



Showing your work

Martin Frigaard

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

This version was published on 2019-06-13



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

© 2019 Martin Frigaard

Contents

1. showing-your-work	1
2. Part 1: getting started in modern research tools	2
Who should read this	3
Assumptions we've made	3
Language and style	3
You've made it!	4
Researchers to emulate	7
Overcoming language barriers	8
Practicing your communication	9
The path forward	10
3. Part 2: the toolset	12
Open-source tools	12
The computer science in science	13
Command line interfaces (shells, terminals, and text)	16
Use R and RStudio	18
Why should I use R/RStudio and not Python?	18

CONTENTS

Getting R/RStudio set up	20
4. Part 3: An example project	22
Operating systems	23
Some common Terminal commands	25
Why am I learning the command line?	27
Getting more help	33
5. Part 4: keeping track of the changes to your work	35
Tracing your steps	35
“Keep Knowledge in Plain Text”	36
Git	39
Installing Git	41
Adding a key SSH in Terminal	44
Create the ~/.ssh/config file	45
6. Conclusion	47

1. showing-your-work

2. Part 1: getting started in modern research tools

Congratulations! You've just entered graduate school, and you're on your way to more specialized education and training. Hopefully, after a few challenging years and sleepless nights, you'll be on your way to a rewarding career. Provided you did your [research](#), a graduate school degree is still likely to be an excellent investment for your future.

I loved my graduate school experience. I made friends and colleagues I'm still in contact with, and received some excellent advice from outstanding (and unexpected) mentors. As I write this, the cost of college tuition has grown faster than the amount of [financial aid available](#), and studies are showing degrees in some fields [have more value than others](#). In recognition of these changes, my colleague and I have decided to write this technical guide. We felt a few topics were missing from our graduate education, and wanted to share what we've learned in the hopes it would make the transition from school to work a little less shocking.

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

Who should read this

We wrote this book for any graduate student who will be doing research. Whether you're getting a Masters or a Ph.D., if you're doing research, this book is for you.

Assumptions we've made

We assume you've been working on a computer, but mostly to write papers, send/receive emails, and explore the internet. Everyone enters graduate school at different times in their lives, so there is a good chance some of you will know the materials we are covering. If this is the case, hopefully, we cover it in a novel and exciting way that doesn't make reading it feel like a waste of time.

Language and style

Although one person put this information together in this particular format, we use the plural 'we' because this is the result of many conversations, emails, comment threads, and communications that could not have happened in isolation.

You've made it!

You have entered graduate school in amazing times. The internet has made information easily accessible to everyone, and most people are walking around with more computational power in their pockets than previous generations ever imagined possible. With a laptop and internet browser, we can get access to nearly all the accumulated knowledge of the human species (and an unreasonable number of cat pictures). Part of this reason this is all so amazing is the relatively short time it took to happen.

Here is an example to demonstrate this rate of change: in a [2000 paper in Nature](#) by Steve Lawrence and C. Lee Giles titled, “Accessibility of information on the web.” The authors open with the jaw-dropping statistic that the internet is “*800 million pages, encompassing about six terabytes of text data on about 3 million servers.*”

Sixteen years later, Google claims to be aware of [130 trillion pages](#) across the web. Two years after that, Wikipedia has 5.71 million articles and is contributed to by [127,026 Wikipedians](#) (people who actively edit Wikipedia articles).

We've never had more access to information than we do right now, and it's unlikely that there will ever be less available information any time soon.



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

All that information on the internet also means there are competing voices for

people's attention. All that excellent work you'll do in graduate school needs to be discoverable on the internet, which means making sure you have more than just a single thesis, dissertation, or manuscript. If your future collaborators, prospective employers, and fellow graduate students are going to be able to find a catalog of what you've done, you'll need examples written for multiple audiences.

Don't rely solely on your scientific papers to showcase your work

"What a strange document a scientific journal article is. We work on them for months or even years. We write them in a highly specialized vernacular that even most other scientists don't share. We place them behind a paywall and charge something ridiculous, like \$34.95, for the privilege of reading them. We so readily accept their inaccessibility that we have to start "journal clubs" in the hopes that our friends might understand them and summarize them for us." - [How to read a scientific paper](#)

Be honest—how many theses/dissertations have you read? How many peer-reviewed articles would you read for pure enjoyment? I suspect that even if you asked your most bibliophilic friends what their favorite thesis is, or what dissertation they think everyone *must* read, they couldn't give you a single example? These documents aren't a waste of time—they serve a different purpose (and it's not to make sure all of your hard work reaches a broad audience).

