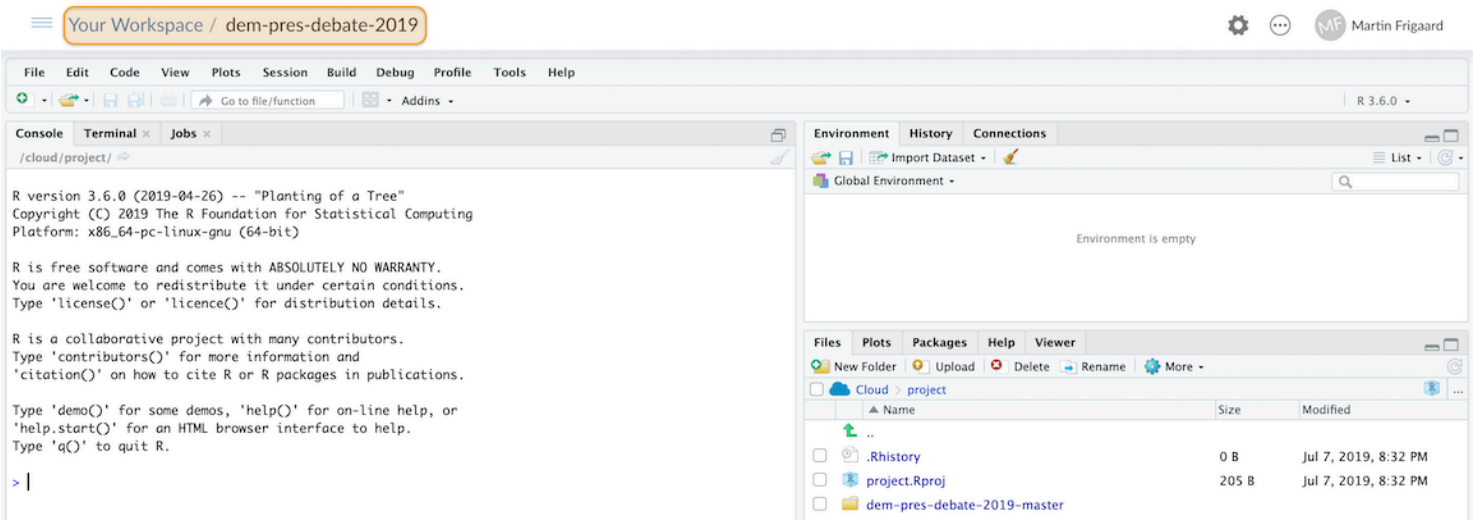
Pick a location on your computer to put your project and download the zipped Github folder.

Step 4: Upload files into RStudio.Cloud

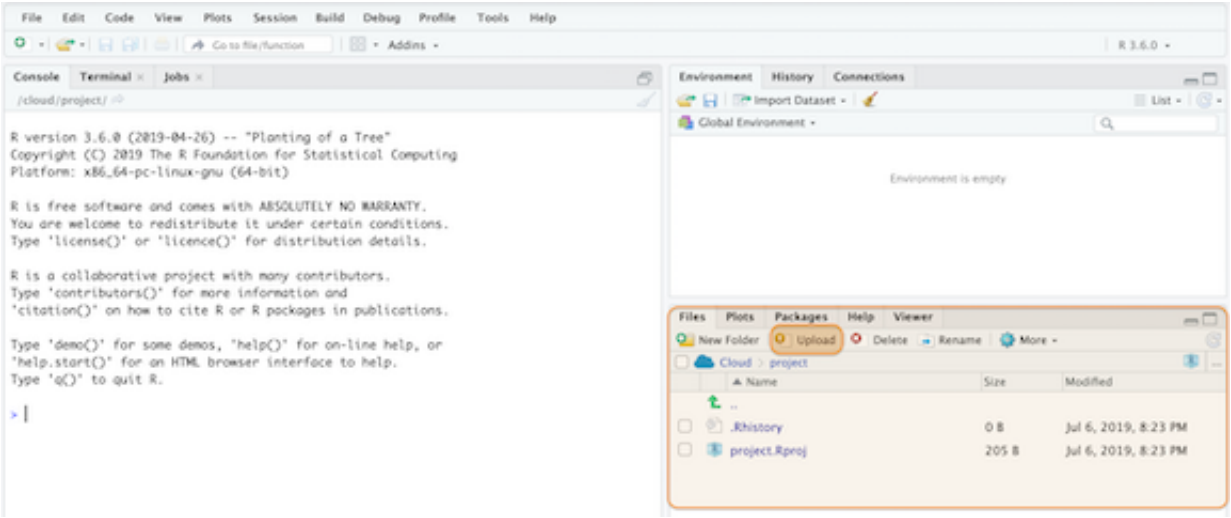
Back in the RStudio.Cloud browser, we're going to click on the **New Project** button. It should display the RStudio IDE in the browser like the image below:



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



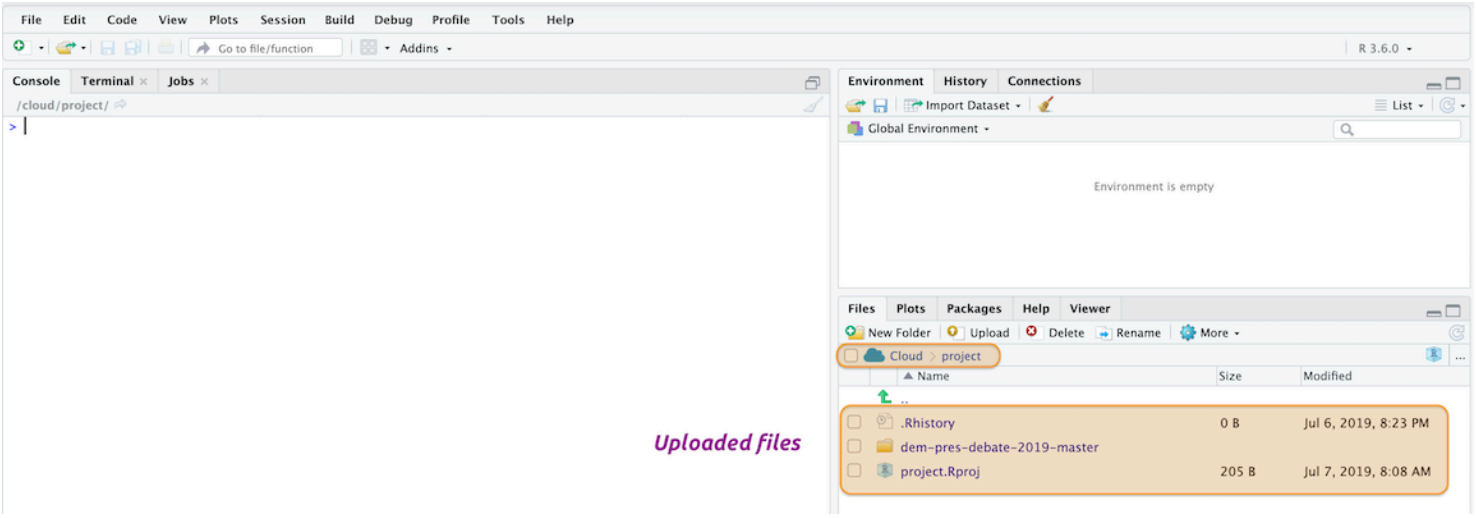
Look at the **Files** pane in the lower right corner and click on the **Upload** button, then click on **Choose files** and locate the recently downloaded zipped Github folder. Upload this file into the RStudio.Cloud project workspace.



upload button

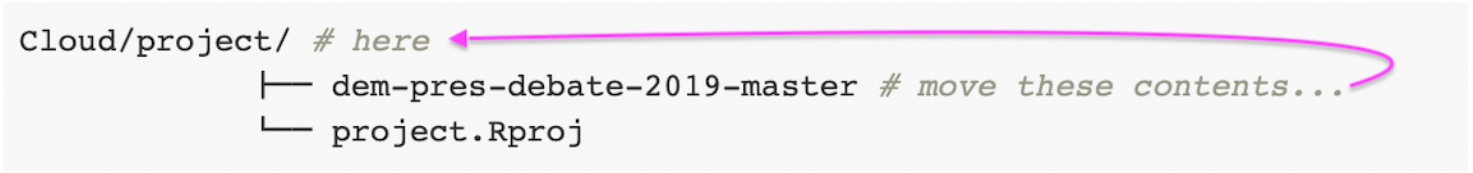
Accessing files in RStudio.Cloud

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.



the newly 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).



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' on operating systems, some **Terminal** commands, and how they work together.

The Command line: Unix and Windows

In 2007, Apple released its Leopard operating system that was the first to adhere to the Single Unix Specification. 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.

Command-line interfaces

The command line interface (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?"

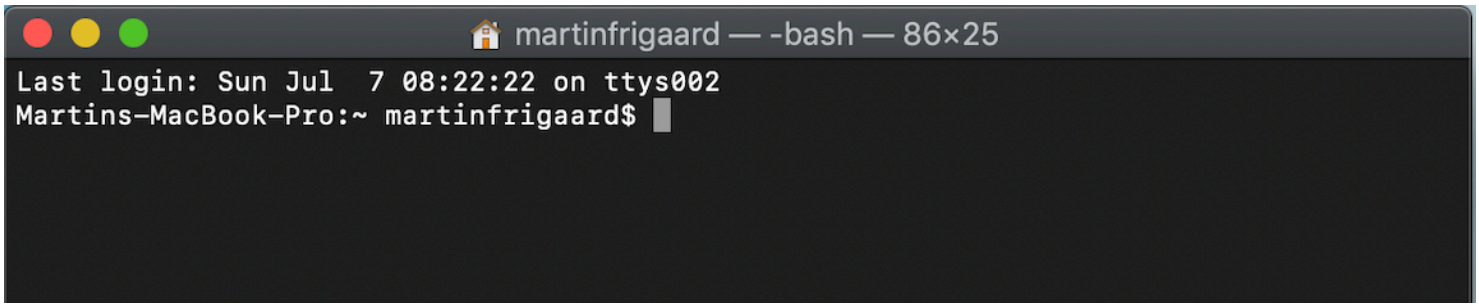
That's a fair question–being able to use the command line gives us more 'under-the-hood' access to any computer. We can use the command line to navigate a computer's directories (folders), install new programs or libraries, and track changes to files. It might seem clunky and ancient, but people keep 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 means each Unix command or tool does one thing very well (or DOTADIW)
- Modularity 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 made us more comfortable when we've had to interact with remote machines or different operating systems (Linux, per se). We will work through an example to demonstrate some of these features.

The Terminal (mac0S)

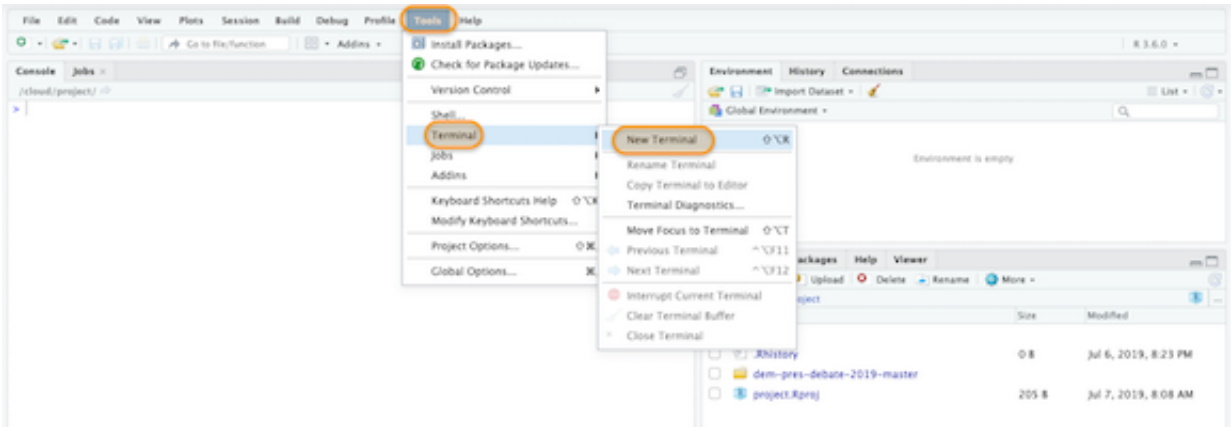
Below is an image of what the terminal application looks like on macOS. On Macs, the Terminal application runs a bash shell, which is why you can see the `bash -- 86x25` on the top of the window. Bash is a commonly-used shell, but there are other options too (see Zsh, tcsh, and sh). *Fun fact: `bash`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



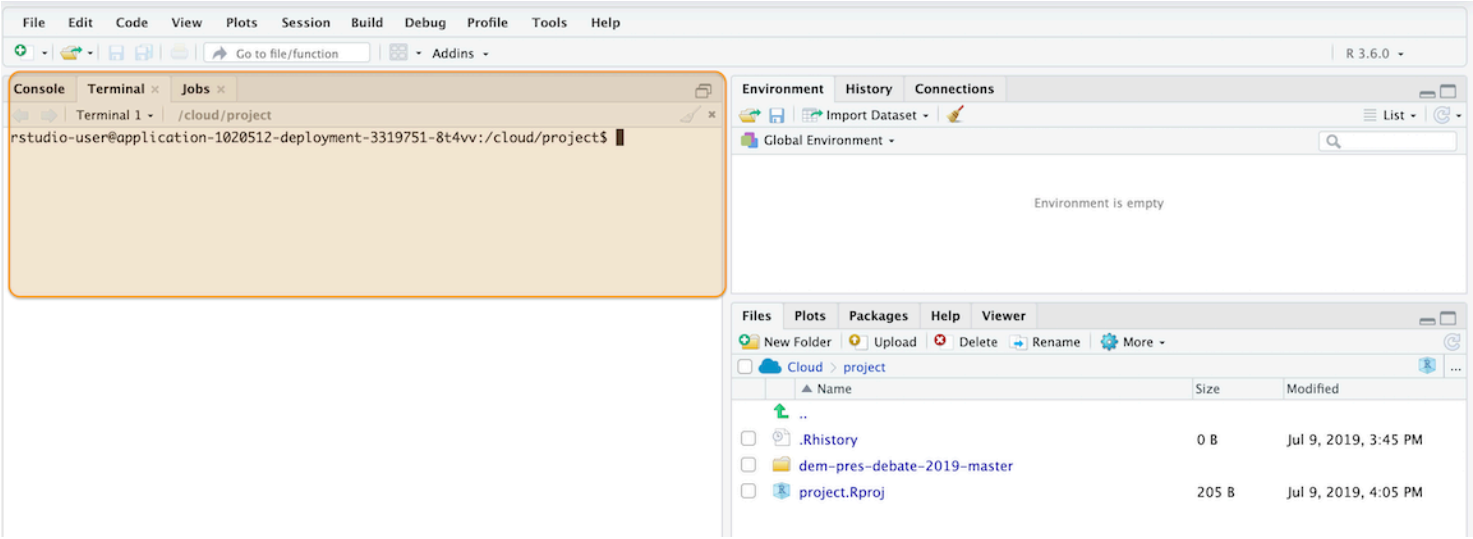
The Terminal is an emulator application for Mac users. Terminal is available as an application under the **Applications** > **Utilities** > **Terminal**.

The Terminal (RStudio)

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



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



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

Good enough command-line tools

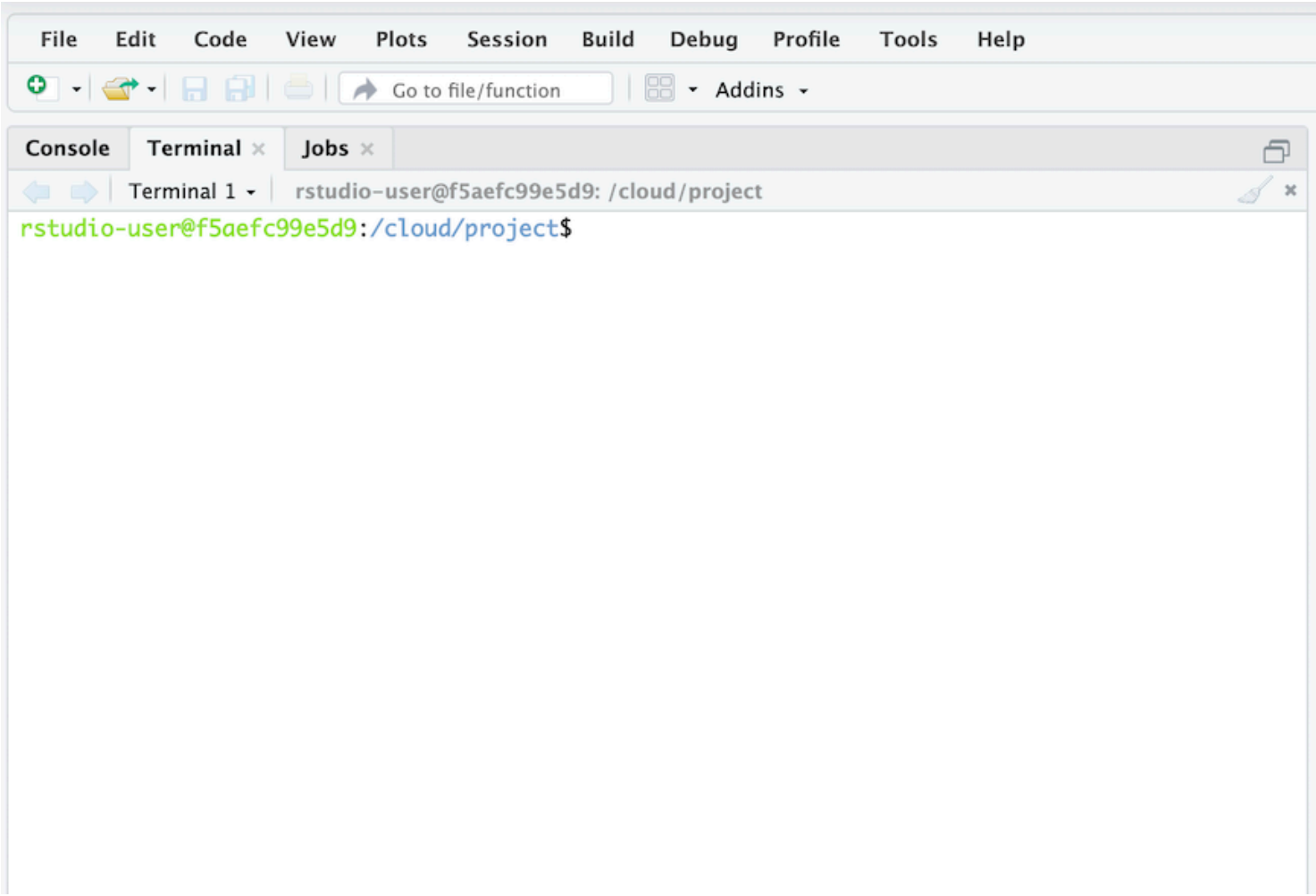
FAIR WARNING–command-line interfaces can be frustrating. Computers don't behave in ways that are easy to understand (that's why GUIs exist). Switching from a GUI to a CLI seems like a step backward at first, but the initial headaches pay off because of the gains we'll have in control, flexibility, automation,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

Here is a quick list of commonly used Terminal commands.

- **`pwd`** - print working directory
- **`cd`** - change directories
- **`cp`** - copy files from one directory to another
- **`ls`** - list all files
- **`ls -la`** - list all files, including hidden ones
- **`mkdir`** - make directory
- **`rmdir`** - remove a directory
- **`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`** - "globally search a regular expression and print"
- **`>>`** 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

The figure above might look 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 `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 preceding 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
README.Rmd  README.md  code  data
dem-pres-debate-2019.Rproj  figs  project.Rproj
```

The output from `ls` shows me there are three sub-folders in the `dem-pres-debate-2019-master` folder (`code`, `data`, `figs`), three `.Rmd` files, one `README.md` file, 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
README.Rmd  README.md  code  data
dem-pres-debate-2019.Rproj  figs  project.Rproj
```

We can add the `-F` option to the end of the command to tell **Terminal** to list the files in the folder at the end of the file path.

```
ls dem-pres-debate-2019-master -F
README.Rmd  README.md  code/  data/
dem-pres-debate-2019.Rproj  figs/  project.Rproj
```

Now we can see the folders have a `/` forward slash at the end of their name to separate them from the other files.

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 on the last folder or file.

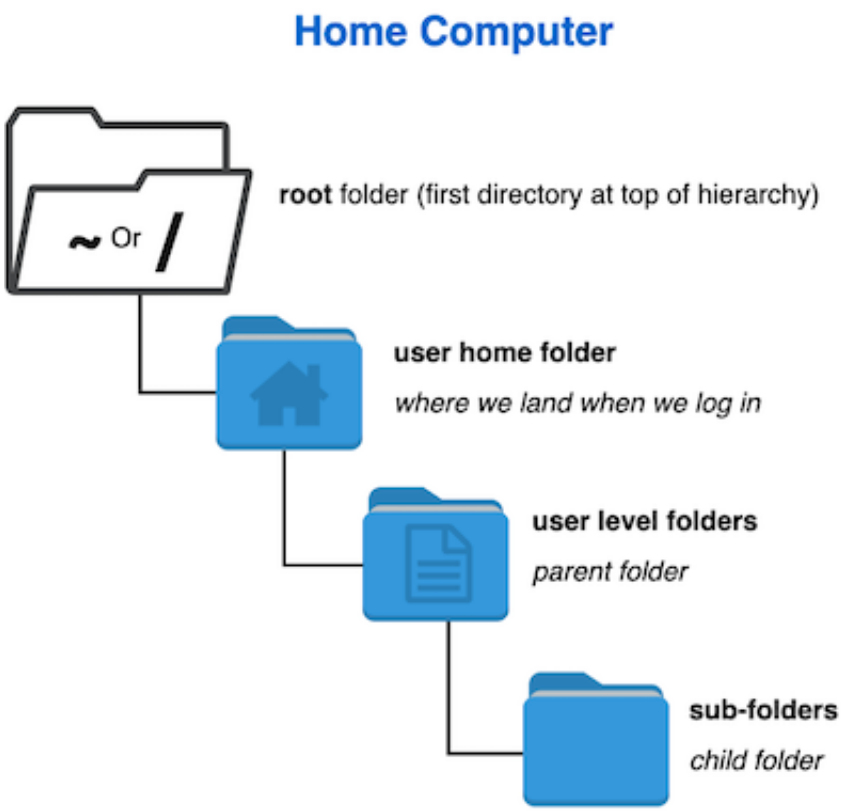
`/start/from/absolutely/where/i/tell/us`

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 "uppermost" 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 privileges, 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 with some standard default folders (`Desktop`, `Documents`, `Downloads`, and `Applications`)

Special considerations: Windows machines

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

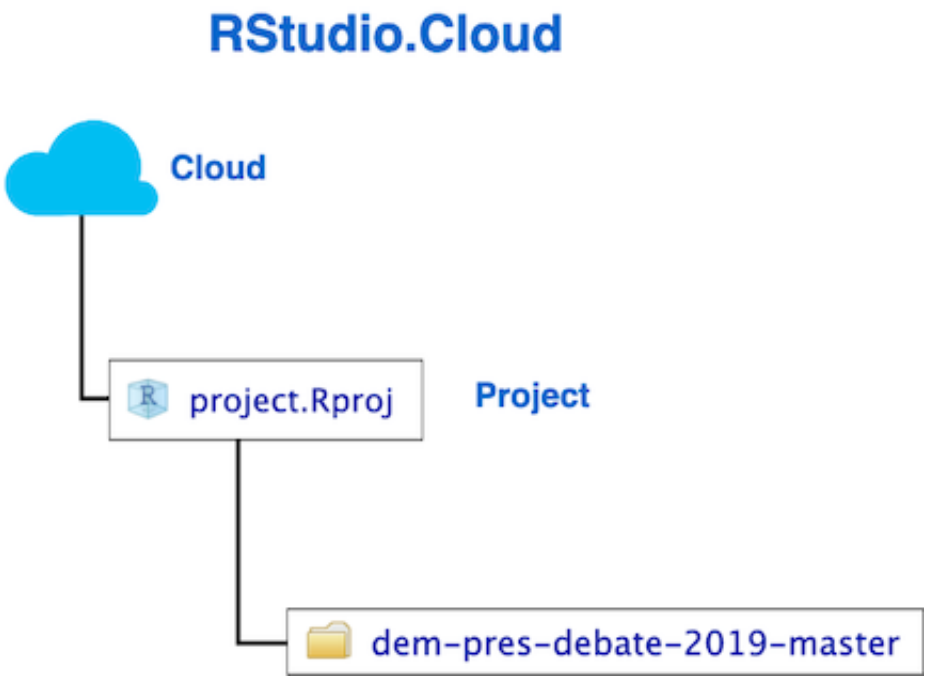
```
C:\Users\martinfrigaard\Documents\dem-pres-debate-2019-master
```

But we would need to write it like this:

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

This odd way of writing file paths is because, in R, the `\` is called an escape character, so 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 two options, `-v` and `*`. There are many other options for using `mv`, read about them here.