Even if you decide to convert your graduate research into a peer-reviewed manuscript,

it'll be for a very niche audience, and rarely in a way that makes the contents interesting beyond a few researchers who are closest to the subject matter.

Scientific papers are still essential to advancing science (and your career as a scientist), but they're not a demonstration of all the skills you've developed in graduate school. These artifacts represent the end of a long process in which you've demonstrated many different skills (reading and summarizing previous research, designing a study, data analysis, and communication). This sentiment is summarized well in the quote from [Jonathan Buckheit and David Donoho at Stanford](#),

“An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures..”

And like most advertising, peer-reviewed papers leave many essential details out. For example, journal articles can't document how you ended up at the particular research question or hypothesis you tested (and why you had to change it). Neither the thesis/dissertation nor the peer-reviewed paper can document what you did in graduate school. This includes things like:

- Coming up with an idea,
- Turning that idea into a research question,
- Convincing people of that idea (or some version of it),
- Collecting your data,

- Teaching labs/lectures,
- Recruiting volunteers,
- Entering your data,
- Managing your committee member's expectations (and egos),
- Cleaning your data,
- Persuading someone to read early drafts,
- Politely reminding your committee to give you constructive criticism (promptly so you can graduate), and
- Solving a million other little day-to-day problems.

A research paper written and edited by committee and a powerpoint presentation won't adequately capture your graduate school adventure, so if you want your work to count, it needs to be communicated to wider audiences, or even in other mediums.

Researchers to emulate

This book will show you some of the tools to make your work more discoverable. We'll introduce you to the technologies, methods, and places used by scientists who have successfully communicated their work. These scientists have used the internet as a tool to engage with broader audiences, create better tools for doing science, document some of their daily struggles/successes, and share more about what it means to conduct research.

For example, [Lucy D'Agostino McGowan](#) 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](#). Her work is *highly discoverable* and showcases a wide range of skills.

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

Both of these researchers did two things very well: they created outstanding work, and they put it online for people to find. Of course, they had to know their subject areas, and have something worth sharing online, but they didn't wait until they were done with their research, either. They started engaging with people while they were completing their research training.

Overcoming language barriers

"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 the jargon. Because this book sits at the intersection of computer science, statistics, and web technologies, the 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.

Practicing your communication

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

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

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

When we were kids in math class, the teacher would ask us to “show our work.” Teachers gave these instructions so they could follow our thought processes through a problem, and see where our thinking was incomplete or mistaken (and

probably also to make sure we weren't looking at someone else's paper). If you regularly show your work, you'll be able to follow your line of thinking as you progress through graduate school. More importantly, people who find your work will see you're an actual human, with a full range of human experiences, and not just another CV and headshot.

The path forward

Communicating our work should be the goal of anyone doing research. After leaving graduate school, I realized how few people had the skills I was taking into the world (and how many people would benefit from them). One of the most attractive things about understanding math and science is that small investments in understanding can yield significant returns.

I've realized we don't communicate the importance of being 'good enough' at math and science, and that's a shame. Being 'good enough' means you could read about technology and be capable of distinguishing it from magic, or that you can imagine a metric that might matter to your business or personal life, and then devise a way to measure it. Perhaps more importantly, an emphasis on a 'good enough' understanding of math and science could help break down the "us vs. them" mentality that arises when science inevitably makes its way into the public sphere.

In our opinion, the job of a scientist or researcher isn't done when they defend a thesis/dissertation and get their degree; it isn't done when their research has

been submitted and accepted to a conference or high-impact journal; it isn't even done when someone reads that article or attends their talk.

As researchers, we consider our jobs are done when someone has heard and understood our research, and knows the impact it will have on the world.

::Footnotes::

1. The scientific journal industry is not looking out for your best interests. They have a [clearly unethical business model](#), even [prominent universities can't afford their prices](#) (which means fewer people reading your work), and they won't [compensate](#) you for your efforts.
2. The metrics previously used to measure success in academic publishing are [unreliable and susceptible to being gamed](#). You don't want to have these be your sole measure of productivity.
3. Read more about the reproducibility crisis in science [here](#), and why it's so important to share more than just your papers

3. Part 2: the toolset

This text is an opinionated technical manual for graduate students to share their work with a broad audience through a variety of mediums. We'll be making recommendations based on what we were taught to use in school, learned to use at various jobs, and what we've abandoned. We are not saying there aren't equally effective or productive ways of accomplishing the same tasks; we've just found the most success using the tools in this text.