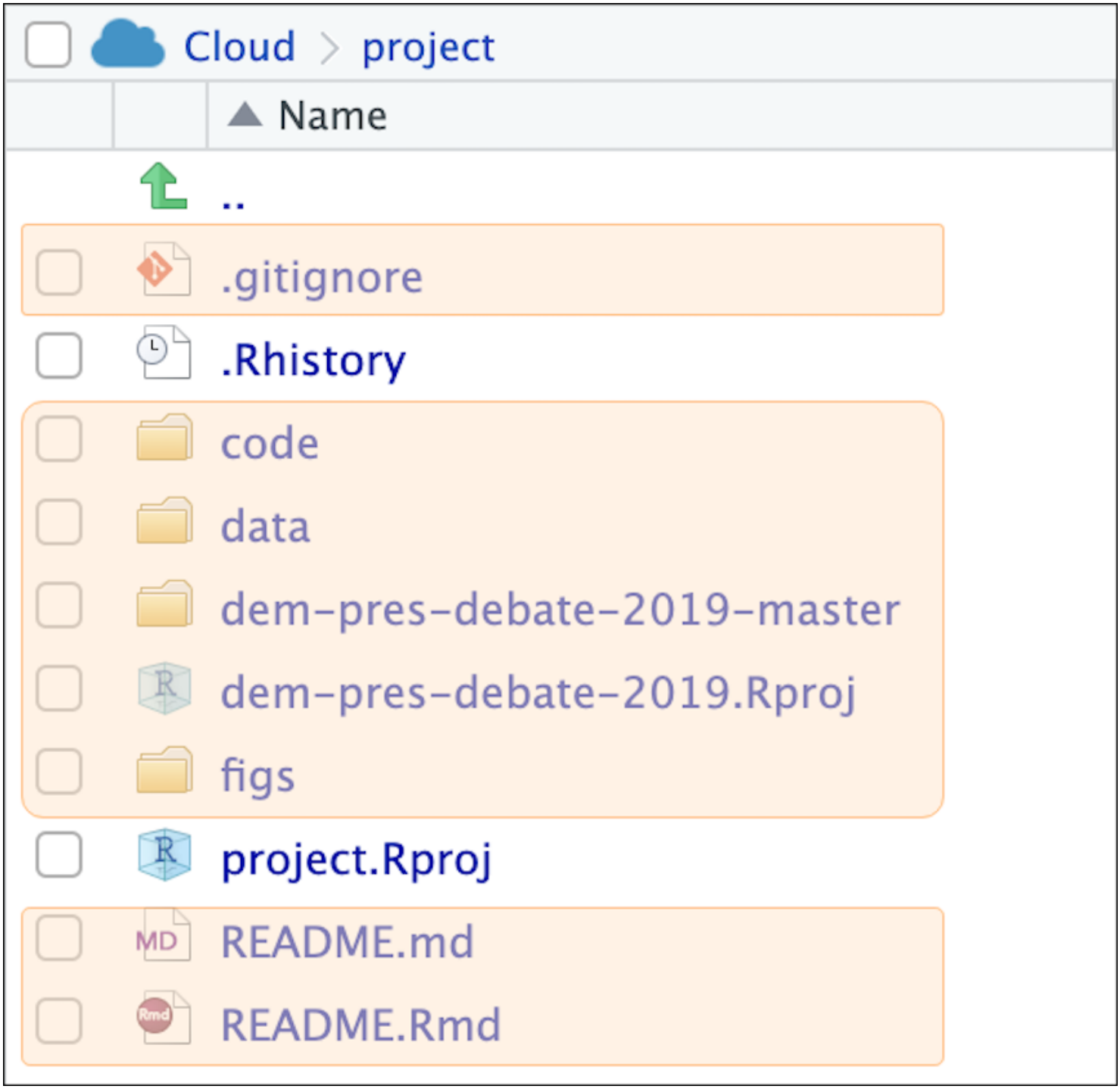
The sequence of commands we'll enter in the RStudio.Cloud Terminal are below:

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

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

```
'dem-pres-debate-2019-master/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-2019.Rproj
'
'dem-pres-debate-2019-master/figs' -> '/cloud/project/figs'
```

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



mv-ed files

Now we know we've successfully moved all of the files. But we will want to get rid of the old folder, `dem-pres-debate-2019-master`.

Command-line skill #5: Delete 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 with us before doing anything. 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 ourselves, but that's 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`

```
$  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.

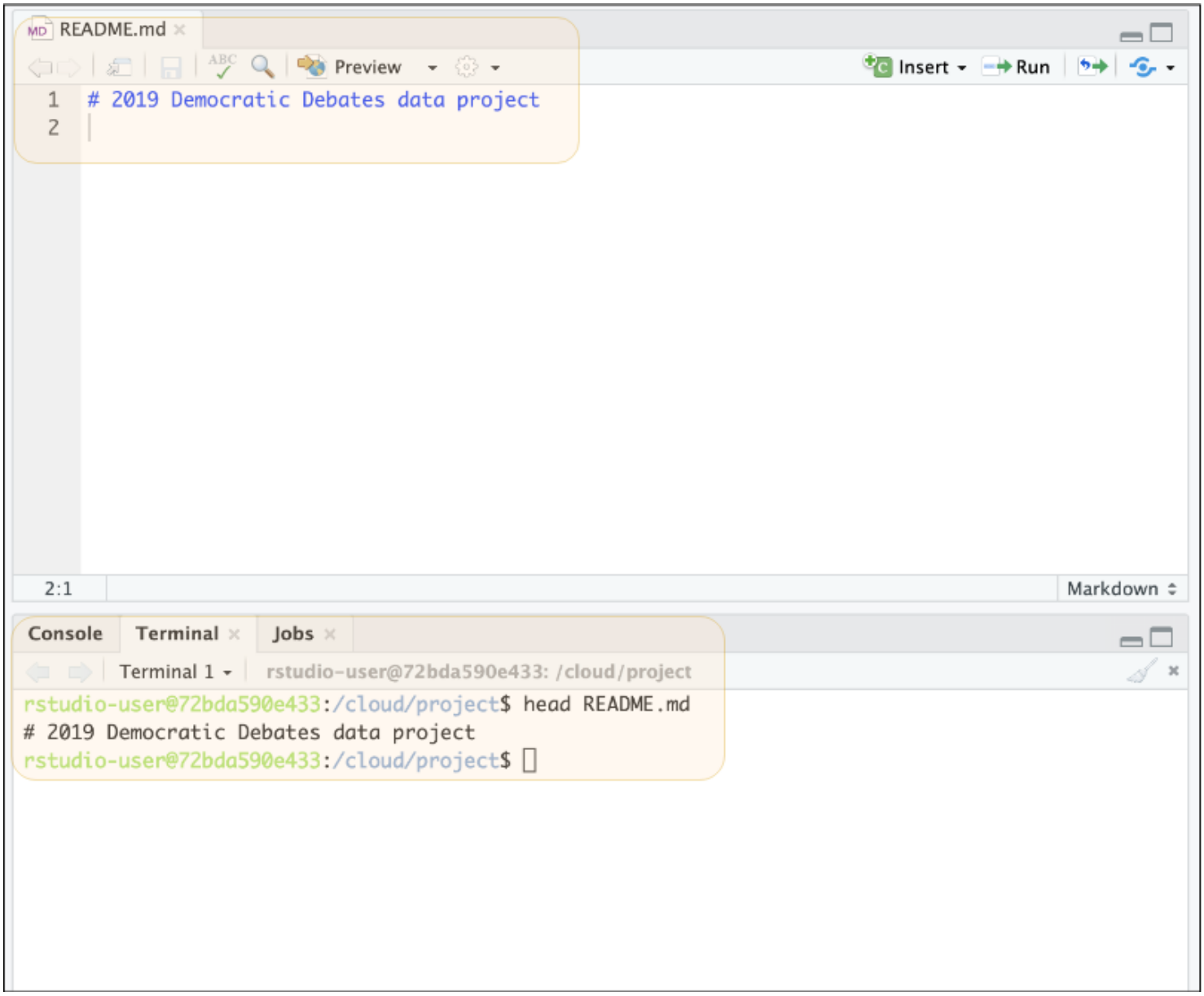
The **Terminal** pane should have the following contents when we're finished:

```
$ 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: Print 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 that 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 printing all the contents, we want to see how big the file is, and we can do that using `wc` (which stands for "word count").

```
$ wc README.md
# 1   6 39 README.md
```

The three numbers above are the number of lines (`1`), the number of words (`6`), and the number of characters (`39`).

`wc` is telling us that `README.md` won't be hard to read on the Terminal window. If it was, that's where the `less` command comes in.

```
$ less README.md
```

`less` will display the contents of `README.md` 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 `$(date)` (which 'attaches' the output from the command date with "some thoughts"), so we will repeat the process above, but include today's date with `$(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`.

```
$ cal
     July 2019
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

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

```
$ ls code | grep "download" | less
```

This should display the following result:

```
00.2-download-538.R
00.3-download-google.R
00.4-download-tweets.R
00.5-download-wikipedia.R
(END)
```

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

Other command line stuff: homebrew

The bash shell on macOS comes with a whole host of packages we can install with homebrew, 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` 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.

```
├── README.Rmd
├── README.md
├── code
    ├── 00.1-inst-packages.R
    ├── 00.2-download-538.R
    ├── 00.3-download-google.R
    ├── 00.4-download-tweets.R
    ├── 00.5-download-wikipedia.R
    ├── 01-import.R
    └── 02-wrangle.R
├── data
├── processed
└── raw
    ├── 538
    └── 2019-07-06-Cand538Fav.csv
    ├── google-trends
    ├── 2019-07-10-Dems2020Night1Group1.rds
    └── 2019-07-10-Dems2020Night1Group2.rds
    ├── twitter
    ├── 2019-07-06-Night01Tweets.rds
    ├── 2019-07-06-Night01TweetsRaw.rds
    ├── 2019-07-06-Night01TweetsUsers.rds
    ├── 2019-07-06-Night02Tweets.rds
```

```
    ├── 2019-07-06-Night02TweetsRaw.rds
    └── 2019-07-06-Night02TweetsUsers.rds
    └── wikipedia
        ├── 2019-07-10-WikiDemAirTime01Raw.csv
        ├── 2019-07-10-WikiDemAirTime02Raw.csv
        └── 2019-07-25-PollingCriterionRaw.csv
├── dem-pres-debate-2019.Rproj
└── figs
    └── 03-538-night-one-debates.png

9 directories, 23 files
```

Note that the `CHANGELOG.txt` file is not included in this tree (because the changes were made on a local repository). 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.

More on 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`. We didn't choose these folder names 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". If you already have an organization scheme, we still recommend reading at least this section of the paper–it's full of great information and links to other resources.

Getting more help with command-line tools

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, 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
- Data Science at the Command Line
- Software Carpentry Unix Workshop

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). For example, the GNOME is a desktop environment based on Linux which also has a Terminal emulator, but this gives users access to the Unix shell.

Part 4: Keep track of changes with version control

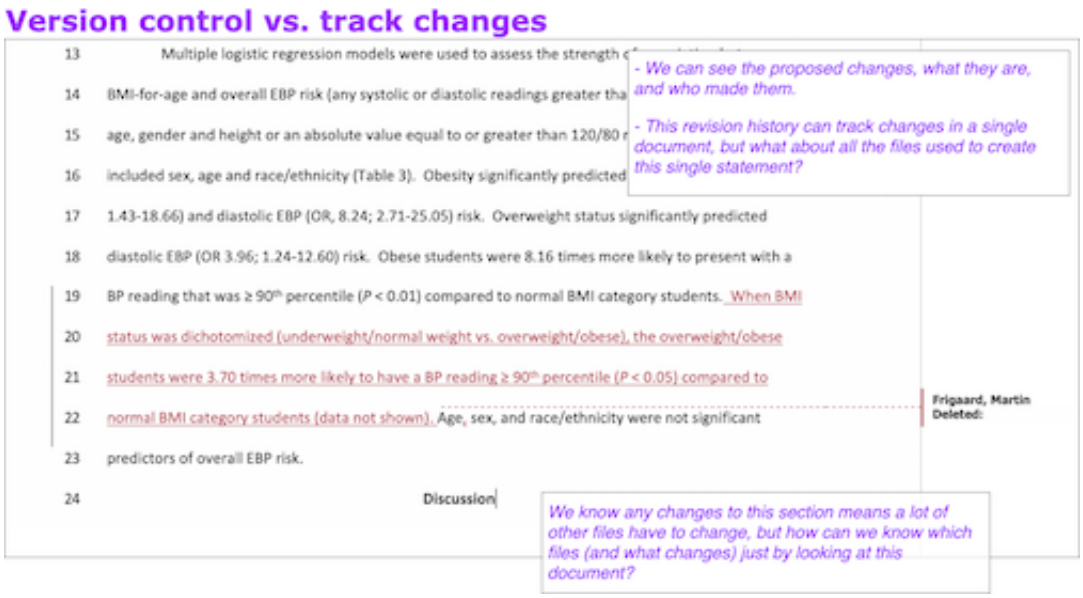
In the previous sections, we've covered using **Terminal** in the RStudio. If you're unfamiliar with these topics, please start there. This section will include tracking changes with version control, specifically Git, Github, and RStudio.

Tracing your steps

'Sharing your' can take a few forms. You can finish a project, then share your work for people to see and use in what they're doing.
Another option is to share what you're currently working on in a way that allows other people can collaborate with you along the way.

To accomplish the second option, we need a means showing how our work has changed over time. For example, maybe you've used the 'Review' tools in Microsoft Word, or you've collaborated in a Google sheet document. Both are types of version control because they're a formal system of managing changes to information over time.

Consider the image below of a .docx file:



The file is an earlier version of a manuscript. A coauthor has suggested changes to the results section. Sound version control systems let us see four aspects of changes:

1) what the differences are,
2) who recommended them,
3) the time/date of the proposed changes, and
4) any comments about the change

Unfortunately, tracked changes in Word only applies to a single document at a time. When you're working collaboratively (which is quite often), you know asking someone to change a single sentence can result in changes to dozens of files. That's why we need a way to track changes across a project's multiple files.

Git

Git is a version control system (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.

Why use Git?

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

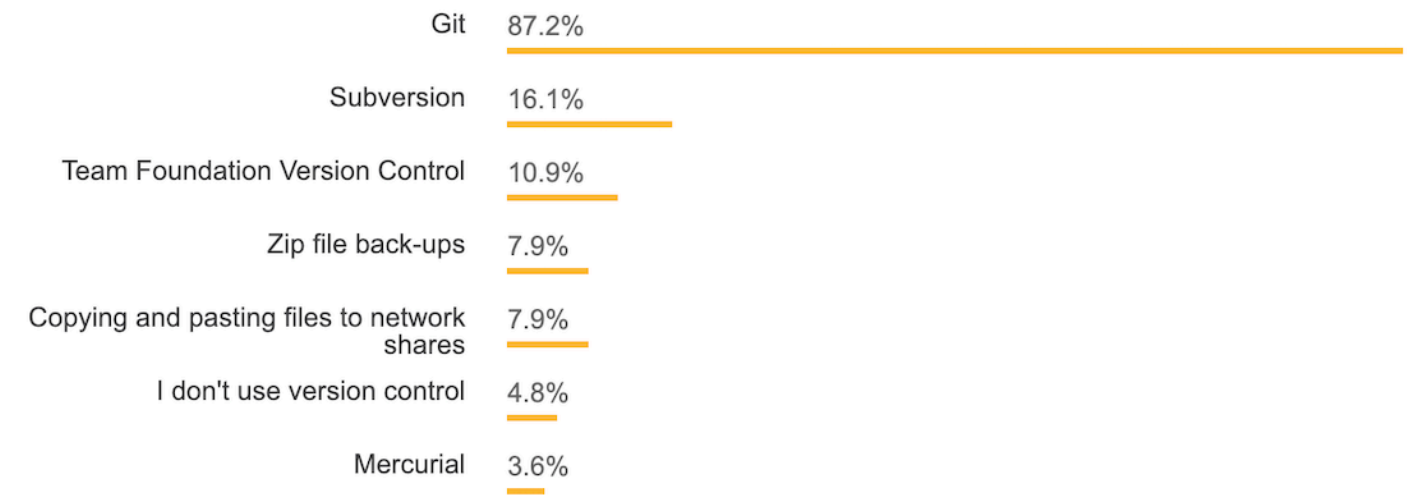
We've included these sections to remind you that you're making a sound choice.

Plain text + Git

Software developers keep track of their code with Git. We learned in the last chapter that most code files get kept in plain text, so Git plays well with plain text.

Everyone else is doing it

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



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

source: StackOverflow Developer Survey Results

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 'adds', 'commits,' 'pushes,' 'pulls,' and 'repos.'

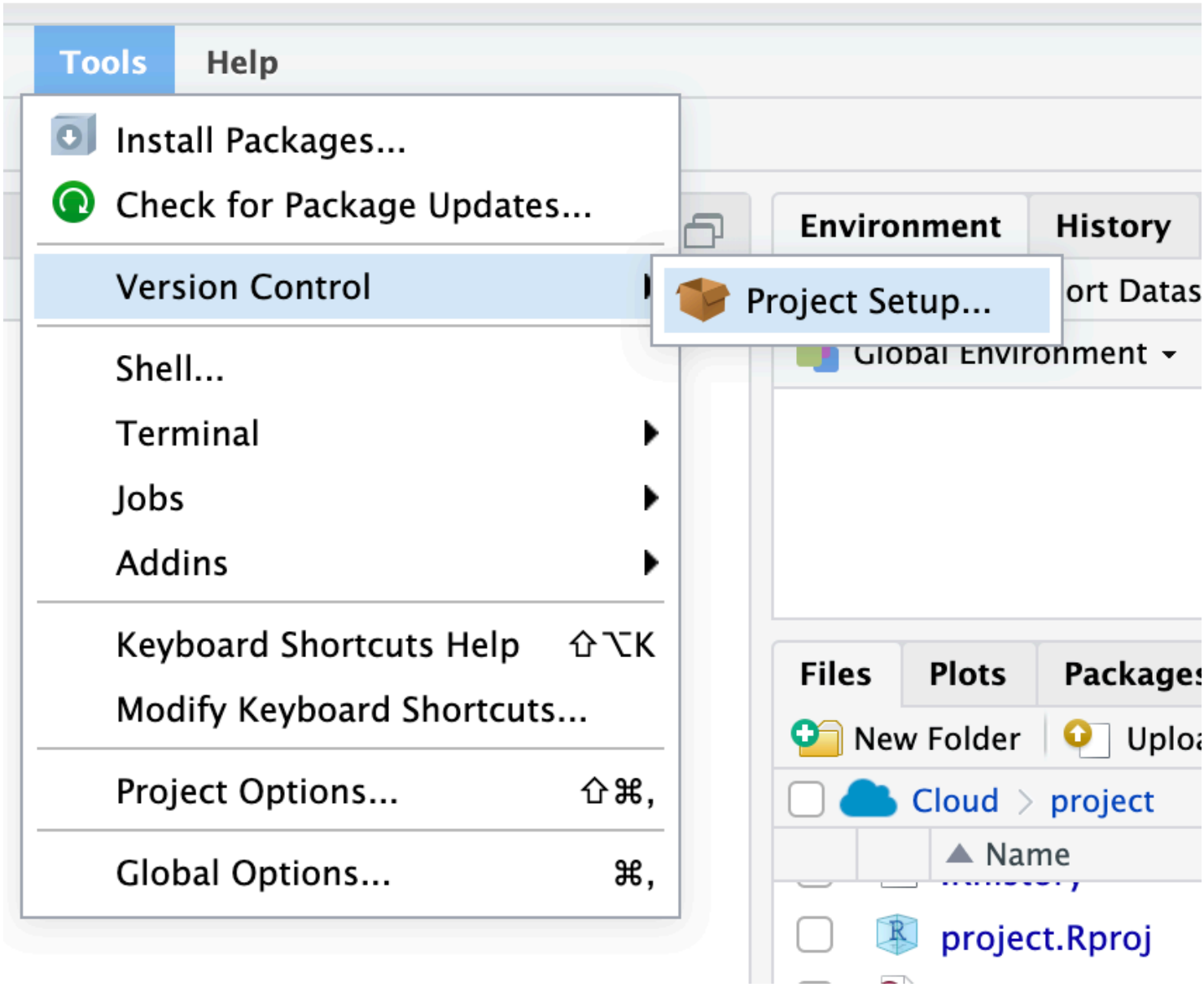
As someone who analyzes data regularly, these concepts are also countable, which means you can quantify change and work in more exciting ways.

Setting up Git

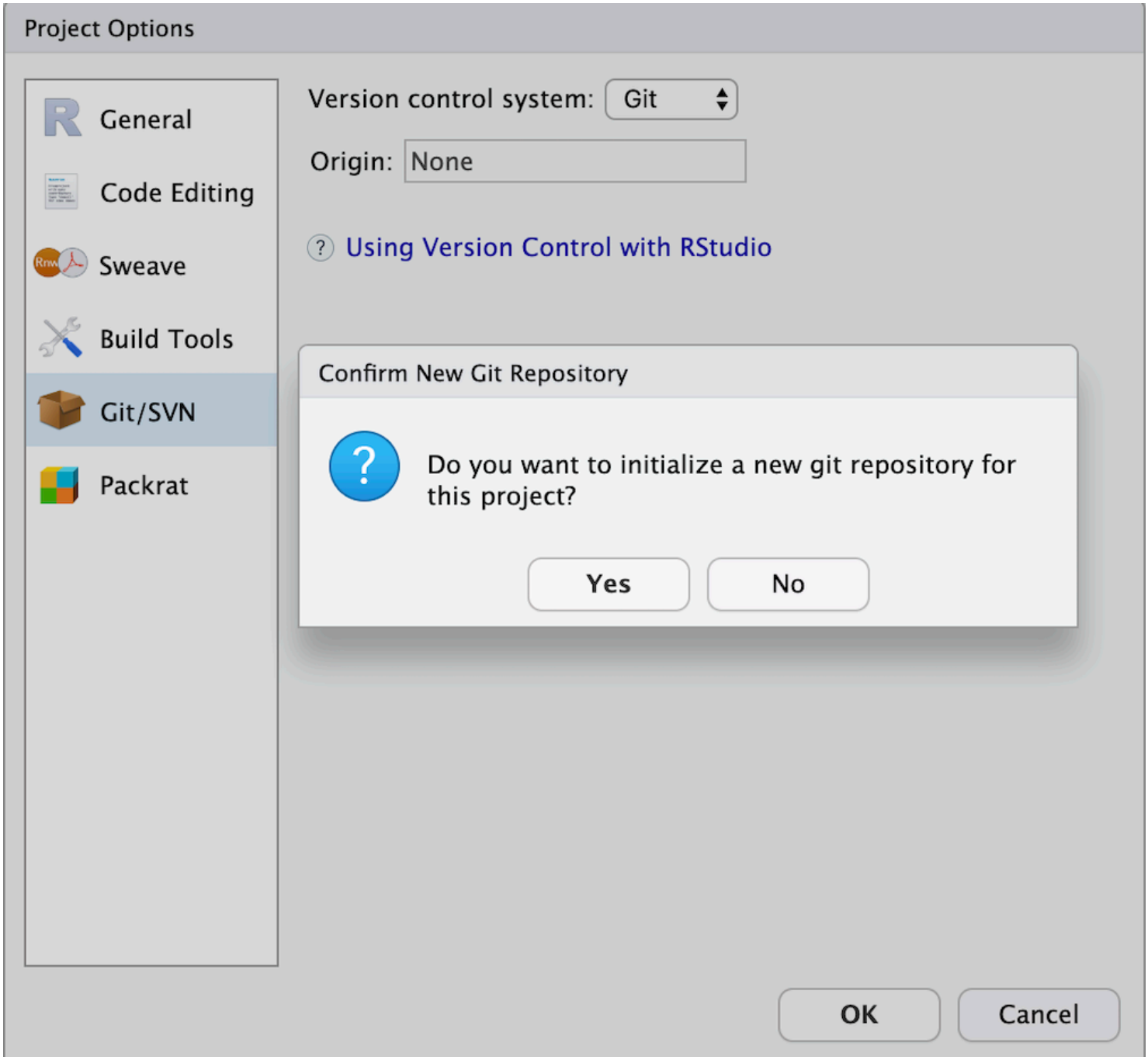
If you'd like to install Git on your local machine, you can do so following these two links:

1. Download and install Git.
2. Create a Github account.

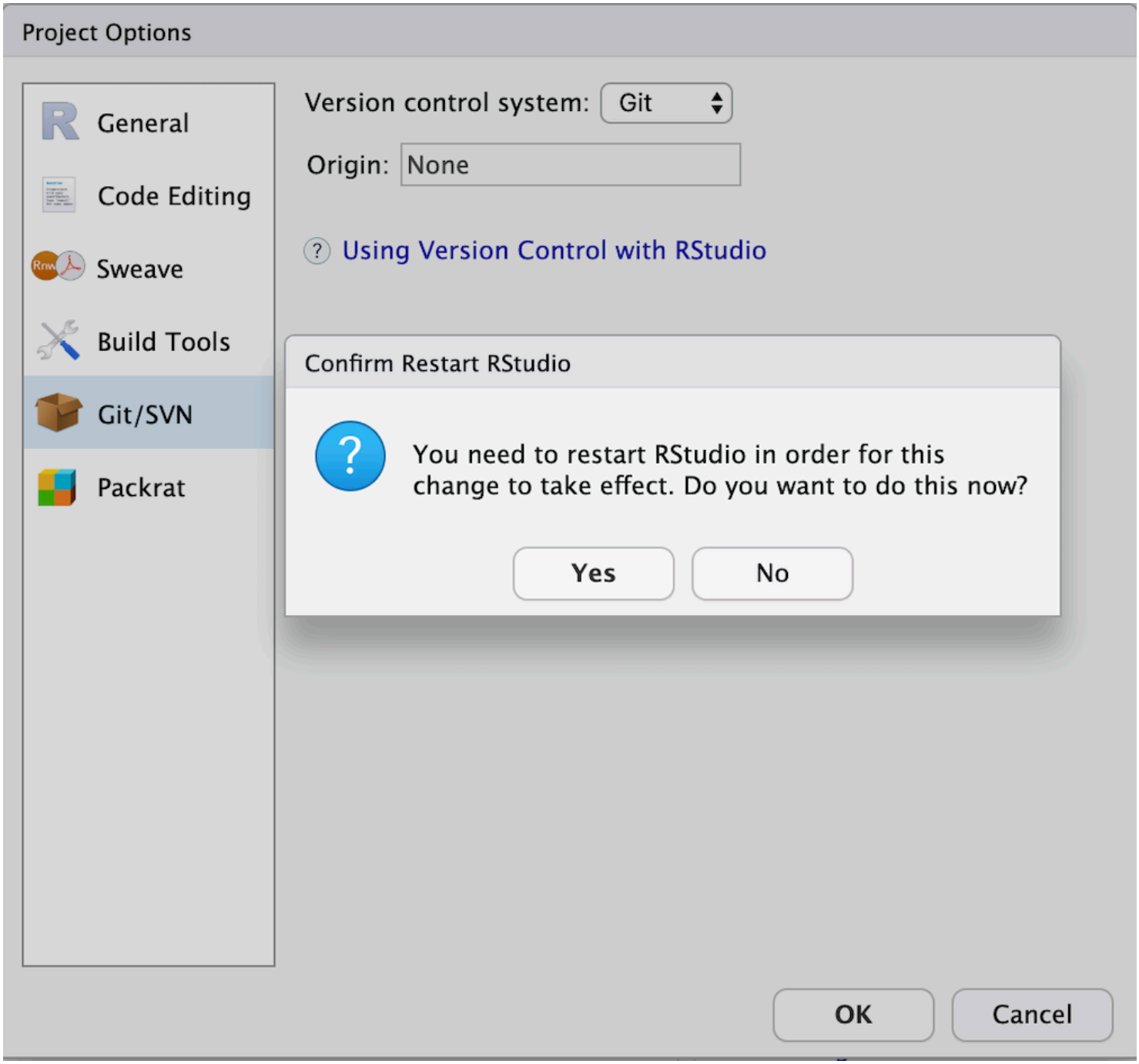
In RStudio.Cloud, we want to add version control to this project from *Tools* > *Version Control* > *Project Setup*



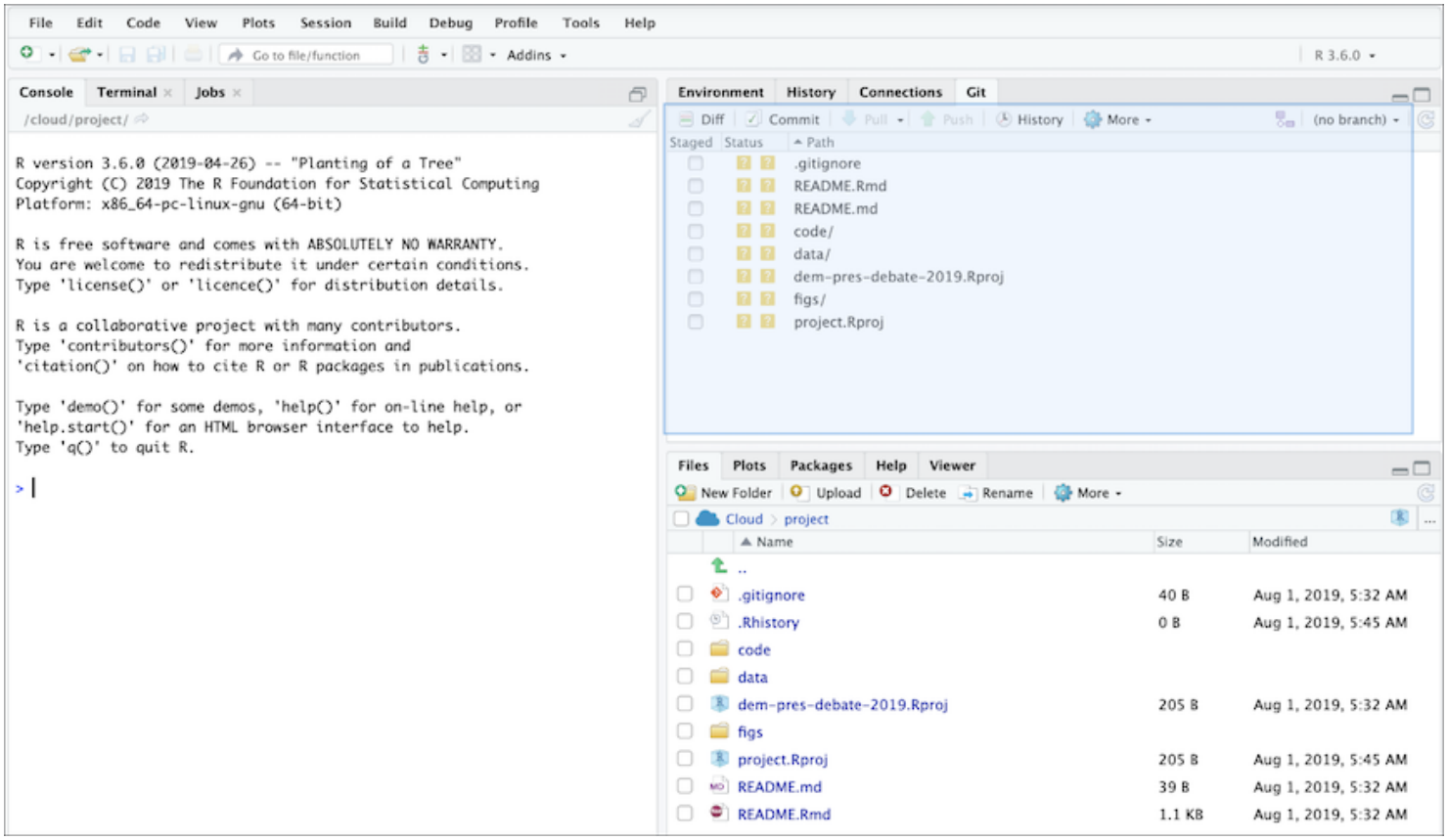
From here, you will see the *Git/SVN* option on the sidebar, where you will select *Git* from the dropdown list next to *Version control system*. After this, RStudio.Cloud will ask if you want to *initialize a new git repo*, which you do.



Then you will be asked if we are ok to restart RStudio.Cloud (and we do).



After restarting the RStudio.Cloud IDE, we should see the **Git** tab in one of the panes.



Configuring Git

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`

We'll set our Git `user.name` and `user.email` with `git config --global` so these are configured for all projects.

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

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

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

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

```
user.name=Martin Frigaard
user.email=mjfrigaard@gmail.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

On our local machine, the `user.name` and `user.email` are in the `.gitconfig` file.

We can view this using:

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

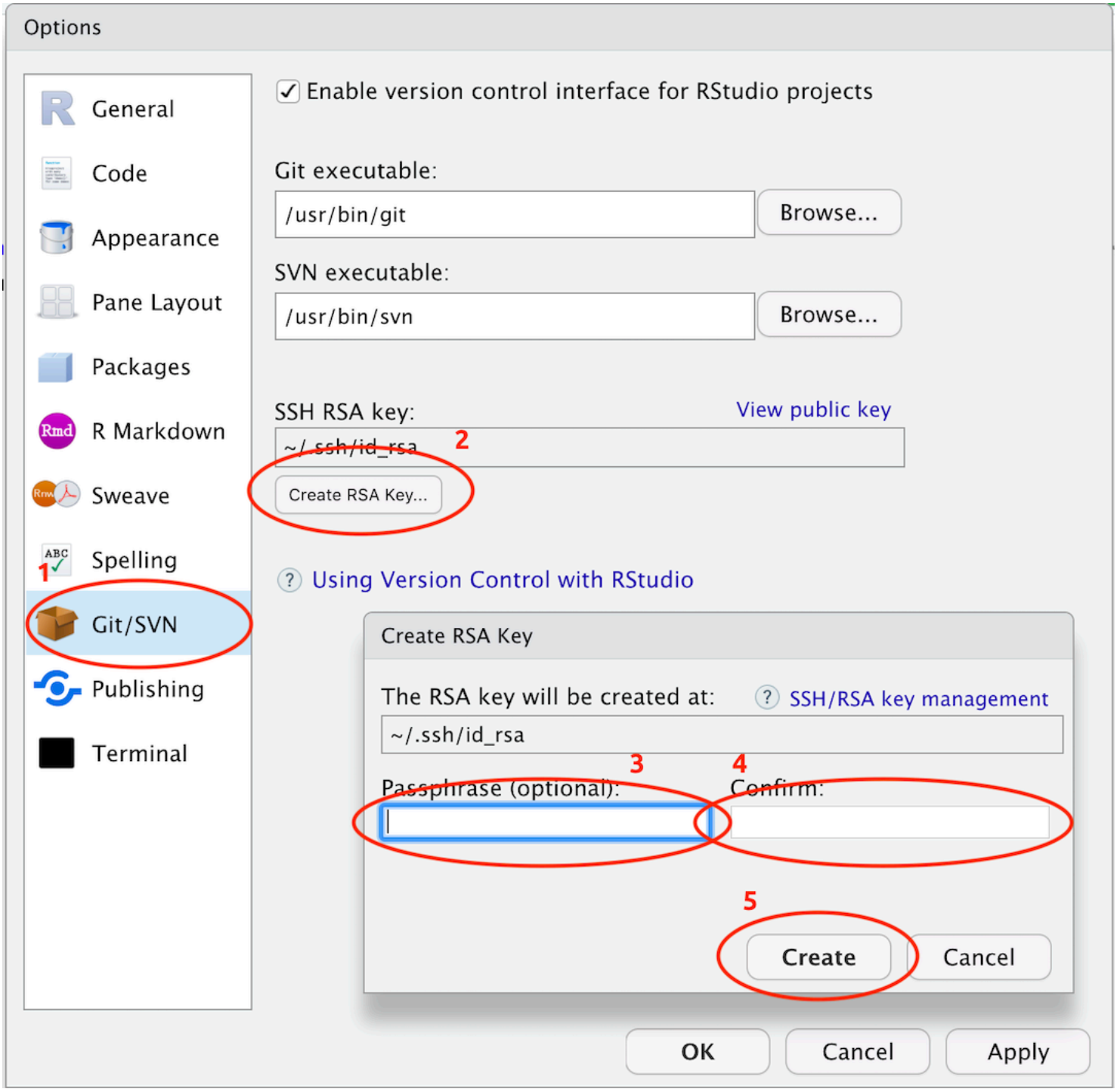
Synchronizing RStudio and Git/Github

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

The first step is setting up the RSA Key and passphrase.

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:

```
Generating public/private rsa key pair.
Your identification has been saved in /home/rstudio-user/.ssh/id_rsa.
Your public key has been saved in /home/rstudio-user/.ssh/id_rsa.pub.
The key fingerprint is:
[redacted]
The key's randomart image is:
+---[RSA 2048]----+
|      .=0=..o0Oo  |
|     ..=. o ++oo  |
|      o .o oo.ooo.|
|       .  Xoo.Eo..|
|     [redacted]   |
|                  |
|                  |
+----[SHA256]-----+
```

Or like this on your local machine.

```
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]-----+
```

Great! We need to go back to Terminal and store this `SSH` from RStudio.

Adding a key SSH in Terminal

In the Terminal, we enter the following commands.

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

The response tells us we're an `Agent`.

```
Agent pid 007
```

Now we 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 we generated, and
- `~/.ssh/id_rsa` is the location of the *SSH RSA*.

```
$ ssh-add -k /home/rstudio-user/.ssh/id_rsa
```

Enter the passphrase, and then git should tell us the identity has been added.

```
Enter passphrase for /home/rstudio-user/.ssh/id_rsa:
Identity added: /home/rstudio-user/.ssh/id_rsa (/home/rstudio-user/.ssh/id_rsa)
```

Create the `.ssh/config` file

Most operating systems require a `config` file. We can do this using the Terminal commands above.

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

```
$ cd /home/rstudio-user/.ssh
```

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

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

We 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, we 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'll extend our Github skills by moving the contents of a local folder to Github.

More on Git and Github and data organization

Fortunately, many articles have come out in the last few years with excellent, practical advice on organizing data analysis projects. 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:

- the importance of using version control
- sharing data with collaborators
- how to name your files

Part 5: Working in RStudio.Cloud

In this chapter, we're going to learn about R packages, navigating the RStudio panes, some additional benefits of working in Rmarkdown and plain text, and how to make and monitor changes with Github.

Navigating the panes in our workbench

The next few sections will walk through a few of RStudio.Cloud's panes. To recap:

1) We uploaded files into the `project` folder, which now has the following structure:

```
project/
    ├── CHANGELOG.txt
    ├── README.Rmd
    ├── README.md
    ├── code/
    ├── data/
    ├── dem-pres-debate-2019.Rproj
    ├── figs/
    └── project.Rproj
```

2) There are three folders in this project: `code/`, `data/`, and `figs/`.

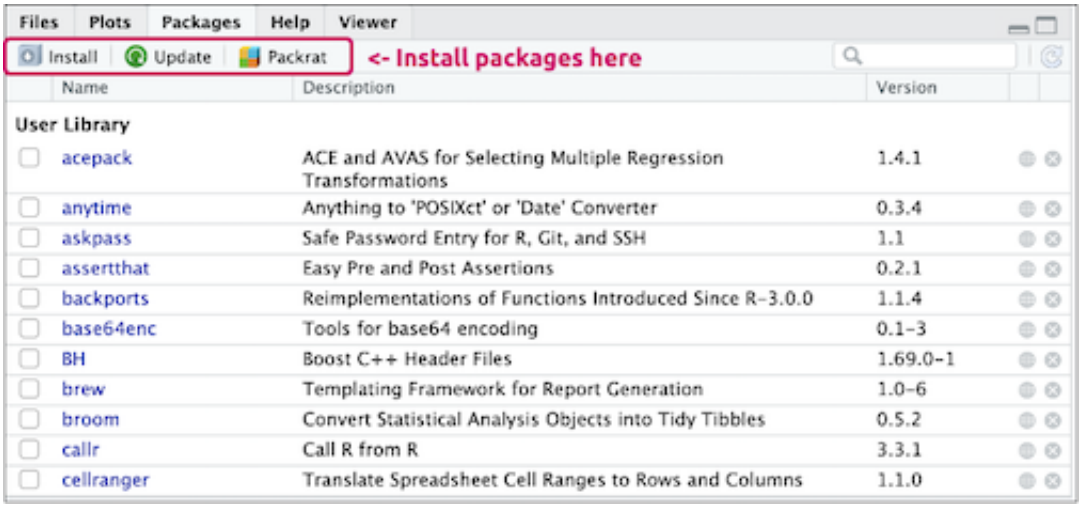
3) There are three types of plain text files in the parent folder: `.md`, `.Rmd` and `.txt`, and two project files: `dem-pres-debate-2019.Rproj` and `project. Rproj`.

We'll go over these folders and files in more depth in the following sections.

Packages

Packages are a vital part of the R ecosystem. R packages are collections of functions and objects collected together with a specific purpose. When we started our RStudio.Cloud session, a few packages get loaded automatically. R users typically refer to these default packages and functions as "base R." Base R packages cover a wide range of statistical procedures and visualizations. Unfortunately, base R can also be challenging to learn because of its inconsistent syntax and style conventions.

A list of the packages and their descriptions are available in the **Packages** pane. We've installed the packages we will use with the `00.1-inst-packages.R` file. If we wanted to, we could also install more packages using *Install* icon, and then enter its name. We don't recommend this, though, because you'll want a record of the packages you used in the project.



You can click on the names of each package to learn more about them and load them into the RStudio session. One of the great things about R is the many user-created packages that greatly expand the number of functions. At the time of this writing, R users have contributed 14638 packages available for us to download and use.

The `tidyverse`

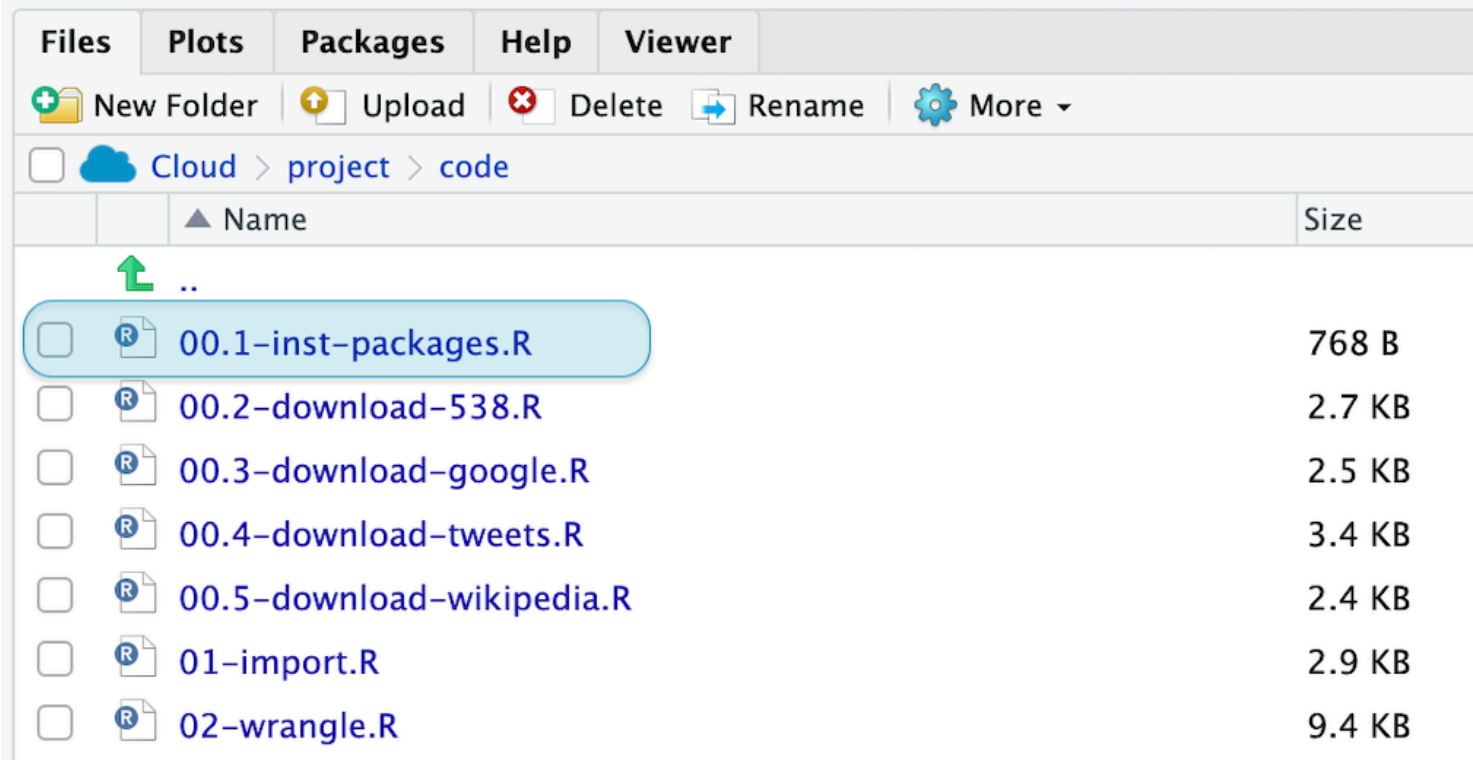
R is open-source software, so users can write packages to expand and enhance its functionality. The tidyverse is a collection of R packages from RStudio for doing data science. All `tidyverse` packages share a few similar underlying principles that allow them to work well together.