Open-source tools

All of the tools in this book are available completely free. The reason we recommend using open-source software is the communities that you'll become a part of when you start adopting them. The other purpose is because we believe these are the best options for researching a computer. The [‘four freedoms’ of open source software](#) outlined below capture the reason and ethics behind our choice.

- **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.

The computer science in science

Computers, code, and the internet have become pretty standard in modern professional work, especially if that work involves research. Just about every field of science now has a ‘computational’ area or journal to accompany it. [Archaeologists](#) use computers to study geographical information systems (GIS) data and simulate human behavior. [Chemists](#) use data and simulation to determine the arrangements and features of molecules and particles, or to estimate binding affinities for drug molecules on a given receptor or target. [Biologists](#) use computers to build models and simulate biological, ecological, behavioral, and social systems. The list goes on and on...

- [Economics](#)
- [History](#)
- [Finance](#)
- [Linguistics](#)
- [Law](#)

- [Sociology](#)

I suspect most of the people in these fields probably didn't enter them thinking they'd be writing code or designing algorithms, but widespread adoption of computation throughout science is a sign of its near-universal utility. To fully realize the potential of what a computer can do, you'll have dig a little deeper into how they work (and how we interact with them).

Doing research, sharing your work online, and creating/building tools that highlight your work and abilities will require you to know more about how computers work than the average person.

Graphical user interfaces (the gooey)

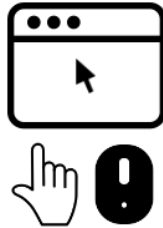
Most people interact with their computers using a [graphical user interface](#) or GUI (pronounced 'gooey'). GUI's are quick and easy to learn because the operating system or software application environment usually mimics an actual physical space (i.e., desktops, folders, documents). If a new task is needed, an additional software application gets installed in this virtual environment to perform that specific function. Below is a list of standard computer tasks, and the associated software GUIs (point-and-click):

- Browsing the internet: Chrome, Safari, and Firefox
- Word processing (articles & reports): Microsoft Word, Apple iWork Papers, and Google Docs
- Composing emails: Microsoft Outlook or Apple Mail

- Building presentations: Microsoft PowerPoint or Apple iWork Keynote
- Creating spreadsheets for numerical calculations to organize data: Microsoft Excel, Apple iWork Numbers, or Google Sheets



In a command line interface, lines of text are used to communicate intentions to the computers



In a graphical user interfaces (GUI), a mouse, trackpad, or finger is used to communicate intentions to the computers.

Users interact with a GUI using a mouse, trackpad, or touchscreen. These devices serve as digital appendages for transmitting intentions to their computers, whether this means opening an application by clicking on it, deleting a file by dragging it into a virtual trash bin, pinching fingers together to zoom in on an image, etc.

Having a [user-centered design](#) has made software applications (and other technologies) available to a broader range of people, and reduced many of the frustrating experiences many of us had in the early days of computing.

But all the benefits of GUIs come with a cost. Creating applications and operating systems that encourage clicking around until users can figure out they work sounds harmless, but it also presents challenges.

For example, it's hard to keep track of everywhere a user clicks (or the order of

things they clicked on) in a GUI, which makes it hard for automation. Furthermore, most GUIs come with a limited collection of possible operations a user can choose from (all of which were selected by the designer of the software).

Command line interfaces (shells, terminals, and text)

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.

Don't worry– we're not going to advise you start only interacting with your computer via the command line. There are plenty of tasks that are better suited for a GUI (*imagine how fun it would be if you had to play angry birds on a command line*). But as someone who'll be using a computer to document and communicate their research, you do need to understand the technologies that are used to store, manipulate, and analyze data.

We also need to make a distinction between the command line, GUIs, and writing code. **We are recommending you write code.** We've noticed that clarity in writing brings clarity in thought, and code is a form of expression.

Hadley Wickham made this point in an excellent talk aptly titled, “[You can’t do data science in a GUI](#)”

“The gooey is the easiest type of approach where you point and click, and everything is laid out in front of you. All of the options are laid out in front of you, which is great because you can see everything you can do. But it’s also terrible because you have constraints—you can only do what the inventors of (SAS or Excel) wanted. Whereas with R—or other programming languages—is the opposite. All you get is this blinking cursor, and it’s just telling you can do literally anything, but it’s not gonna give you much...”

“So I think an important thing about programming languages—like R or Python—is they give you a language to express your ideas, they give you very few constraints, which makes life tough for your learning or doing data science things occasionally, but the payoff for investing in a programming language is you get this whole this new language, and what you can express with them.

You should write code because it makes you think explicitly about what you want to do with your computer. Writing out instructions for how to use a GUI is possible, but it amounts to a bunch of pictures with text saying “*click here, then click here*”. Learning to code well is like learning to write well—the better you get, the more clear your intentions become to *both* your computer and everyone else reading your code.

Use R and RStudio

What is R?

[R](#) is a free statistical modeling software application and language.

What is RStudio?

[RStudio](#) is an integrated development environment (IDE) for using R. If you can't install RStudio on your computer, you can also use [RStudio.cloud](#).

RStudio is a free and open-source [integrated development environment](#) (IDE) for R. You should explore different IDE's on your own– you'll see there are many options, both paid and unpaid.

These applications typically come with a code editor (with syntax highlighting), a graphical/drag-and-drop tools, and some debugging display. Other examples of IDEs are [DataGrip](#) for relational data, [Spyder IDE](#) for Python, or [Stata](#). *These are not free.*

Why should I use R/RStudio and not Python?

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

We recommend R/RStudio for three reasons: First, this book is written for people doing research in graduate school. Most graduate students have a research question, or general curiosity, that they will turn into a data set that needs to be analyzed. Thus, the entry point for almost all graduate students into data science is *that they have data they need to analyze*, and this is exactly what R was built to do.

The second reason we recommend R/RStudio is the time saved by switching between software applications. For example, when I was in graduate school, I had to have *a minimum of five applications open* to do my daily work of data analysis (MS Word to write, MS Excel to create tables, Stata for analysis, the browser for internet research, and Adobe for reading .PDFs). Five different GUIs, each with their own design characteristics, and each costing me valuable neurons every time I had to switch between them (read more in the footnotes). With R/RStudio, this number is cut to two (RStudio and the browser).

‘The only factor becoming scarce in a world of abundance is human attention’ – Kevin Kelly in Wired

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

“They’re 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 each of these examples our effective intelligence is amplified, but not in the way of complementary artifacts. In the case of competitive artifacts, when we are deprived of their use, we are no better than when we started. They’re not coaches and teachers—they are serfs.”

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

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

Getting R/RStudio set up

1. Download and install R from [CRAN](#)
2. Download and install [RStudio](#), the integrated development environment (IDE) for R

3. An alternative to downloading and installing 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](#)

You'll also find a massive network of support on [Stackoverflow](#), [RStudio Community](#), and [Google Groups](#).

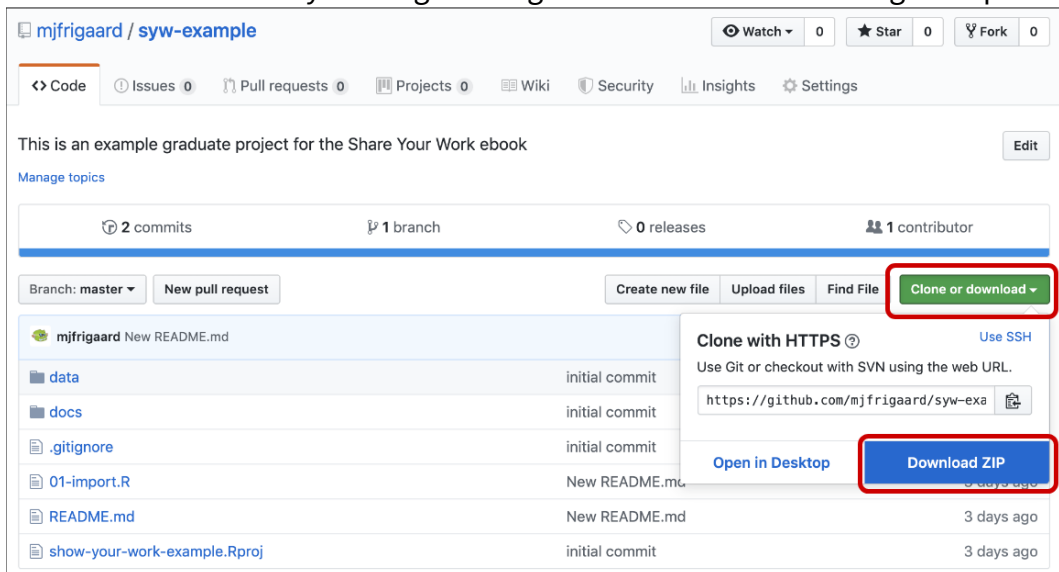
::FOOTNOTES::

- The Ford Foundation report, "[Roads and Bridges](#)", outlines some other reason you should be using open source software.
- Read this articles on multitasking (then try to stop doing it).
 - [Why is it so hard to do my work? The challenge of attention residue when switching between work tasks - ScienceDirect](#)
 - [Information, Attention, and Decision Making](#)
 - [Causes, effects, and practicalities of everyday multitasking](#)

4. Part 3: An example project

To help guide you through learning these technologies, we've made a code repository of some files typically found in a research project. The files in this repository were used to create this [master's thesis](#) and [this peer-reviewed publication](#).