Unlike base R, the `tidyverse` also a consistent grammar and syntax, which makes it easier to read and write. You can learn more about this syntax in the R for Data Science text or on the tidyverse webpage.

So far, the code we've run comes from base R. Going forward; we're going to use various packages from the `tidyverse`.

Files

The **Files** pane displays the files and folders in this project. The file path is visible in the top portion of the pane, beneath the options for *New Folder*, *Upload*, *Delete*, *Rename*, and *More*. An image of this folder's contents is below:



The script to install the packages for this project (`00.1-inst-packages.R`) is in the `code` folder. Navigate to this folder either by clicking on the file path (`Cloud/project/code`).

We can see `00.1-inst-packages.R` and other scripts are in the **Files** pane. We now know that because the `.R` files are plain text files, so they have code for the computer to execute, and comments for a human to tread.

In R, we can create comments with a preceding # on any line.

```
# this is a comment
```

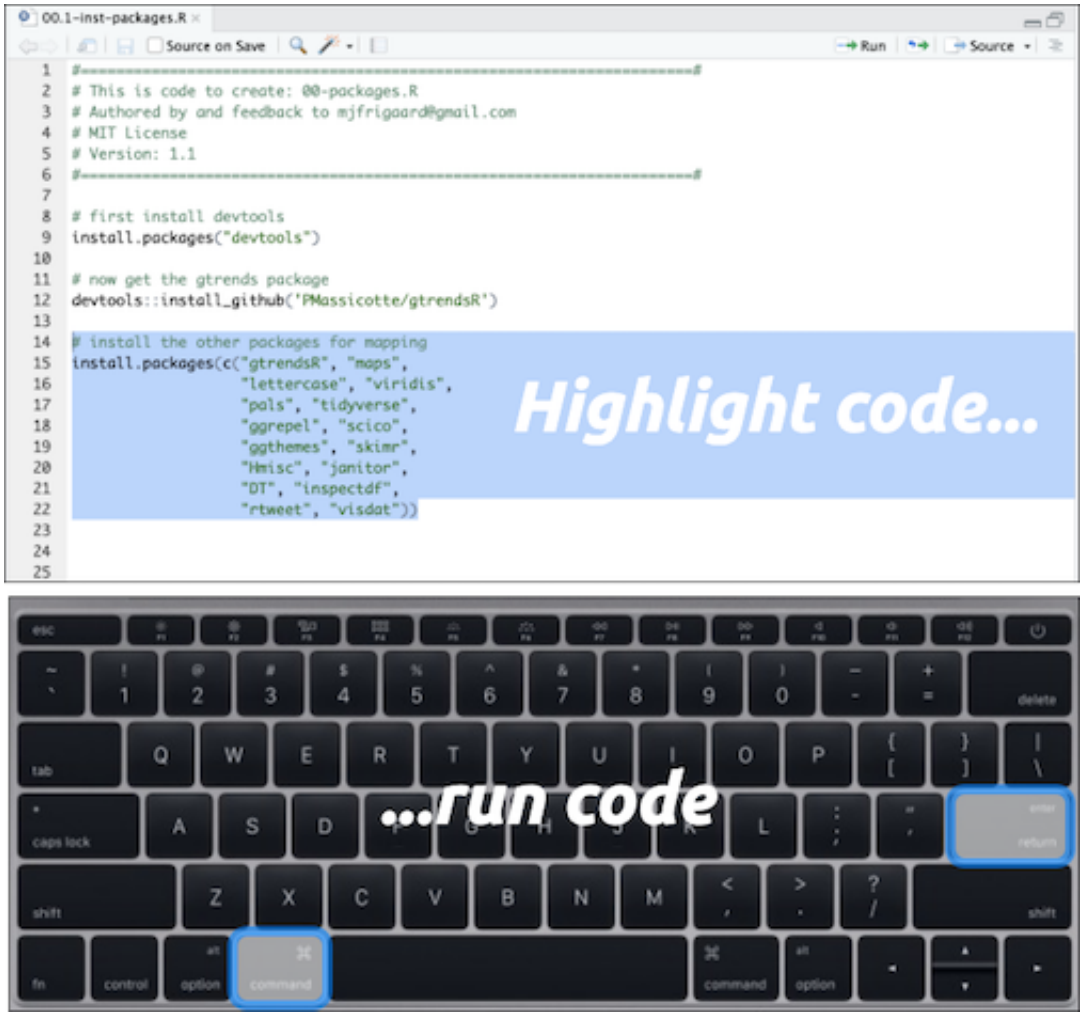
When we click on the `00.1-inst-packages.R` file, we can see it open in the **Source** pane. RStudio.Cloud is giving us a few hints about the packages referenced in the .R file. We're being told we need to install a package before moving forward:

> ***Package devtools required but is not installed.***

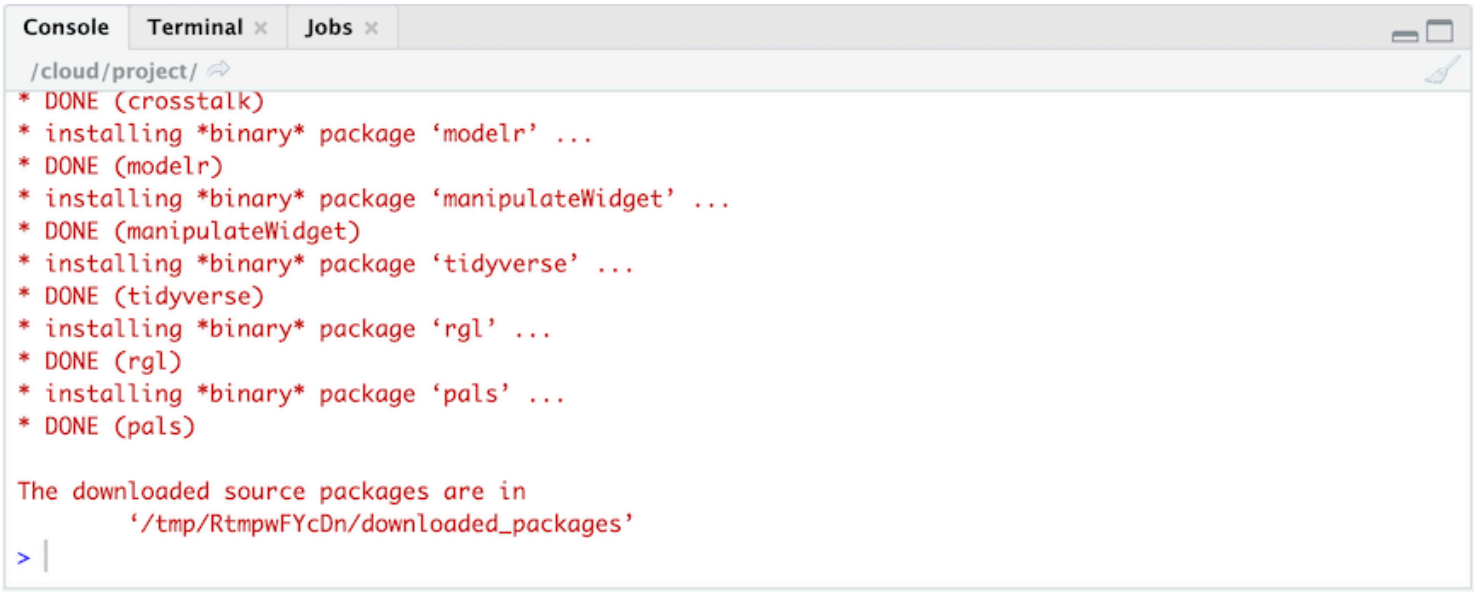
RStudio gives us a choice to ***Install*** or ***Don't Show Again***, and we'll click on the ***Install*** option. Packages vary in the length they take to install, but we'll wait patiently for `devtools` package finish downloading. After the install has finished, you should see the > prompt in the **Console** pane.

Source

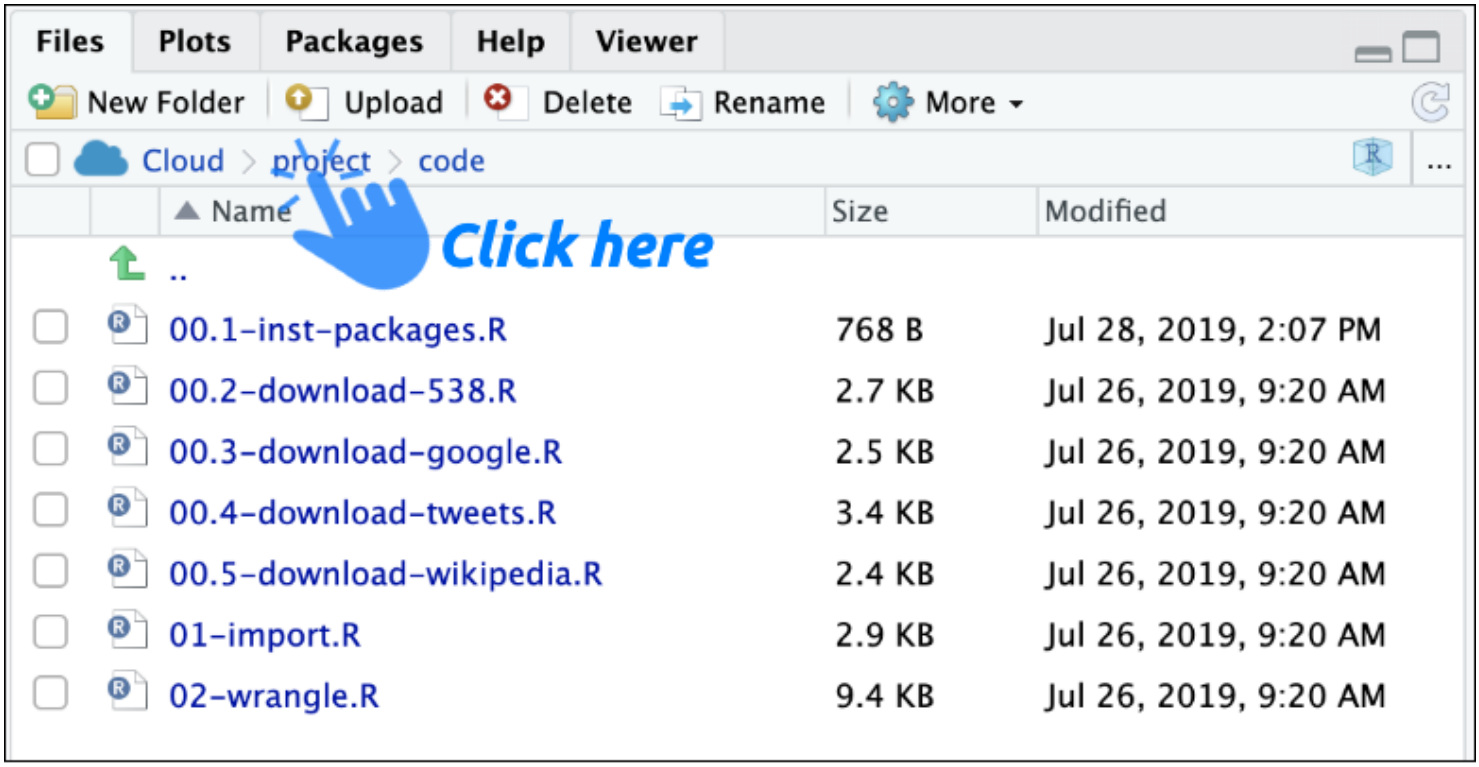
When working in RStudio, sometimes we'll write some code, and want to run that code and see if it works. To do this, we can highlight the rest of the script with the mouse cursor (lines 14–22), then hold down `ctrl` or `cmd` and hit `enter` or `return`.



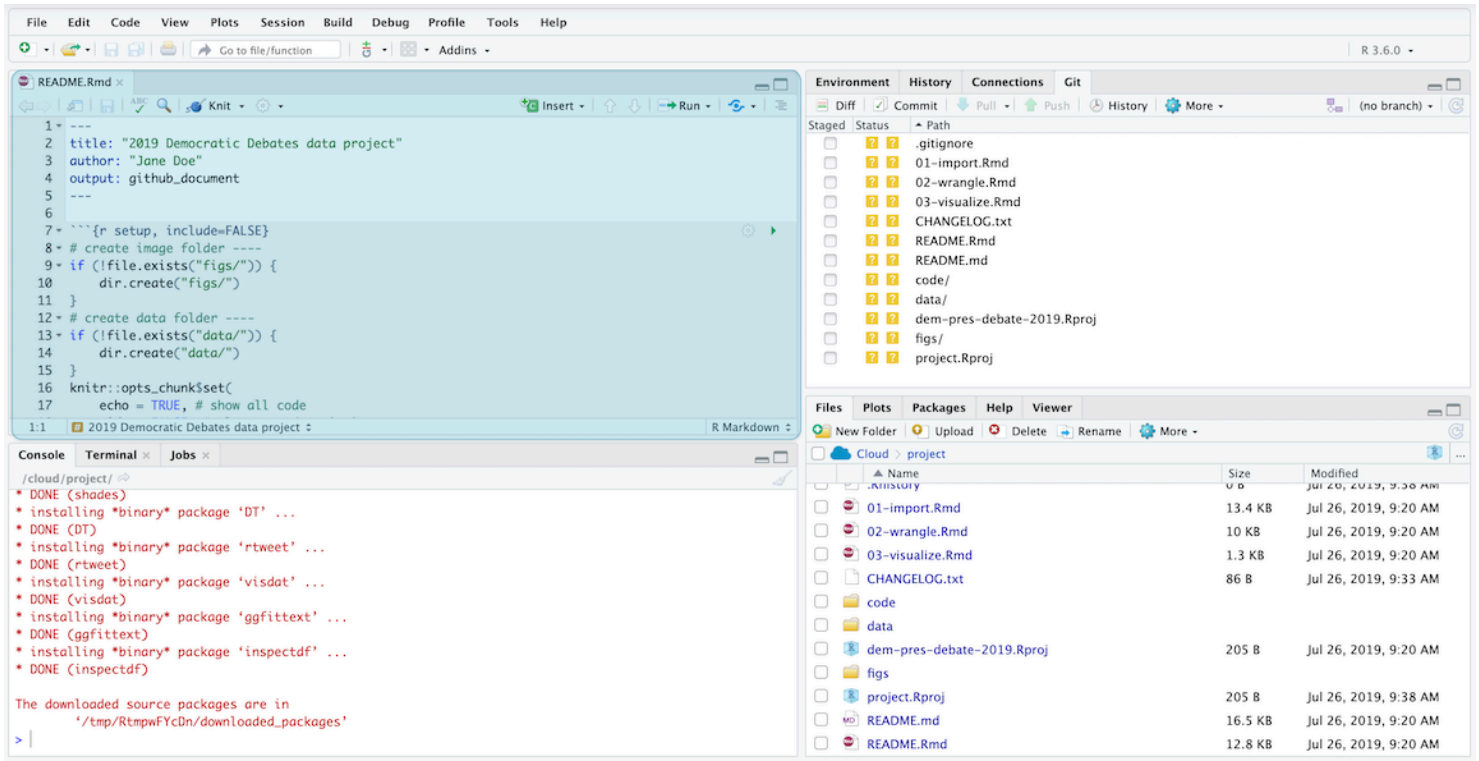
After the installation has completed for all the packages, we'll see the following in the **Console** pane.



Back in the **Files** pane, we can navigate back to the `project/` folder by clicking on the path above the files.



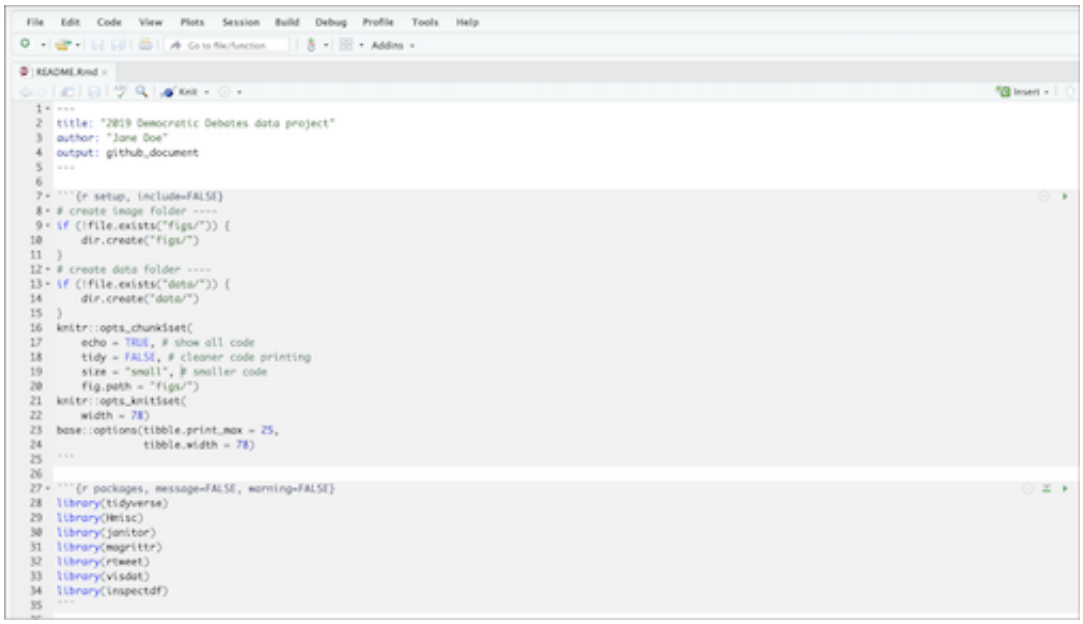
We can see the `README.Rmd` file in the **Files** pane in the lower right corner. We'll click on it, so the file opens in the **Source** pane (see image below).



We've highlighted the pane of RStudio that's displaying the `README.Rmd` file. As you can see, this is a relatively small area in the IDE. Working in a tiny corner of the screen can be hard, but fortunately, RStudio gives us the ability to expand any pane into a fullscreen view. In this case, we'll zoom in the `README.Rmd` file.

If we want to focus on the **Source** pane, and can zoom in using `shift+control+1`.

As soon as you click `1`, your screen should expand to look like the image below:



Since the **Source** pane is where we'll write most of our code, it's also the pane we are spending most of our time. We've seen that's where our files open, so being able to focus on the area quickly is helpful.

Console

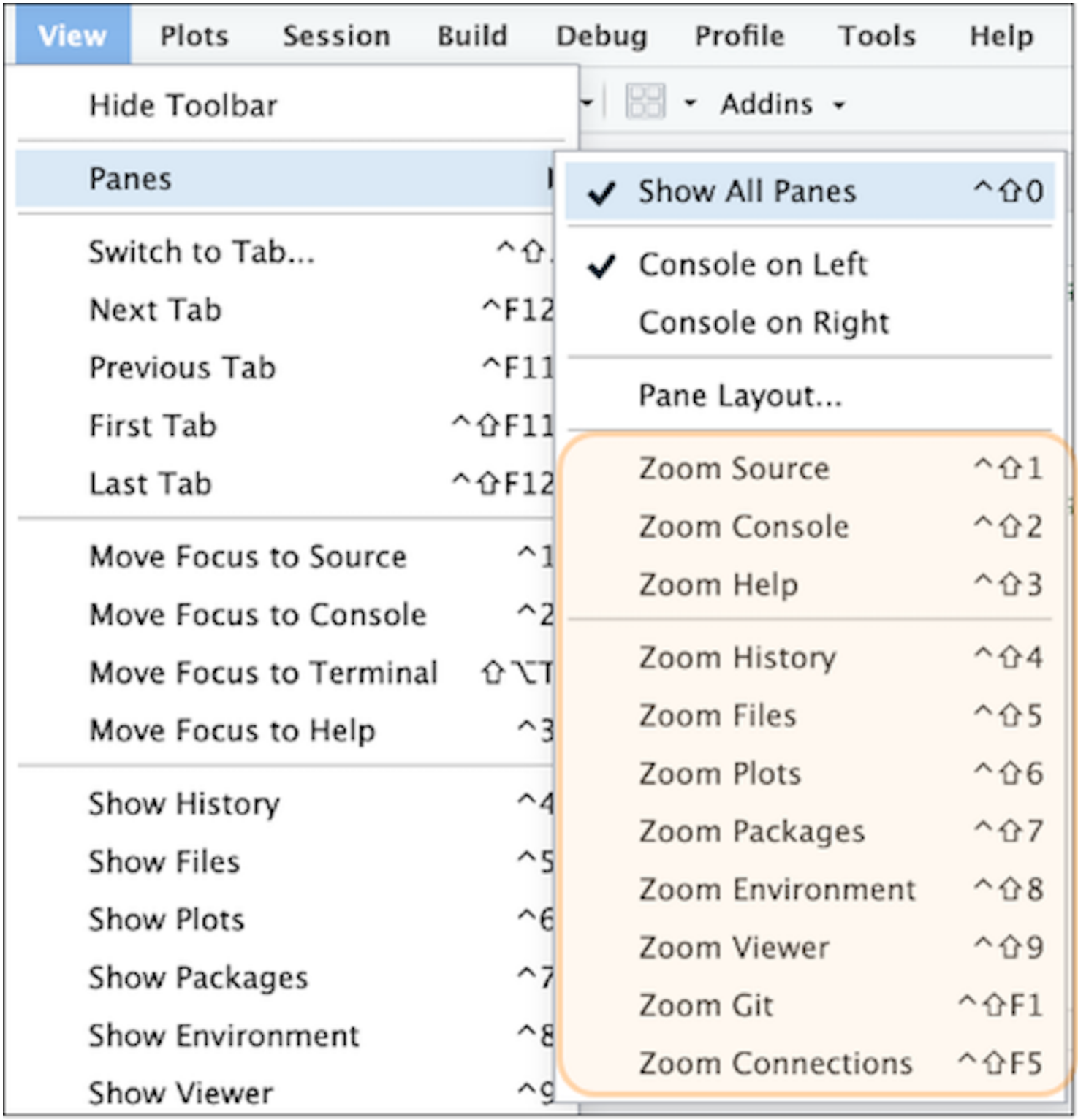
The **Console** is probably the second most used area in RStudio (it displays most of the output), so we also should know how to zoom in on this pane. To focus on this pane, we'll use `shift+control+2`.

What if we accidentally zoom into the wrong pane? We can easily resize the IDE to its original position by holding down the `shift+control` buttons, then clicking either number again. Combining those keys should return the IDE to its default arrangement.

Working in one pane

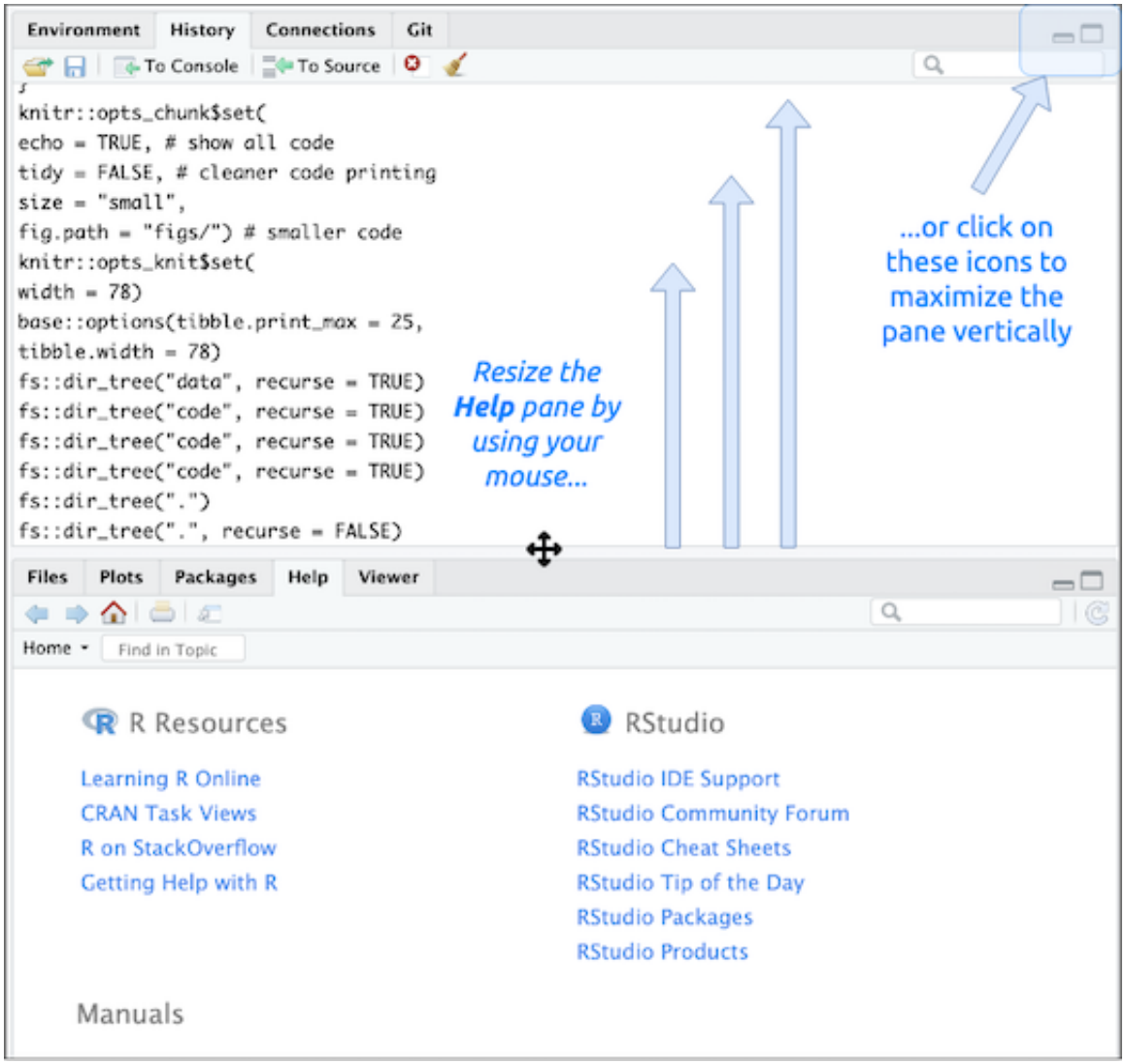
We can move through every pane in RStudio by holding down `shift+control` and clicking on numbers `1-9`. Go ahead and do that now.

If you ever forget which number corresponds to which pane, you can always find them under *View* > *Panes* (see image below)



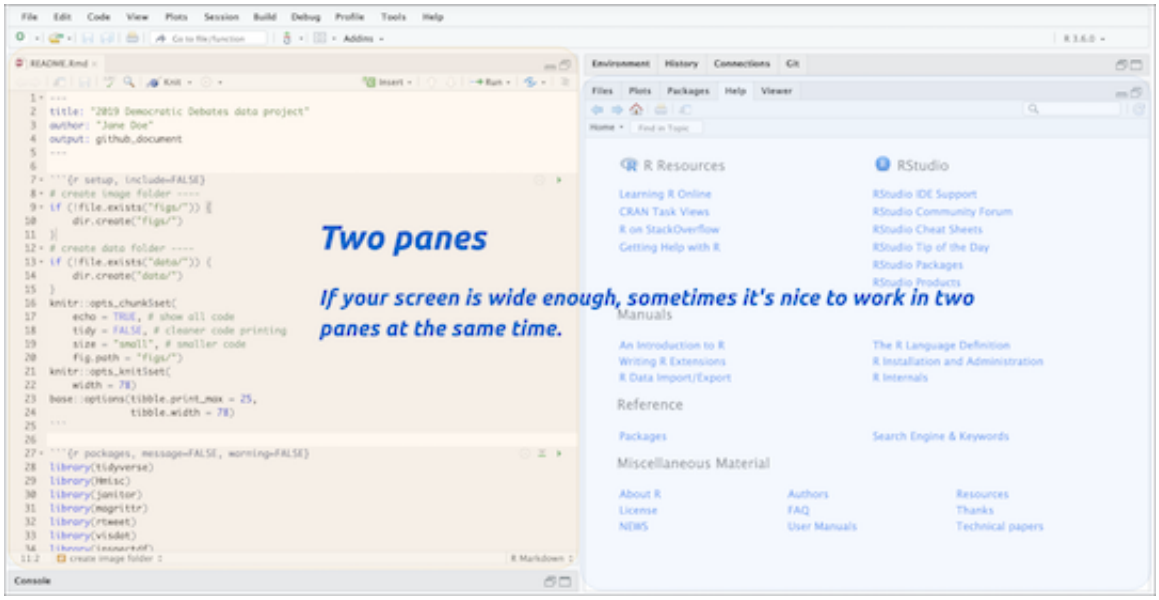
Help

Inevitably, we'll write something that doesn't work. When things aren't working (try to remain calm), you're going to want a place to start looking for solutions. The **Help** pane is accessible in the lower right corner of the IDE.



Working in two panes

Sometimes we'll want to do our work in more than one pane at a time. For example, what if we're working in the **Source** pane, but have a question about a function? We can get answers to a lot of problems using RStudio's internal **Help** pane. After resizing the **Help** pane, we should see the following layout in our IDE.



Recap on RStudio panes

As we pointed out earlier, RStudio is like a workbench built around different panes. Each pane serves a specific purpose and being able to move between them allow us to work quickly and efficiently.

Keyboard shortcuts are time-savers. Not having to switch from keyboard to mouse over and over again keeps us focused on the task at hand.

Where do we write code?

You might be wondering why we have provided you with both `.R` scripts and `.Rmd` files. Well, we want to show you a few options for documenting your work in RStudio. The two sections below outline two standard options (but these are by no means the only way to work with these tools!)

Option 1) everything goes in scripts

`rmarkdown` is a relatively new package, so early `R` users didn't have an option to put everything in `.Rmd` files (we fit into this group). We have found this approach is excellent when you're reasonably confident about the project. If you're familiar with the data files, wrangling techniques, visualizations, and products, you can quickly outline the scripts and predetermine their names and contents.

For example, the folder tree of our code folder tells us what we should expect from this project.

```
code
    ├── 00.1-inst-packages.R
    ├── 00.2-download-538.R
    ├── 00.3-download-google.R
    ├── 00.4-download-tweets.R
    ├── 00.5-download-wikipedia.R
    ├── 01-import.R
```

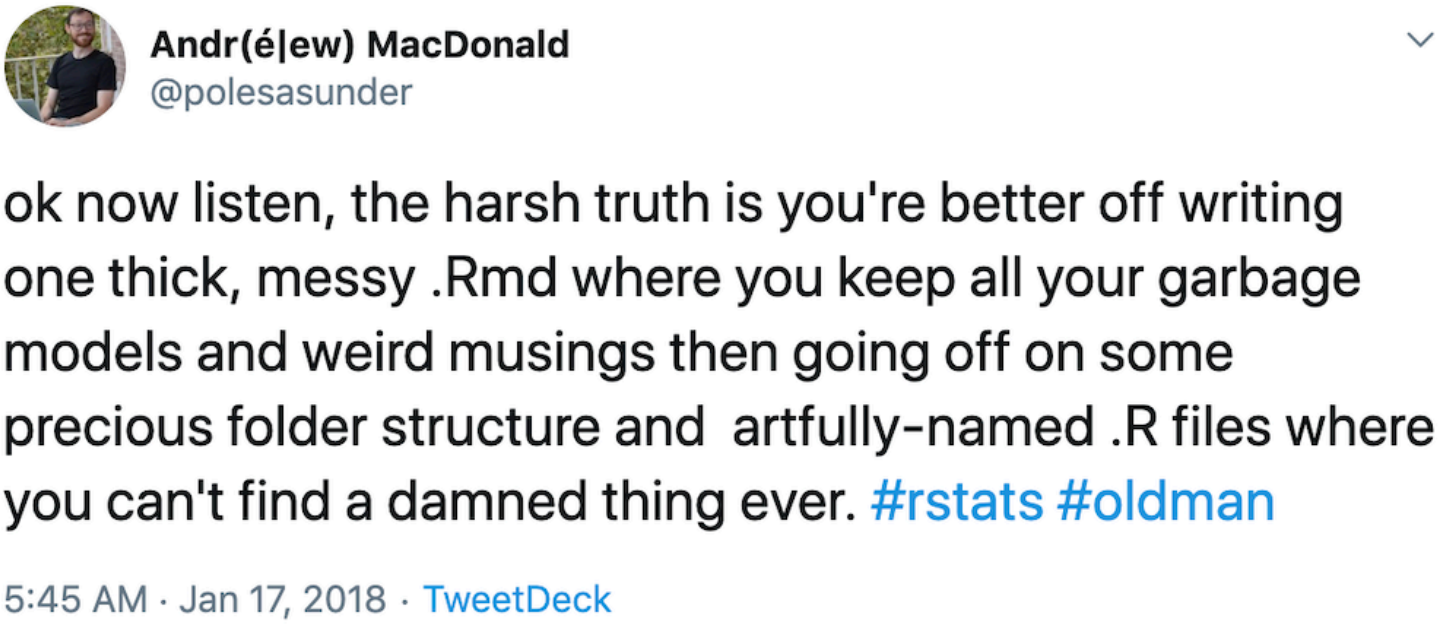
```
└── 02-wrangle.R
```

We can see there are seven script files, each one with a numerical prefix. These prefixes tell us the order to run the code files, too. The file names also tell us what each script does (`download`, `import`, `wrangle`, etc.).

The layout above is an example of how to organize a set of `.R` scripts that follow the guidelines mentioned in previous chapters. These naming conventions make it easier for newcomers to the project to get oriented to its contents.

Option 2) everything goes in the Rmarkdown

The second option is well illustrated in the tweet below by Andrew MacDonald:



https://twitter.com/polesasunder/status/953624238266646529?s=20

In our experience, this accurately captures the reality of data projects (especially in the beginning stages). Throwing everything into the Rmarkdown file gives us a lot more flexibility by allowing us to add multiple types of content.

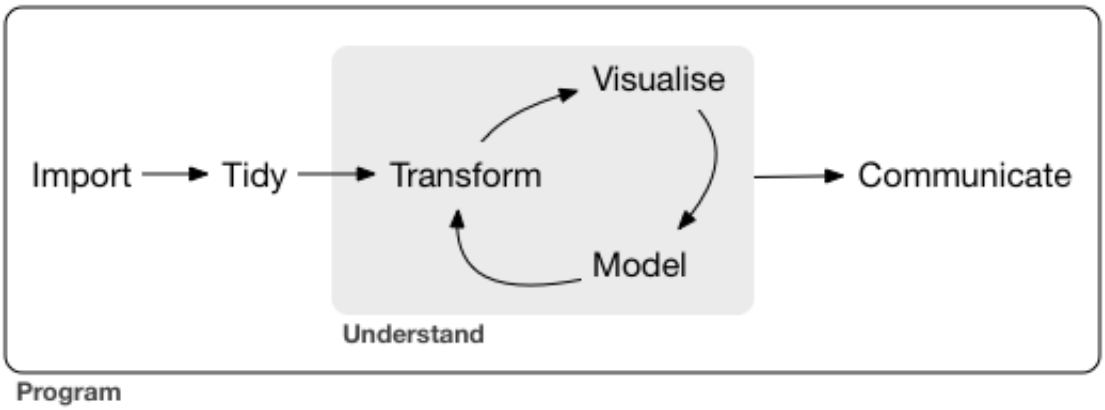
All that being said, it's always good practice to come back and revise the `README.Rmd` document, so it's contents outline all steps in the analysis thoroughly. We consider documentation to be a form of communication with our future selves, and this typically involves clearly describing each step. During the revision, we might also break down each of the chunks into individual scripts. We can also add more details to the text portions, with links to external content, etc.

We've found each project usually starts at a bit of a sprint, so we try to capture as much code and content in `.Rmd` files early. We can focus on trimming it down later, but the flexibility of markdown combined with functional code is better than limiting ourselves to code and comments `.R` script files.

Rmarkdown

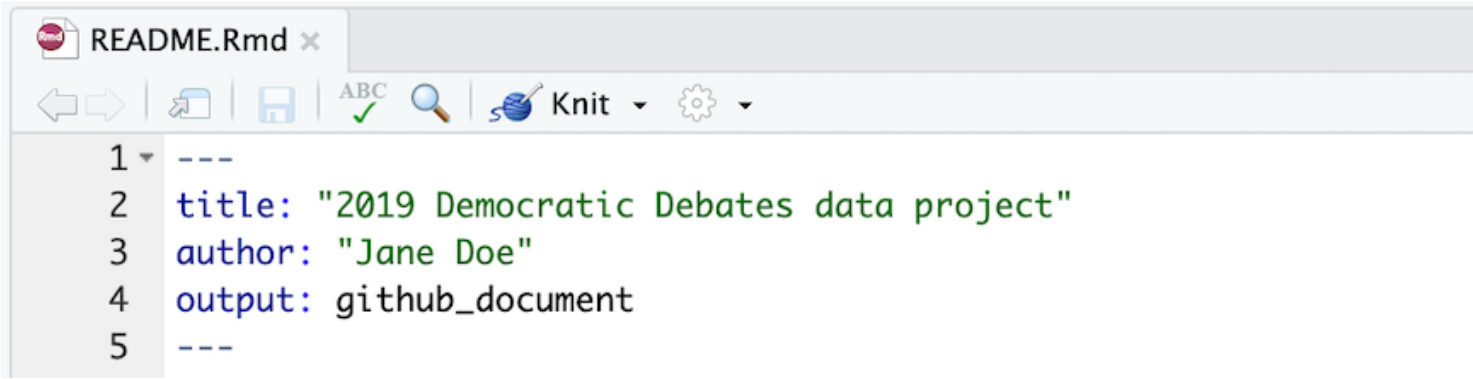
We're going to move forward assuming we're documenting everything in a `README.Rmd` file. In the next few sections, we'll add various components to the `README.Rmd` we've placed in the project folder.

If you're ever wondering how to outline your `README.Rmd` file, the figure from R for Data Science isn't a wrong place to start.



.Rmd step 1: the YAML header

At the top of the `README.Rmd` document, the first thing we see is what's called the "`YAML` header", and it's going to tell RStudio.Cloud the file's `title`, the `author`, and what the `output` file will be.



The `YAML` header always goes at the top of the `README.Rmd` file, between two sets of three dashes:

```
---
title: "2019 Democratic Debates data project"
author: "Jane Doe"
output: github_document
---
```

People typically use `YAML` in configuration files, which makes it perfect for setting some default options in our `README.Rmd` document. Read more about YAML headers in the RMarkdown book and on the YAML website. `YAML` actually stands for "YAML Ain't Markup Language."

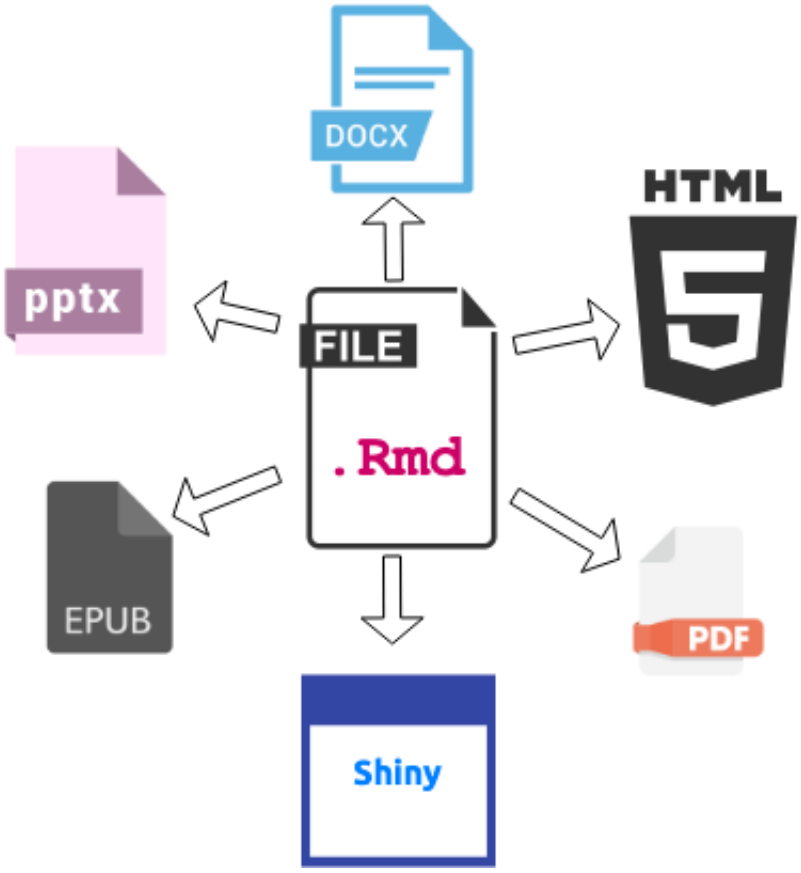
We can change the `output` argument to `html_document`, `word_document`, or `pdf_document` and create a different file from the plain text we are going to be working in. For now, we are going to focus on the `github_document` output.

.Rmd step 2: Knit output options

Another benefit of working in the `README.Rmd` document is that when we combine it with the powerful `knitr` package, we drastically extend the kind of files we can produce from our analysis. `Knitr` follows a principle of literate programming put forth by Donald E. Knuth.

> *"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do."*
> *- Donald E. Knuth, Literate Programming, 1984*

The **Knit** button will render the plain text `.Rmd` document into a variety of different output options.



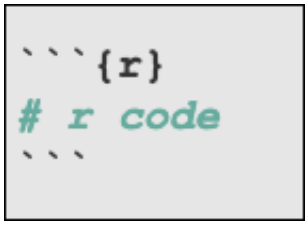
The `.Rmd` files also give us the ability to document our intentions, write and execute code, and interpret and explain the results. After we've outlined (and revised) our thought process, we can go about organizing the code in more efficient ways to carry out our intentions.

Functional code chunks

We like to think of the `.Rmd` document like a stack of two sheets of paper, and each piece representing a different file type (`.R` and `.md`). When you lay the markdown file on top of the .R script, you get an Rmarkdown file.

R code chunk

This combination of files gives us a syntax for formatting our text (stuff like *italic*, **bold**, `code`, etc.), and a functional code script we can get access to by inserting code chunks. For example, in the `README.Rmd` file we can type directly onto the paper using the markdown syntax. But if we want to run some R code, imagine tearing a little hole in the markdown paper, and revealing an R script underneath.



R code chunk

These code chunks allow us to run R code in-between the markdown text.

R Code chunk options & labels

Code chunks come with a long list of options (feel free to experiment with all possible combinations found here and here). The most common are `echo`, `eval`, and `include`.

- `echo` = *show the code in the output?*
- `eval` = *run the code chunk?*
- `include` = *put the results from the code in the output?*

We will show a few examples of these with labels below:

```
```{r run-show-nothing, include=FALSE}
# runs and shows nothing
```
```

```
```{r run-show-code, eval=TRUE, echo=TRUE}
# runs code and shows results
```
```

```
```{r run-hide-code, eval=TRUE, echo=FALSE}
# runs code, shows results, but hides code
```
```

```
```{r no-run-show-code, eval=FALSE, echo=TRUE}
# shows code, but doesn't run code
```
```

The code chunks above are labeled, and we recommend *labeling all code chunks*. We've found it forces us to think through the analysis as a series of operations, and reduces random calculations spread throughout the document.

The setup code chunk

After the `YAML` header, our `setup` chunk tells us what we're going to be doing with the code, text, figures, and output in this `README.Rmd` file.

Our `setup` chunk does the following:

Project folders: We check to see if the project has image or data folders, and if not, it creates them.

```
# create image folder ----
if (!file.exists("figs/")) {
    dir.create("figs/")
}
# create data folder ----
if (!file.exists("data/")) {
    dir.create("data/")
}
```

Chunk options: We've included `echo=TRUE`, which means the code will print in the output file. The `tidy=FALSE` makes sure the code doesn't get reformatted when the document gets rendered. We set both options to their default values, but we've included them below so that you can see more examples.

The `size = "small"` and `fig.path = "figs/"` are used to change the size of the printed code, and the location of any output visualizations.

```
knitr::opts_chunk$set(
    echo = TRUE, # show all code
    tidy = FALSE, # cleaner code printing
    size = "small", # smaller code
    fig.path = "figs/") # where the figures will end up
```

Knit options: The `knitr::opts_knit$set()` function gives us the ability to stipulate options for what happens when we render the document (knitted and rendered are somewhat synonymous).

```
knitr::opts_knit$set(
    width = 78)
```

Base options: the final setting tells RStudio.Cloud how we want the tables to print (`25` rows, and `78` columns).

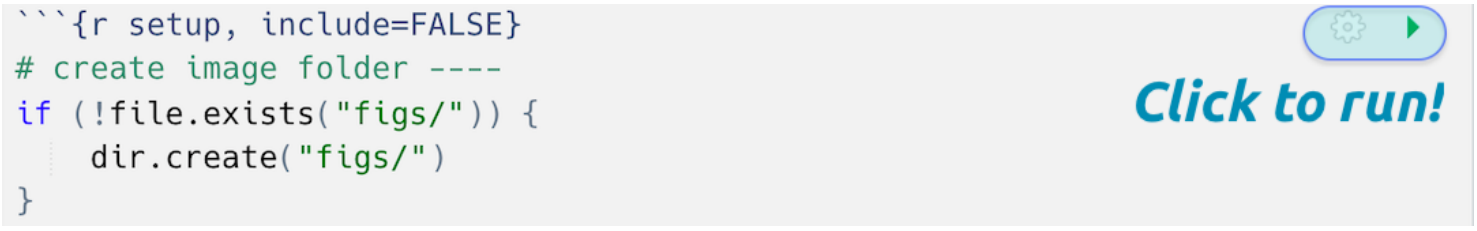
```
base::options(tibble.print_max = 25,
              tibble.width = 78)
```

Why `78`? The standard code file is 80 columns across, which is the length of a punch card (read more here). We go two columns less than 80 (to give a little wiggle room).

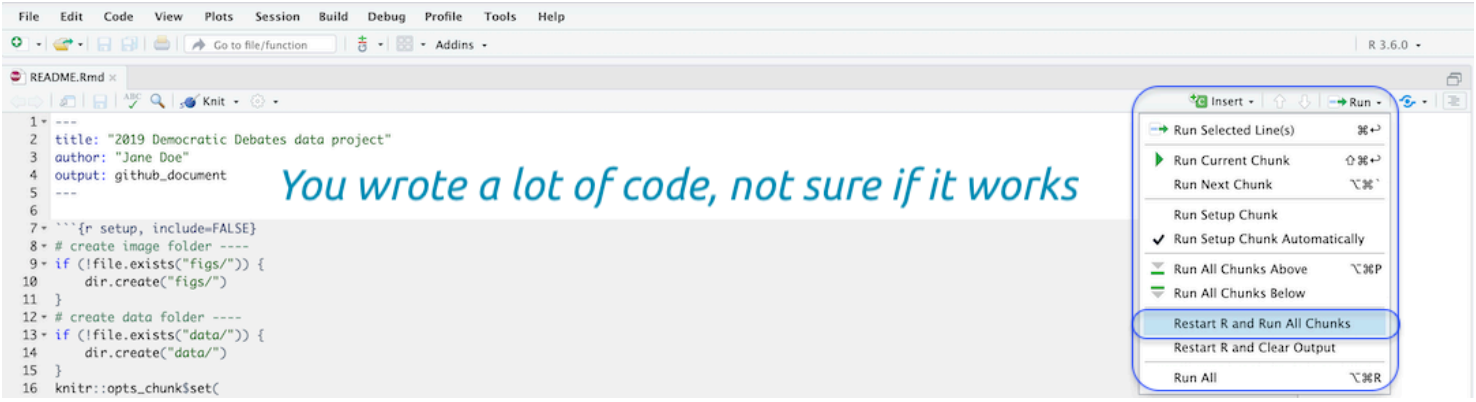
Running code chunks

To run the code in your document, we have a few options. First, we can use the same keyboard shortcuts we use for executing code in the `.R` scripts (`ctrl+enter` or `cmd+return`).

The second option is the small green play icon on the far right-hand side of the code chunk. Running code one chunk at a time is a great way to test what the output of a graph looks like, or to see the data table from a join.



The third option is we run a series of chunks (or the entire document) using the drop-downs at the top of the `.Rmd` file. We use this option if we've added a few chunks, and we want to make sure everything works together.



Don't worry if all of this is confusing! We will be creating our very own `README.Rmd` for this project, so you'll get some practice!

Start with a README.Rmd

The `README.Rmd` file we've provided for this project is in the **Source** pane, and we can start by looking at the contents of the file (look at the `YAML` header, `setup` chunk, and `packages`).

As we've stated before, we won't be going through the code collects and downloads the data for this project. We'll provide links to resources where you can get all that information, bu we're going to focus on the *workflow for using data to create tables, images, and products*.

Hit the enter/return key until we a few lines of padding beneath the `packages` code chunk. Our cursor should be around line 37.

We will begin this document with the following text under our first line header (#).

```
# Motivation

We read an interesting article on `fivethirtyeight` about the democratic debates in June of 2019. In p
articular, the image below displays how voters had changed their minds after watching the candidates.
```

As the text above states, we saw something cool, and we're going to see if we can recreate or verify some of the information. The source of this inspiration comes from this fivethirtyeight article.

> ***Including an image in the .Rmd file:*** *If you want to include an image in your `README.Rmd` file, read about the `knitr::include_graphics()` function by entering `??include_graphics` in the* ***Console*** *pane.*

Data sources

Now that we have an idea why we're here, we want to start exploring the data. We're going to document all of our steps in the `README.Rmd` file.

We will start by importing all of the data for this project. We've put the raw data files in the `data/raw/` folder. **Always leave raw data in its own folder, and never alter the raw data files (read more here).**

In our `README.Rmd` file, we'll create a new level-two header (##) called 'Data files' and a code chunk labeled "`locate-data`" (it should look like the image below).

````
41 ## Data files
42
43 Below are the data files sorted by size:
44
45 ```{r locate-data}
46
47 ```
48
````

File management with the `fs` package

To locate the files in `data/raw/` folder, we like to use the `fs` package. `fs` stands for 'file system', and this package gives us the ability to navigate our project folders and files.

Check out the package website here for a full description and examples. `fs` is loaded as part of the `tidyverse`.

Start with a folder tree of the data folder.

```
# where are the data?
fs::dir_tree("data")
# data
# ├── processed
# └── raw
#     ├── 538
#     |   └── 2019-07-06-Cand538Fav.csv
#     ├── google-trends
#     |   ├── 2019-07-10-Dems2020Night1Group1.rds
#     |   └── 2019-07-10-Dems2020Night1Group2.rds
#     ├── twitter
#     |   ├── 2019-07-06-Night01Tweets.rds
#     |   ├── 2019-07-06-Night01TweetsRaw.rds
#     |   ├── 2019-07-06-Night01TweetsUsers.rds
#     |   ├── 2019-07-06-Night02Tweets.rds
#     |   ├── 2019-07-06-Night02TweetsRaw.rds
#     |   └── 2019-07-06-Night02TweetsUsers.rds
#     └── wikipedia
#         ├── 2019-07-10-WikiDemAirTime01Raw.csv
#         ├── 2019-07-10-WikiDemAirTime02Raw.csv
#         └── 2019-07-25-PollingCriterionRaw.csv
```

The raw data are in four separate folders, each representing the data source (`538`, `google-trends`, `twitter`, `Wikipedia`).

Determine the size of the data files

Size can be a significant impediment to getting your work done quickly, so it's best to determine the size of the raw data files before importing.

The code chunk below tells us how big each file is and the folder in which its located.

```
# how big are each of these files?
fs::dir_info(path = "data", recurse = TRUE) %>%
    # only files
    filter(type == "file") %>%
    group_by(path) %>%
    # sort by size
    tally(wt = size, sort = TRUE)
# A tibble: 12 x 2
#    path                                                          n
#    <fs::path>                                             <fs::bytes>
#  1 data/raw/twitter/2019-07-06-Night02Tweets.rds                7.99M
#  2 data/raw/twitter/2019-07-06-Night02TweetsRaw.rds             7.87M
#  3 data/raw/twitter/2019-07-06-Night01Tweets.rds                6.92M
#  4 data/raw/twitter/2019-07-06-Night01TweetsRaw.rds             6.83M
#  5 data/raw/twitter/2019-07-06-Night02TweetsUsers.rds            1.9M
#  6 data/raw/twitter/2019-07-06-Night01TweetsUsers.rds           1.65M
#  7 data/raw/google-trends/2019-07-10-Dems2020Night1Group2.rds  111.47K
#  8 data/raw/google-trends/2019-07-10-Dems2020Night1Group1.rds  109.52K
#  9 data/raw/wikipedia/2019-07-25-PollingCriterionRaw.csv           722
# 10 data/raw/538/2019-07-06-Cand538Fav.csv                          581
# 11 data/raw/wikipedia/2019-07-10-WikiDemAirTime02Raw.csv           202
# 12 data/raw/wikipedia/2019-07-10-WikiDemAirTime01Raw.csv           191
```

The output tells us the twitter data files are the largest (`7.99M`). All of these files are small enough for RStudio.Cloud to handle, though.

Loading the data into R

We can import the data using the `01-import.R` file in the `code` folder. Feel free to open this file and examine its contents in the **Source** pane. We will use the `base::source()` function to run all of the code in the `01-import.R` file.

```
# import data using 01-import.R file
source("code/01-import.R")
```

The `01-import.R` file loads all of the data files into the RStudio.Cloud session. We can verify this by examining the contents of the environment with `base::ls()`.

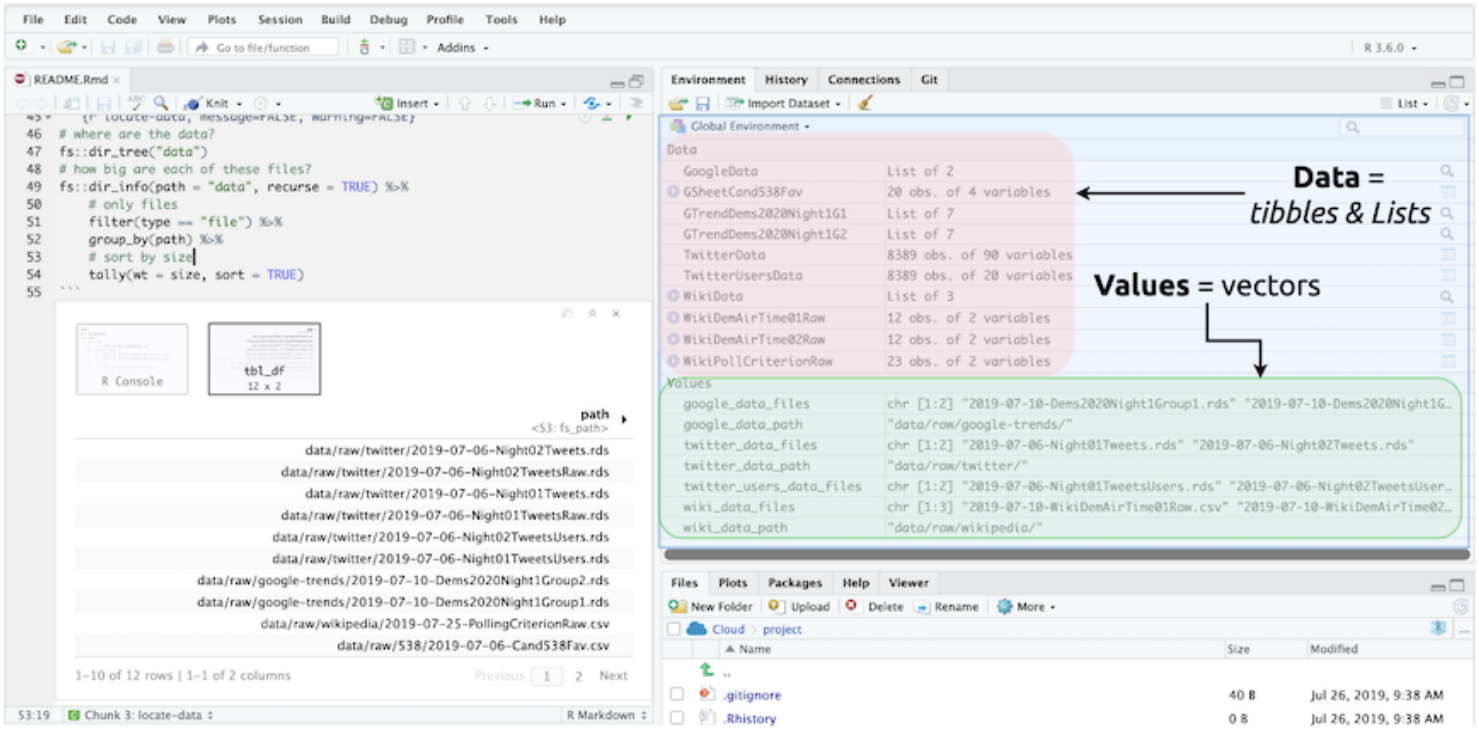
```
base::ls()
```

The output tells us the following objects are in our working environment.

```
#  [1] "google_data_files"
#  [2] "google_data_path"
#  [3] "GoogleData"
#  [4] "GSheetCand538Fav"
#  [5] "GTrendDems2020Night1G1"
#  [6] "GTrendDems2020Night1G2"
#  [7] "twitter_data_files"
#  [8] "twitter_data_path"
#  [9] "twitter_users_data_files"
```

```
# [10] "TwitterData"
# [11] "TwitterUsersData"
# [12] "wiki_data_files"
# [13] "wiki_data_path"
# [14] "WikiData"
# [15] "WikiDemAirTime01Raw"
# [16] "WikiDemAirTime02Raw"
# [17] "WikiPollCriterionRaw"
```

We can also see these objects in the **Environment** pane.



The *Global Environment* holds all of our imported data.

Documenting changes to our project with Git

So, we've imported some data into the RStudio.Cloud session, but we need to make sure we're keeping track of the changes to our files.

We can do this by checking our `git status` in the **Terminal** pane.

```
$ git status
On branch master

Initial commit

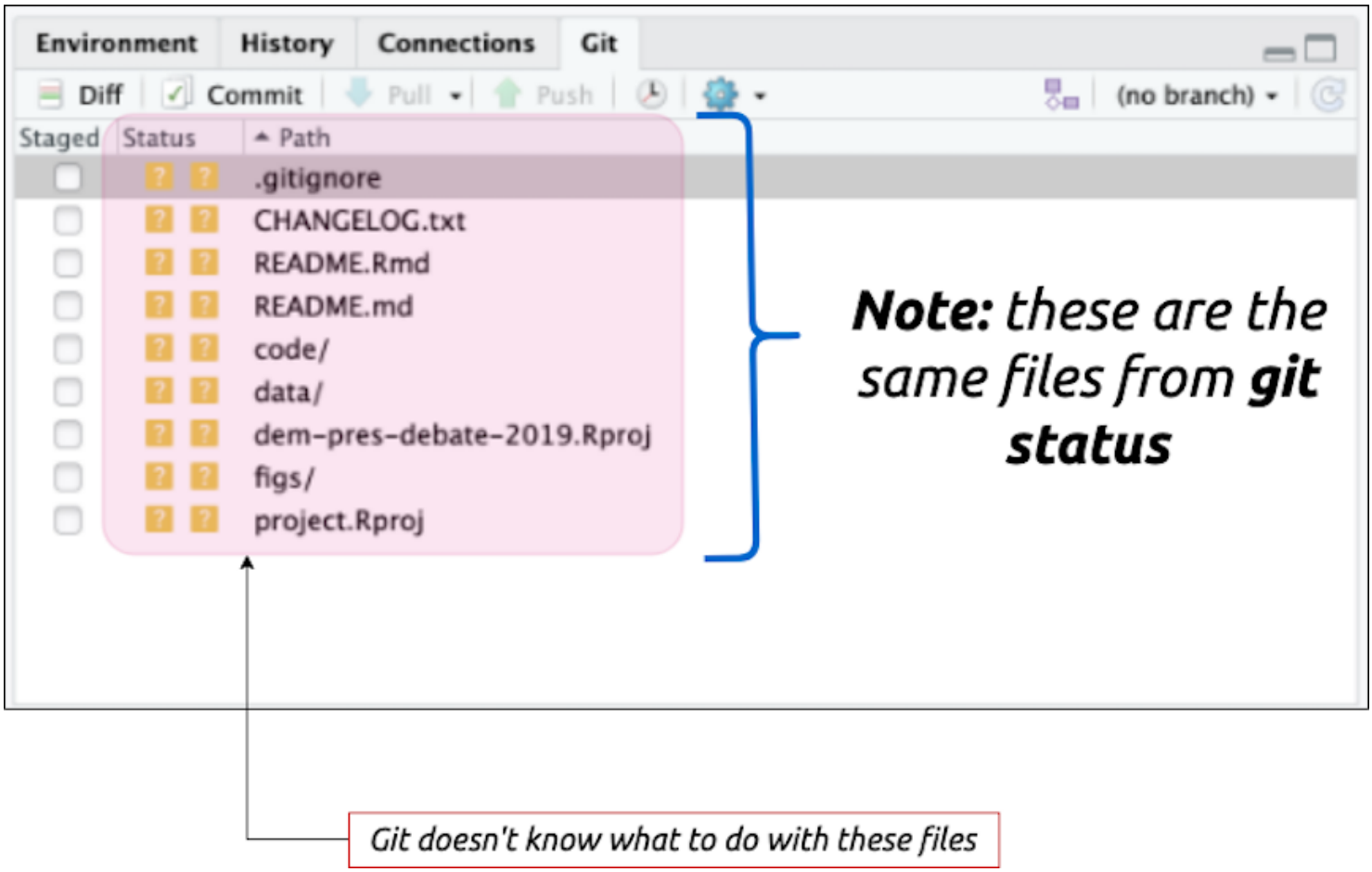
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        CHANGELOG.txt
        README.Rmd
```

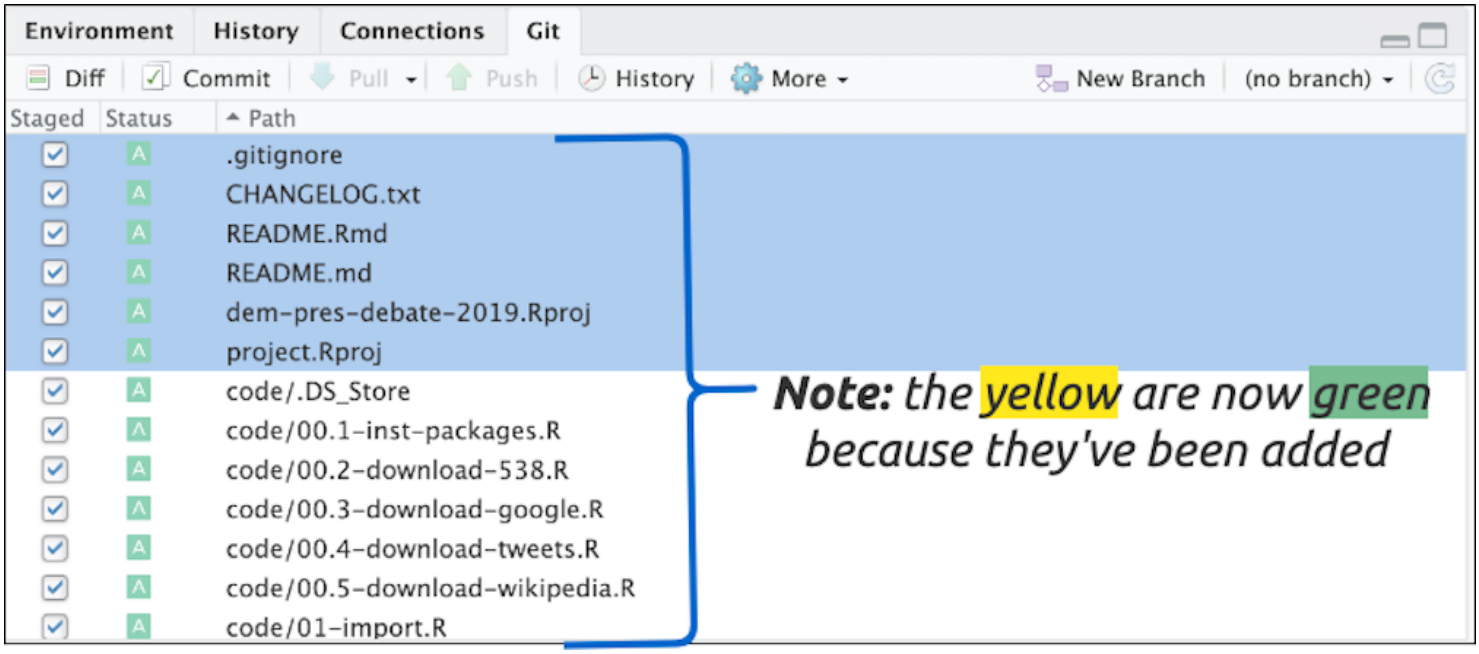
```
        README.md
        code/
        data/
        dem-pres-debate-2019.Rproj
        figs/
        project.Rproj

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

Git is telling us we have quite a few new files in our project (which we expected), but that to git to pay attention to them, we need to `add` them. We can do this in the **Git** pane RStudio provides (it's by the **Environment** pane).



We will click on each checkbox next to the files in the **Git** pane. Don't be alarmed if this list expands to include all the subfiles and folders.



Now we can re-check `git status` to see what just happened.

```
$ git status

On branch master

Initial commit

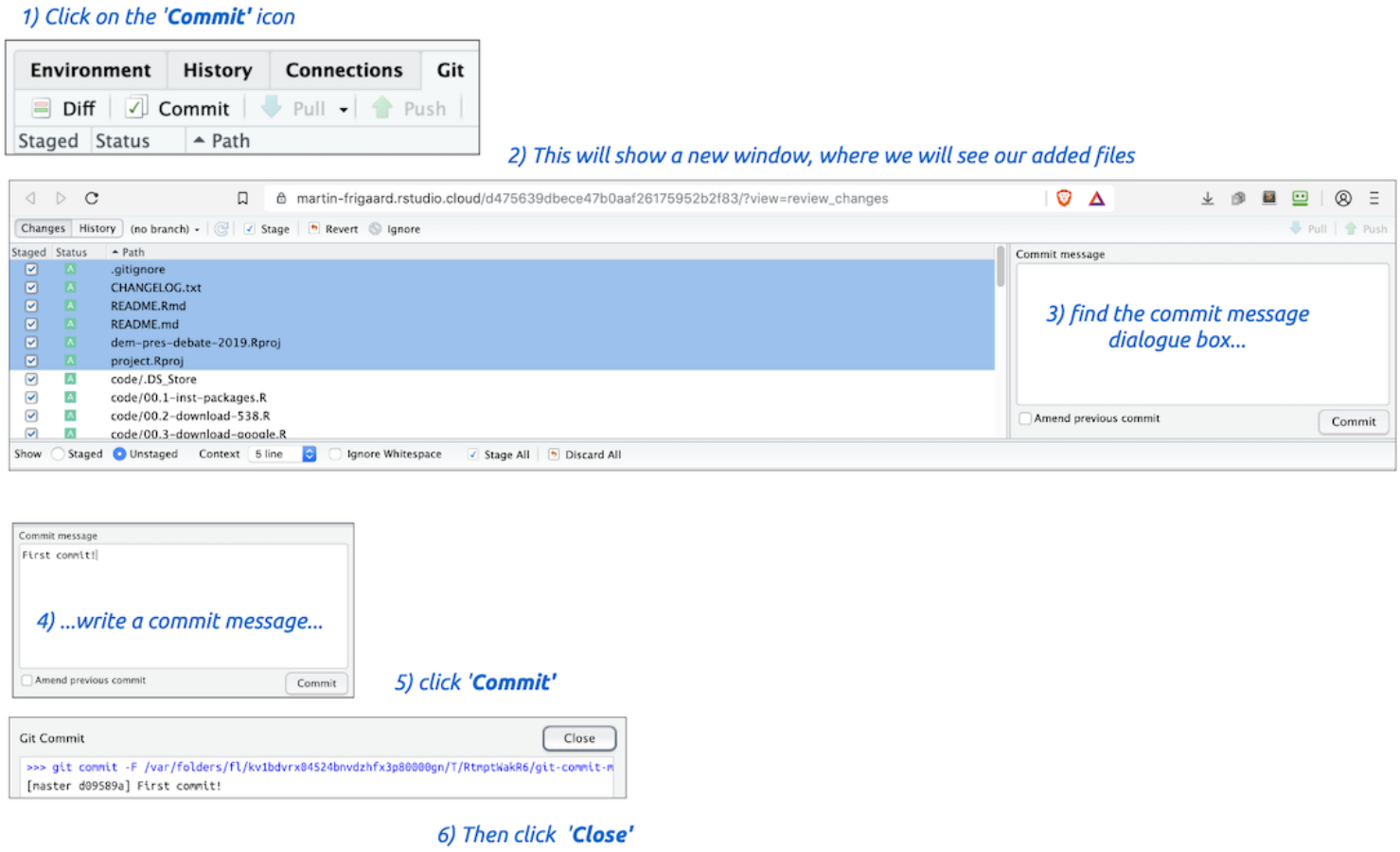
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   CHANGELOG.txt
        new file:   README.Rmd
        new file:   README.md
        new file:   code/.DS_Store
        new file:   code/00.1-inst-packages.R
        new file:   code/00.2-download-538.R
        new file:   code/00.3-download-google.R
        new file:   code/00.4-download-tweets.R
        new file:   code/00.5-download-wikipedia.R
        new file:   code/01-import.R
        new file:   code/02-wrangle.R
        new file:   data/.DS_Store
        new file:   data/processed/.DS_Store
        new file:   data/raw/.DS_Store
        new file:   data/raw/538/2019-07-06-Cand538Fav.csv
        # omitted
```

This long list of files shows all the contents have been added and are ready to be committed.

Committing the changes

If we click on the *Commit* icon in the **Git** pane, this will open a new window in RStudio.Cloud. In this window, we will enter a commit message ("First commit!"), and click the *Commit* button. We've outlined this entire process in the schematic below:



A quick git terminology overview

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

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

status - Git status will tell you what you've done and what is happening. You can check the status of a git repository with `git status` (use this liberally).

add - In order for Git to keep track of the changes we make to files, we have to tell it which files to pay attention to. We can do this using `git add`. The `git add -A` tells git to stage *ALL* the files that are in an initialized repo.

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. Commits are confusing because they can be 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 - repos are 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 repo has the following structure: `https://github.com/<username>/<repository_name>`.

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.

Part 6: Putting your project on Github

In the previous chapters, we've set up a Github account, learned how to download the files from a Github repository, upload files into RStudio.Cloud, create and run R code, and commit changes using Git.

In this final chapter, we are going to create some figures and graphs for this project, then put these changes on Github in a way for people to find and share.

Moving R code into the .Rmd file

In the last chapter, we used the `README.Rmd` file to upload our data into RStudio. We'll continue using this file to add the contents from one additional `.R` scripts, `02-wrangle.R`. We will then create new section and script for visualizing the data (`03-visualize.R`).

We've provided a lot of code and comments in the `02-wrangle.R` script for you to explore, revise, and adapt to your liking. In the next few sections, we are going to move the code from the `02-wrangle.R` script into two new sections of the `README.Rmd` (you probably guessed it "`Wrangle`").

A quick lesson in compassionate programming

Your code will always be communicating to at least two audiences: your computer, and your future self. Be nice to both of them!*

Things like the pipe `%>%` in R can help with clarity. The pipe is part of the `magrittr` package and it takes code written like this:

```
outer_function(inner_function(Data_X), Data_Y)
```

And makes it look like this:

```
Data_X %>% # do this
   inner_function() %>% # then do this
   outer_function(Data_Y)
```

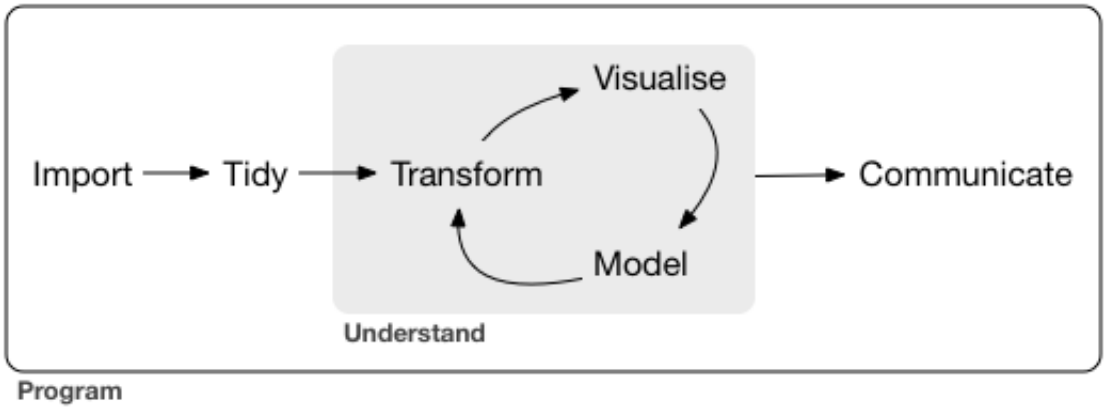
`%>%` is a form of syntactic sugar, which is a fancy way of saying "*something that helps us communicate better.*"

You'll see the pipe throughout the project's R code files, and you can always read it as, "*do this, then do this.*"

Wrangling code

Data wrangling (or cleaning, or munging) is whatever steps need to be taken to take the raw data (which we always remember not to change) into something we can use to create a table, visualization, model, etc.

The `02-wrangle.R` script prepares the data from the `01-import.R` so that they can be used for the visualizations. If you think back to the process outlined in the figure in R for Data Science, you will notice that wrangle isn't listed explicitly. This is because both '**Tidy**' and '**Transform**' would be considered wrangling steps (both of these need to happen before any visualizations or models can be properly run).

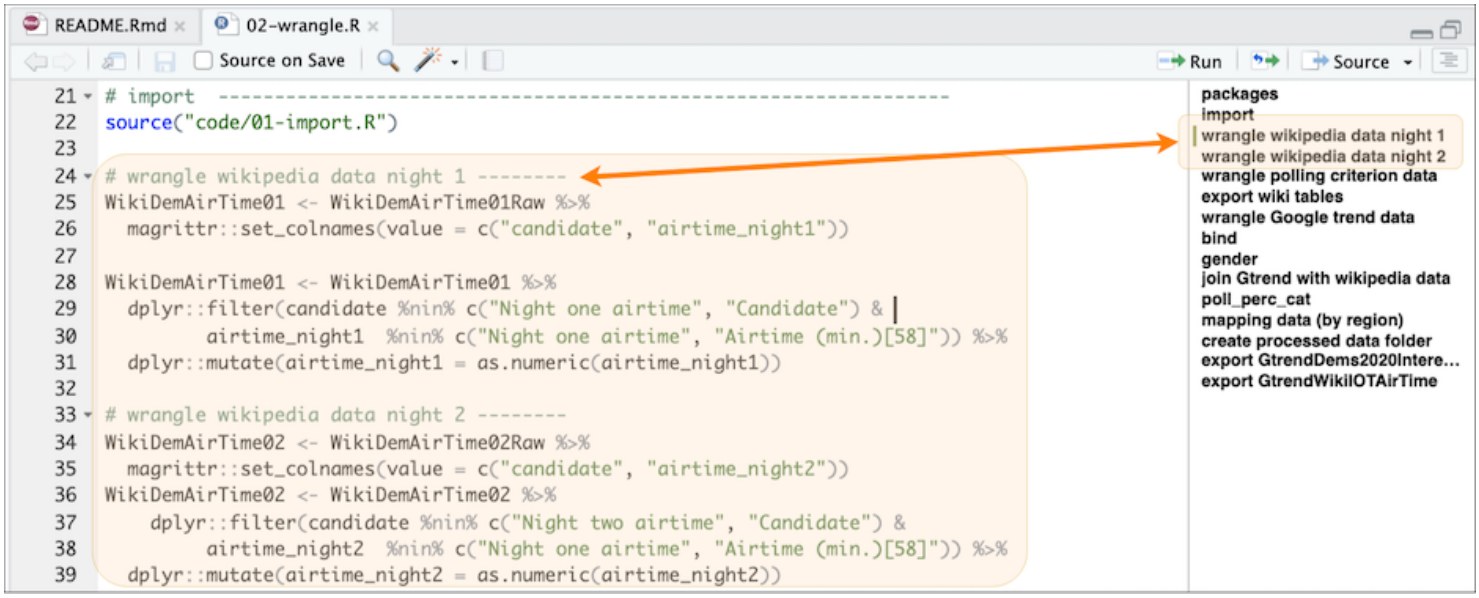


The simplest way to include this script in the README.Rmd file is to create a code chunk, insert the `base::source()` function, and enter the path to the `02-wrangle.R` file.

However, we want to be nice to our future selves, so we will include some language that describes what the functions are doing above each code chunk.

Wrangling the data sets

The first data that need to be wrangled are the Wikipedia tables (seen in the `02-wrangle.R` file on the section below).



These data were stored on the web in Wikipedia (`.html`) tables. We will create some new column names, remove some columns that used to be headers, and make the airtime variable numeric.

The polling criterion Wikipedia data starting at the section titled, `wrangle polling criterion data`. This section actually creates a list of candidates (in `cand_names_wiki`) and uses it to `filter` out the observations we want. Check out this webinar to get an understanding of how `dplyr`'s verbs work.

```
# wrangle polling criterion data ----------------------------------------
# create list from names using dput()
# dput(WikiPollCriterionRaw[ 1:11, 1])
```

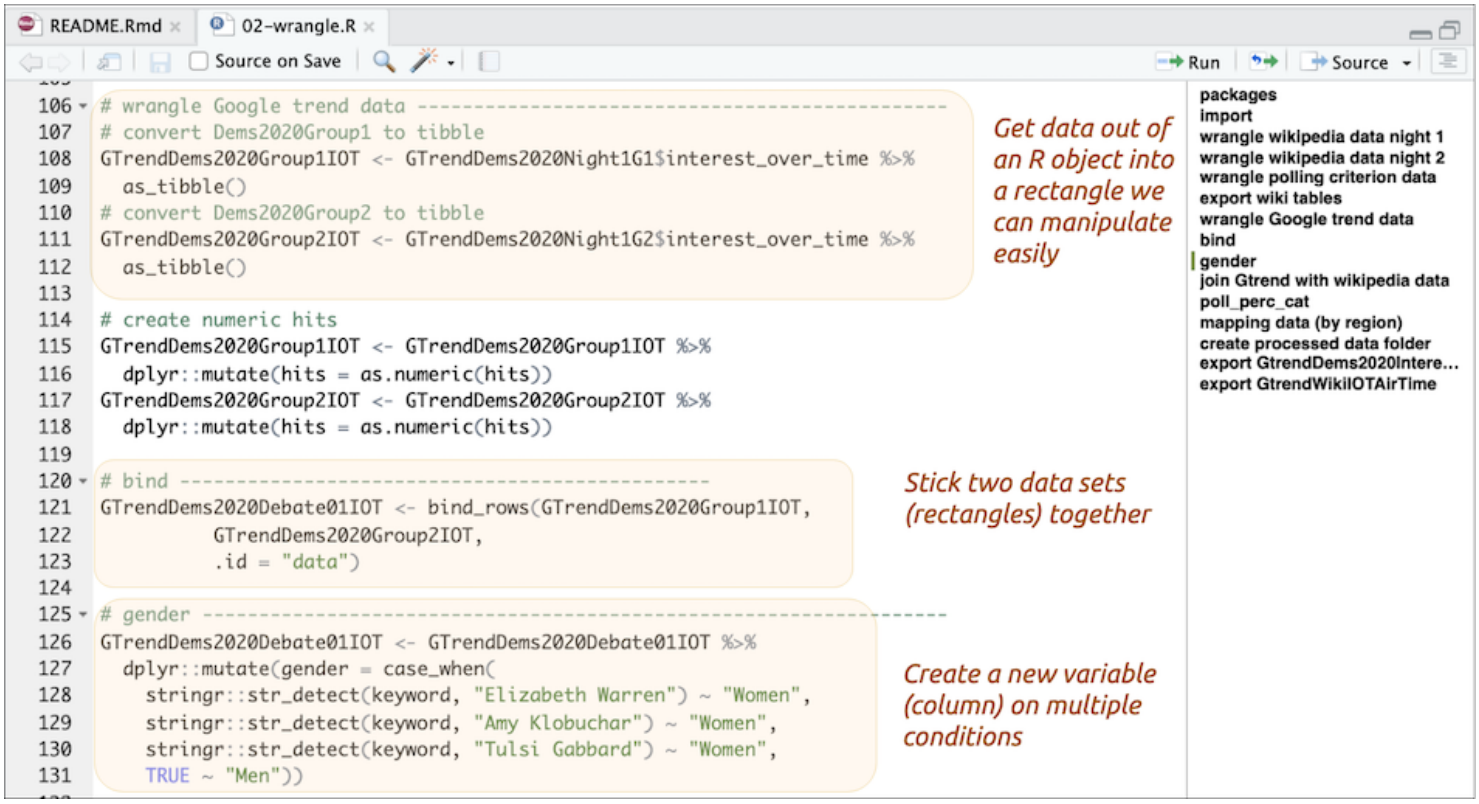
```
cand_names_wiki <- c("Warren[note 2]",
                     "O'Rourke[note 2]",
                     "Booker[note 2]",
                     "Klobuchar[note 2]",
                     "Castro[note 2]",
                     "Gabbard",
                     "Ryan",
                     "Inslee",
                     "de Blasio",
                     "Delaney")
# subset WikiPollCriterionRaw with list from above
WikiPollCriterion <- WikiPollCriterionRaw %>%
  # this will remove all candidates not listed above
  dplyr::filter(`Candidates drawn for the June 26 debate` %in% cand_names_wiki)
```

After the Wikipedia tables have been wrangled, the script exports these files to a new `processed/` folder. This helps ensure they won't be accidentally altered or mistaken for the data files in the `raw/` data folder.

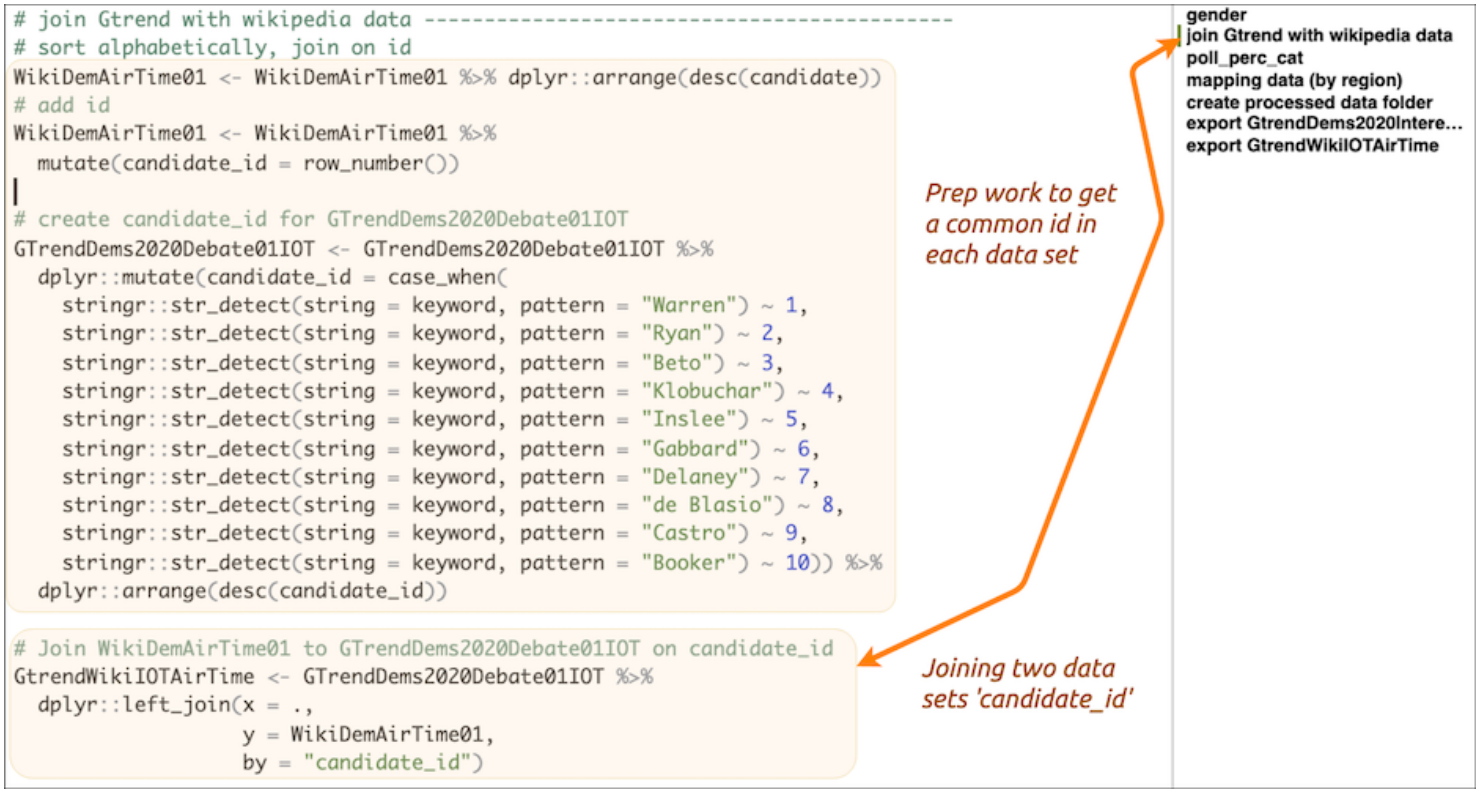
The export section also timestamps each file so we know the last time it was created. Read more about importing and exporting data in this RStudio cheatsheet.

The Google trend data are a little more complicated because they come into RStudio.Cloud as a list, which is a data container in R that doesn't have to be rectangular.

The image below outlines what each portion of code is doing. These are fairly common wrangling tasks, so we recommend going back or bookmarking these files as a reference.



We have different sources of data in RStudio right now (Wikipedia and Google trend data). They both have information on Candidates though. Often times we'll want to join two (or more) data sets on a common column (like candidates). We will perform an example of this in the section outlined below.




```
# join Gtrend with wikipedia data ------------------------------------------
# sort alphabetically, join on id
WikiDemAirTime01 <- WikiDemAirTime01 %>% dplyr::arrange(desc(candidate))
# add id
WikiDemAirTime01 <- WikiDemAirTime01 %>%
  mutate(candidate_id = row_number())

# create candidate_id for GTrendDems2020Debate01IOT
GTrendDems2020Debate01IOT <- GTrendDems2020Debate01IOT %>%
  dplyr::mutate(candidate_id = case_when(
    stringr::str_detect(string = keyword, pattern = "Warren") ~ 1,
    stringr::str_detect(string = keyword, pattern = "Ryan") ~ 2,
    stringr::str_detect(string = keyword, pattern = "Beto") ~ 3,
    stringr::str_detect(string = keyword, pattern = "Klobuchar") ~ 4,
    stringr::str_detect(string = keyword, pattern = "Inslee") ~ 5,
    stringr::str_detect(string = keyword, pattern = "Gabbard") ~ 6,
    stringr::str_detect(string = keyword, pattern = "Delaney") ~ 7,
    stringr::str_detect(string = keyword, pattern = "de Blasio") ~ 8,
    stringr::str_detect(string = keyword, pattern = "Castro") ~ 9,
    stringr::str_detect(string = keyword, pattern = "Booker") ~ 10)) %>%
  dplyr::arrange(desc(candidate_id))

# Join WikiDemAirTime01 to GTrendDems2020Debate01IOT on candidate_id
GtrendWikiIOTAirTime <- GTrendDems2020Debate01IOT %>%
  dplyr::left_join(x = .,
                   y = WikiDemAirTime01,
                   by = "candidate_id")
```

The process usually isn't so involved, but we included extra to give more explicit instructions. Be sure to check out the relational data chapter the R for Data Science book.

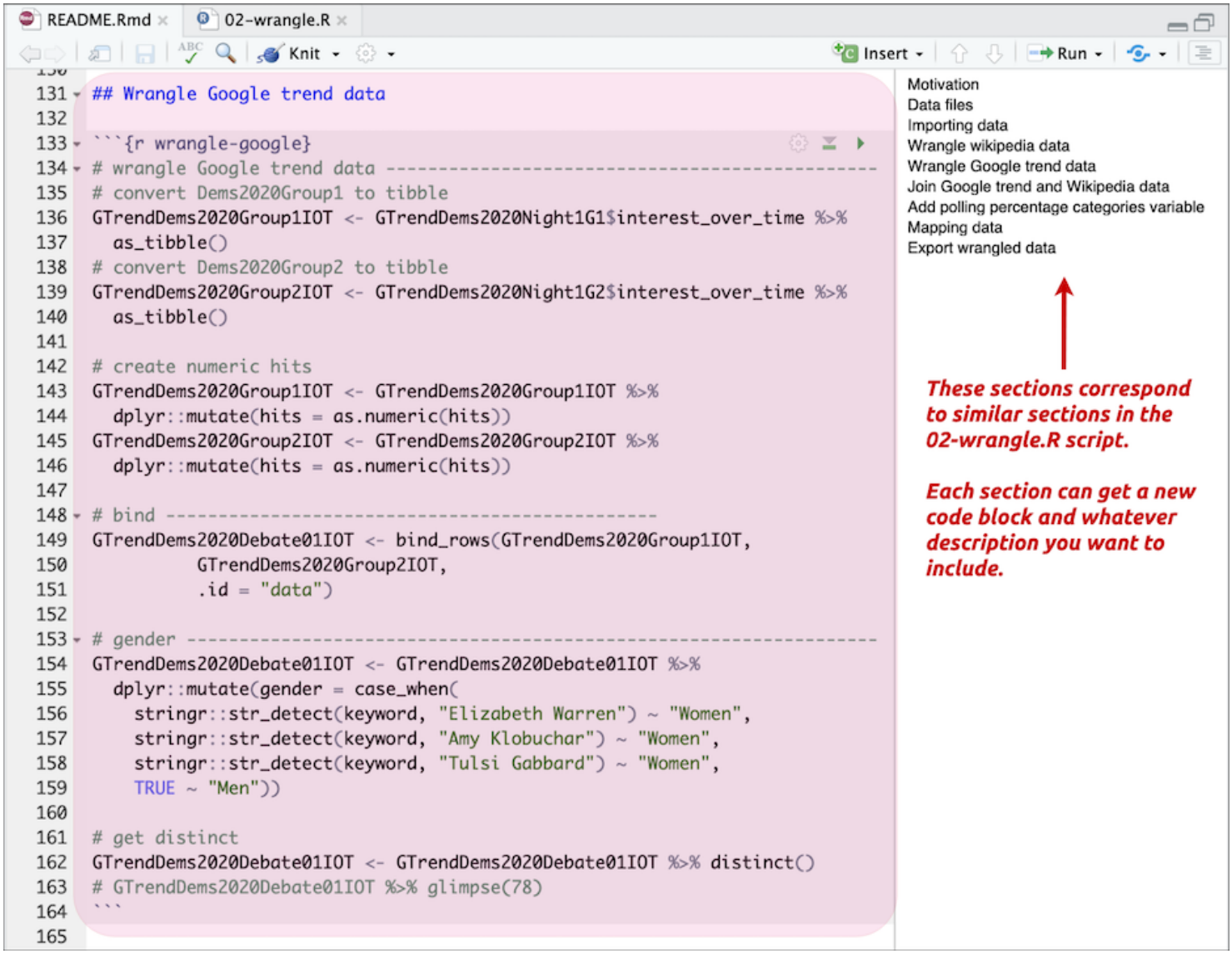
We'll also be creating a map with the Google (or Twitter) data. Doing this requires another common task, which is loading a dataset from a package in R. The code below loads a state-level map into RStudio.Cloud and joins it to the Google trend data.


```
# mapping data (by region) -------------------------------------------------
# convert to tibble (another data structure in R)
GtrendDems2020IBRGroup1 <- tibble::as_tibble(GTrendDems2020Night1G1$interest_by_region)
GtrendDems2020IBRGroup2 <- tibble::as_tibble(GTrendDems2020Night1G2$interest_by_region)
# bind Dems2020IBRGroup1 Dems2020IBRGroup2 together
GtrendDems2020IBR <- bind_rows(GtrendDems2020IBRGroup1,
                       GtrendDems2020IBRGroup2, .id = "data")

# convert the region to lowercase
GtrendDems2020InterestByRegion <- GtrendDems2020IBR %>%
  dplyr::mutate(region = stringr::str_to_lower(location))

# create a data set for the states in the US
statesMap = ggplot2::map_data("state")
# now merge the two data sources together
GtrendDems2020InterestByRegion <- GtrendDems2020InterestByRegion %>%
  dplyr::inner_join(x = .,
                    y = statesMap,
                    by = "region")
```

The Google trend data are also exported with time-stamp into the `processed/` folder. We should continue adding the code into the `README.Rmd` file until we're confident all the functions will run and we don't see any errors.

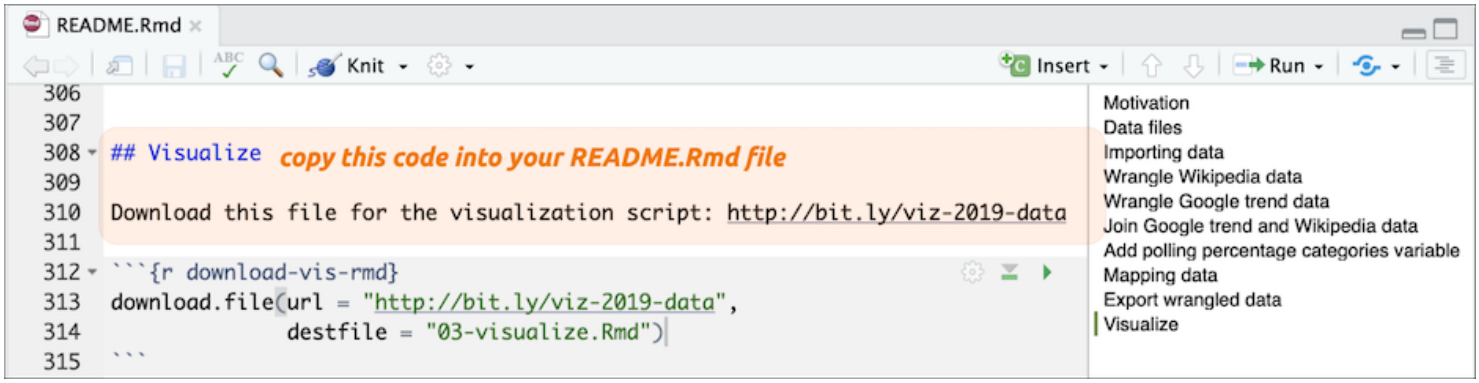


Note: The `02-wrangle.R` file is in the `code/` folder, but you won't have to alter the file paths because you're using an RStudio project file. Read more about how these are so helpful to your workflow here.

Visualizations

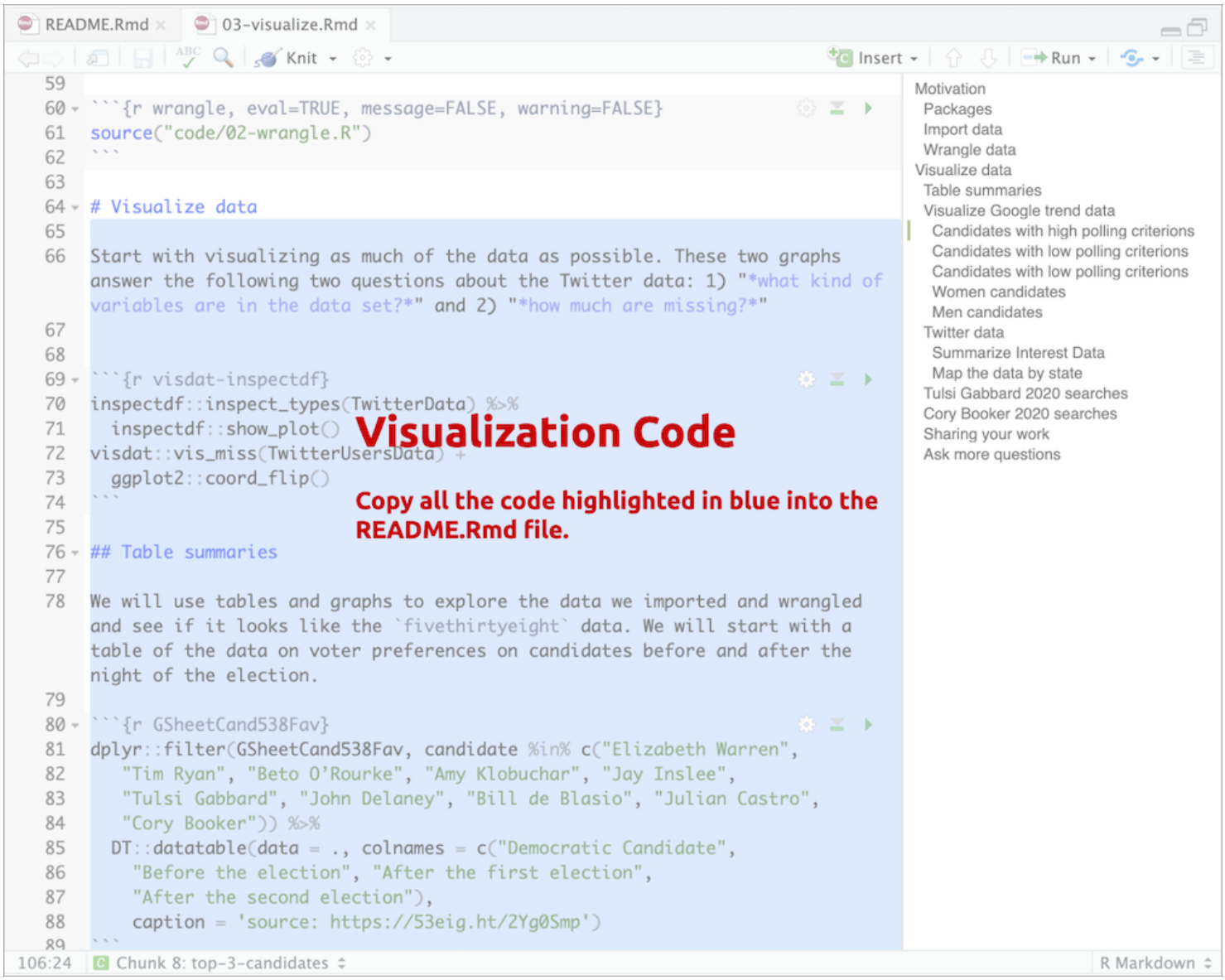
OK, we've completed our section for the wrangling the data. We are going to insert a divider (`***`) and start a new visualize section (`## Visualize`) in the `README.Rmd` file.

We've created a `03-visualize.Rmd` file for you to download from Github. You can do this by typing the following code into your `README.Rmd` file:



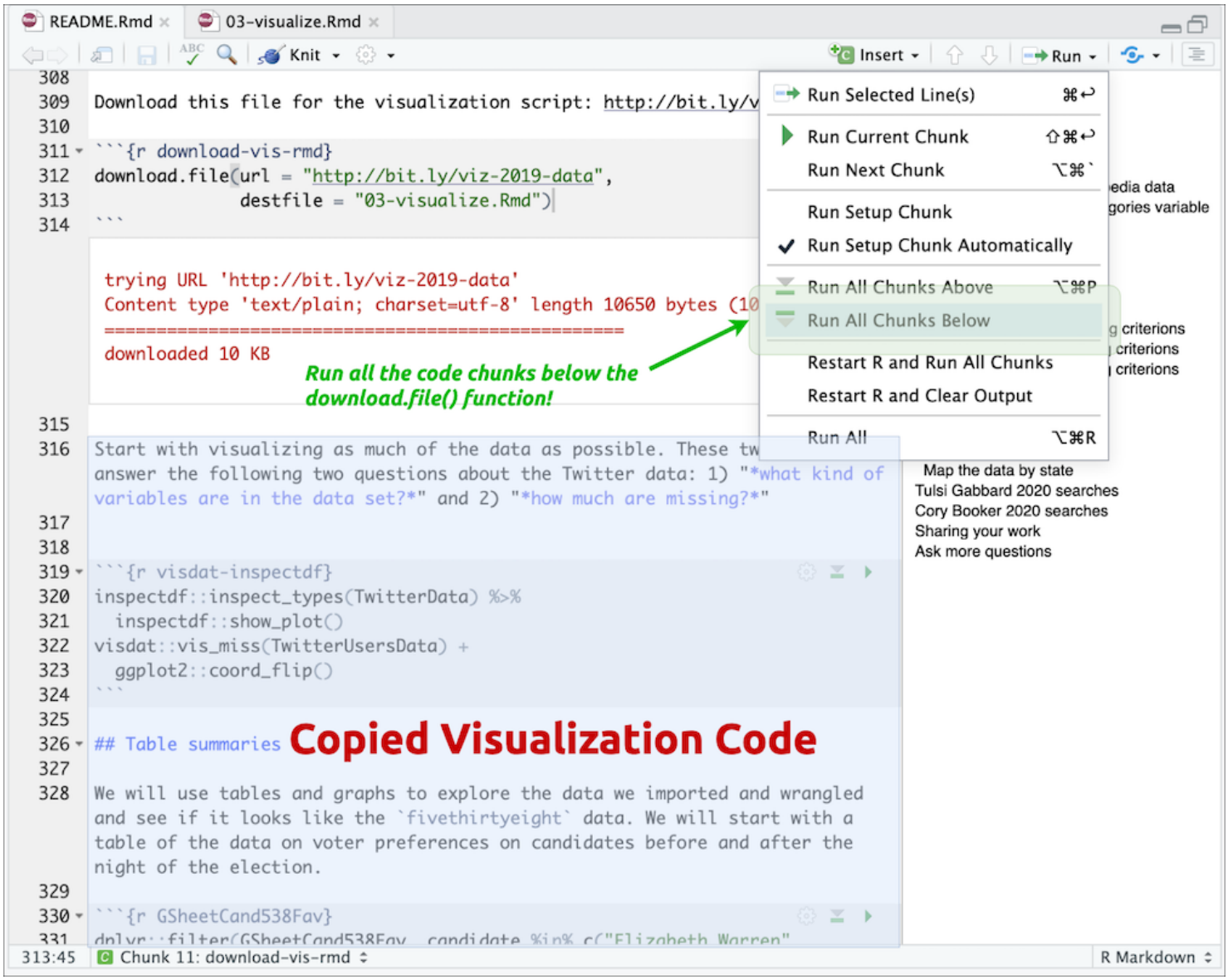
The hyperlink is here: `http://bit.ly/viz-2019-data`

After we've downloaded our `03-visualize.Rmd` file, we can open this file and copy the code starting from the line just below the `# Visualize data` heading (it should be on about line `65`) and extending all the way to the end of the file.



After selecting the code from `03-visualize.Rmd`, we should click on the line directly under the previous code chunk we used to download the `.Rmd` file.

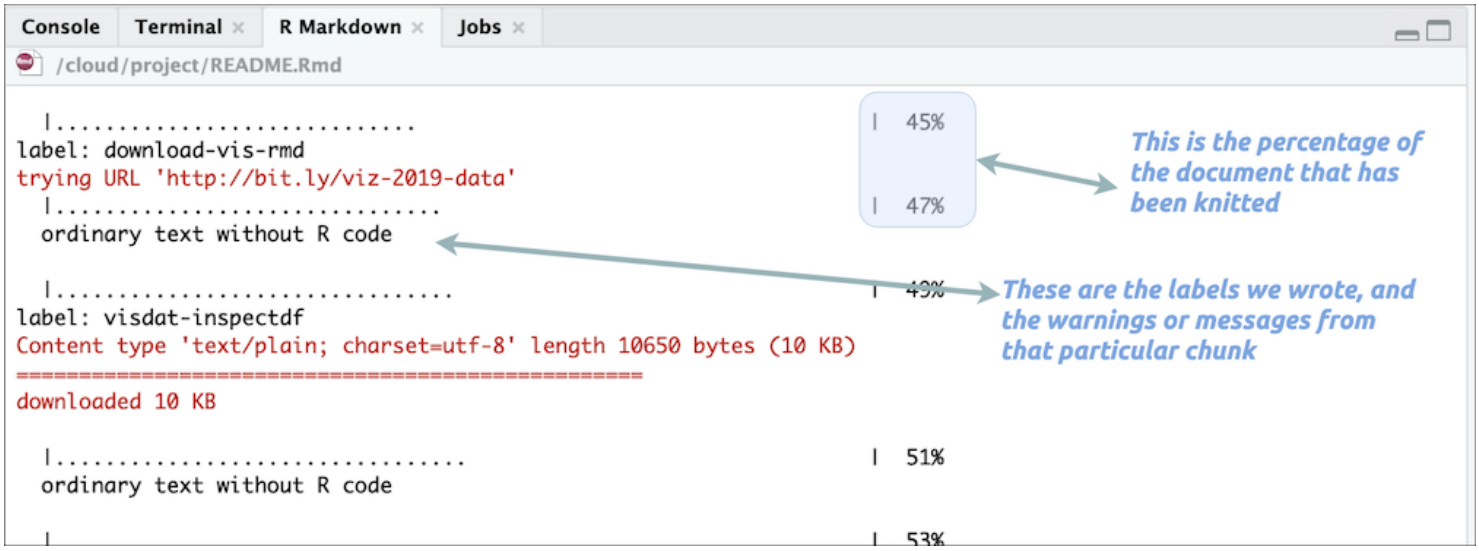
After pasting the code from `03-visualize.Rmd` into the `README.Rmd` file, we can click on the *Run* > *Run All Chunks Below* (this will run all the code starting at line `319` until the end of the document).



Running the code will create multiple tables and figures in the `README.Rmd` file. We'll go over these in more depth below. For now, we'll follow the directions at the bottom of the pasted code and "*Click `knit` to get the markdown file to share.*"

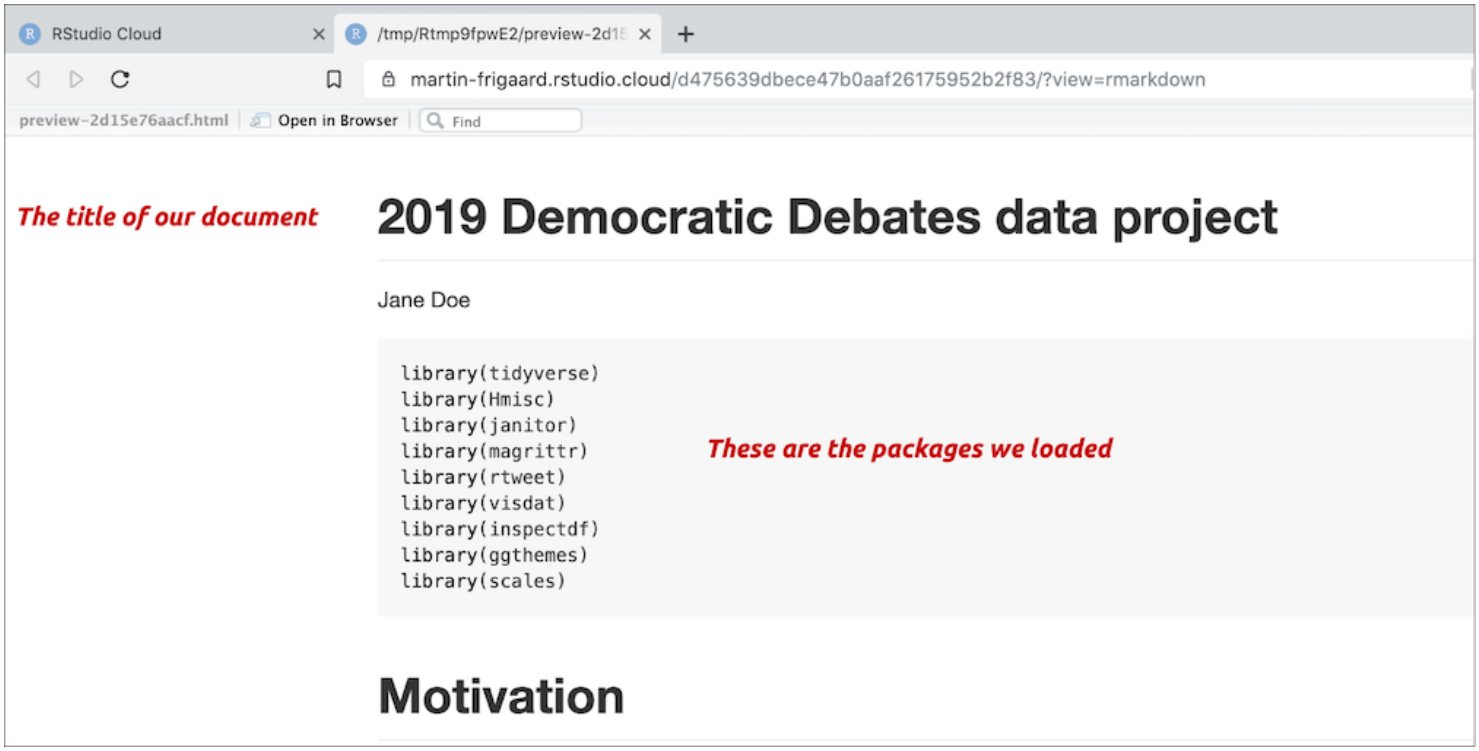
Knitting RMarkdown files

Clicking *Knit* (or clicking `shift+cmd+k`) activates the **Markdown** pane in RStudio.Cloud, and we see the code chunks being run for the entire document.



When the knitting process completes, a new browser window will pop up with our `README.md` document. The `README.md` will have sections of formatted text (from the Markdown), R code, and the various outputs.

The top of the file should list the title and the packages:



Lets scroll down to the visualize section and look at the section titled, **Candidates with high polling criterions)**. We can see the different parts of the Rmarkdown file in the image below:

Header & text for this section

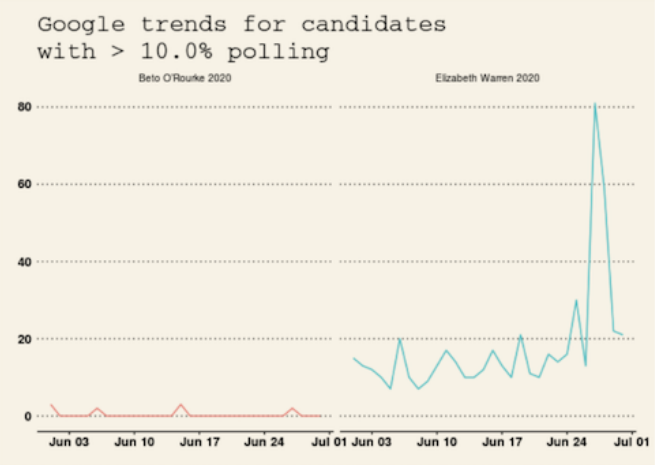
Candidates with high polling criterions

When we look at the candidates with the highest percent of polling criterion on the 26th, we see the following:

Code and comments for this graph

```
GtrendWikiIOTAirTime %>%
    # limit to only the candidates with more than 10% of voters
  dplyr::filter(poll_perc_fct == "> 10.0% of voters") %>%
    # put data on the x,
  ggplot2::ggplot(aes(x = date,
                      # hits on the y
                      y = hits,
                      # color it by keyword
                      color = keyword)) +
    # use a line
  ggplot2::geom_line(aes(group = keyword), show.legend = FALSE) +
    # labels
  ggplot2::labs(
    x = "Date",
    y = "Google search hits",
  subtitle = "Google trends for candidates \nwith > 10.0% polling") +
    # add themes from ggthemes
  ggthemes::theme_wsj(base_size = 9.5,
                      title_family = "mono") +
    # use facets for both candidates on the same graph
  ggplot2::facet_wrap(~ keyword, ncol = 2)
```

Graph output

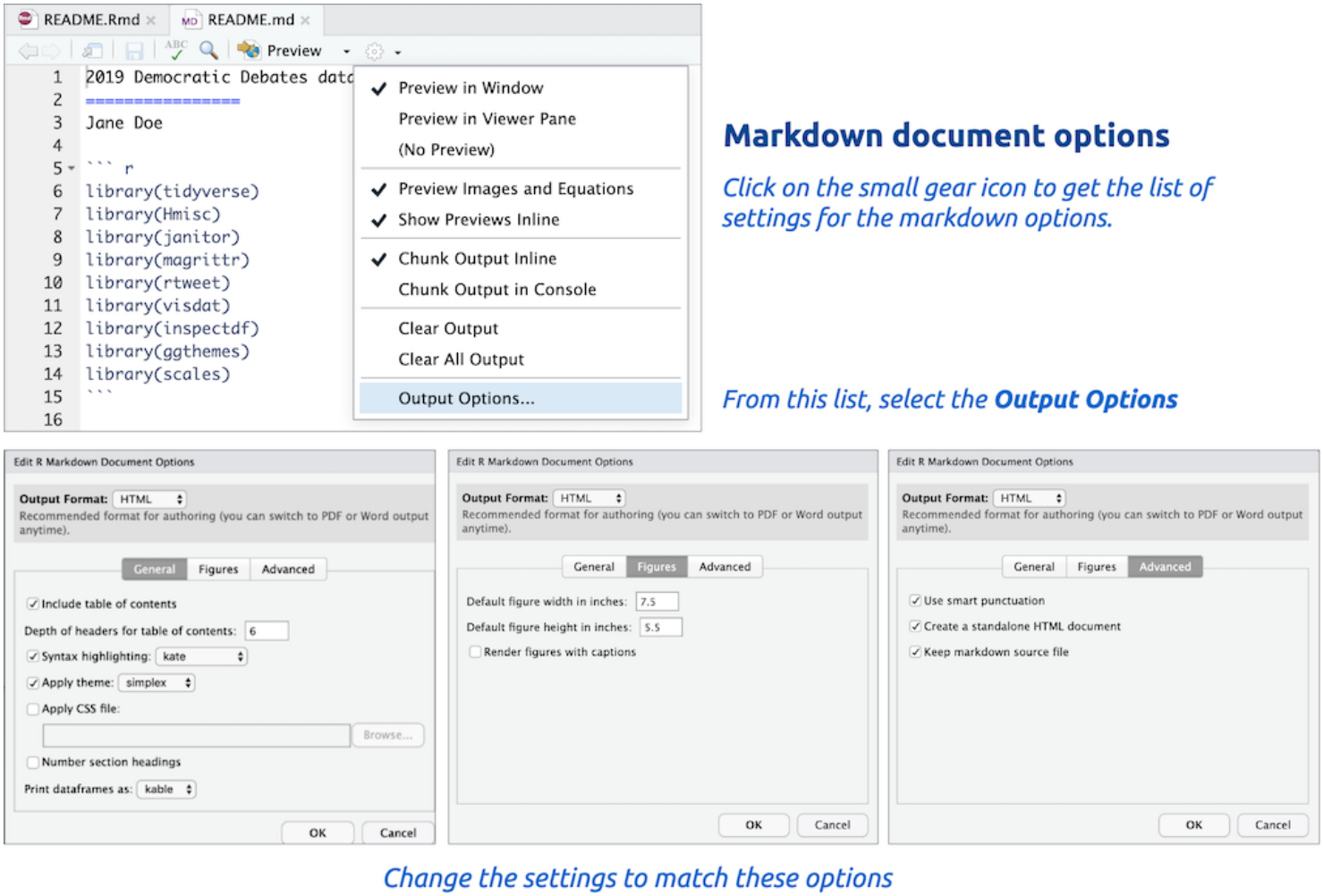


More text interpreting the output

This shows Warren doing well after the first night. If we narrow this down to the week of the debates and widen the list of candidates, we see the following:

The file output is actually a *Preview* of our markdown file (`README.md`). Our browser renders the markdown as a webpage (`README.html`).

Let's keep a version of this file in `.html`. We can do this by opening the `README.md` file in the **Source** pane, and clicking on the small gear next to the *Preview* button. Follow the directions in the figure below to setup the `.html` output file.



When we click *Ok* and look at the top of the `README.md` file we see another `YAML` header.

```
---
output:
  html_document:
    df_print: kable
    fig_height: 5.5
    fig_width: 7.5
    highlight: kate
    keep_md: yes
    theme: simplex
    toc: yes
    toc_depth: 6
---
```

The settings displayed above are some of the ways we can customize our `.Rmd` files. Read more about the html documents here in the Rmarkdown guide.

To see what these settings do, we will click *Preview* on the `README.md` file.

This conversion is incredibly handy for weaving formatted text, code, and media (tables, images, and graphs).

Extracting the .R from the .Rmd

But now we have all our visualize code in the `03-visualize.Rmd` file–what if we wanted this code in an .R script?

We can run the following code in the **Console** pane.

```
knitr::purl("03-visualize.Rmd")
```

We'll see the following script file is generated.

```

```

Push the changes to Github