Download these files by clicking on the green icon and downloading the zip file.



The screenshot shows the GitHub interface for the repository 'mjfrigaard / syw-example'. At the top, there are buttons for 'Watch', 'Star', and 'Fork', each with a count of 0. Below these are tabs for 'Code', 'Issues', 'Pull requests', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. The main content area displays the repository description: 'This is an example graduate project for the Share Your Work ebook'. Below this, there are statistics: '2 commits', '1 branch', '0 releases', and '1 contributor'. A 'Clone or download' button is highlighted with a red box. A dropdown menu is open from this button, showing options to 'Clone with HTTPS' or 'Use SSH'. The 'Download ZIP' button is also highlighted with a red box. The repository structure is listed below, showing files like 'data', 'docs', '.gitignore', '01-import.R', 'README.md', and 'show-your-work-example.Rproj'.

Put the zipped file in a recognizable place (like the Documents folder or on your Desktop). Unzip the folder and examine its contents. We'll be using these files throughout the rest of the text.

If you can't download these files onto your computer, you can use [RStudio.Cloud](#) (which we will cover in the next sections).

The next few sessions cover some background on common operating systems, jargon, and some handy commands.

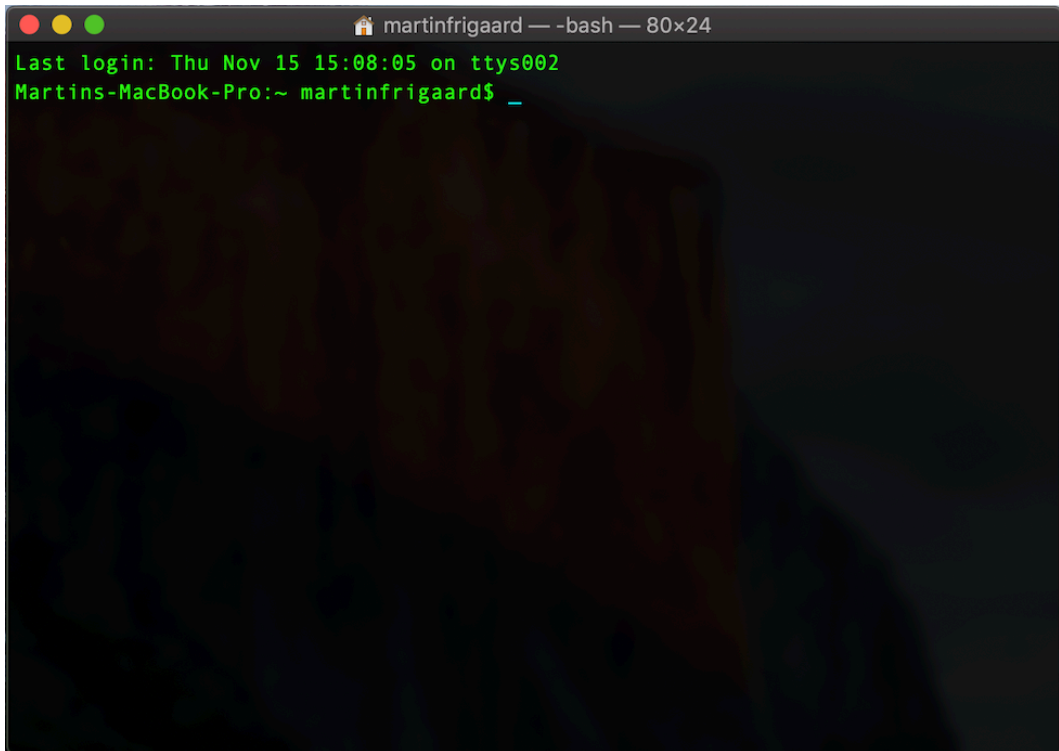
Operating systems

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). You can use most Linux commands on a Mac.

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](#).

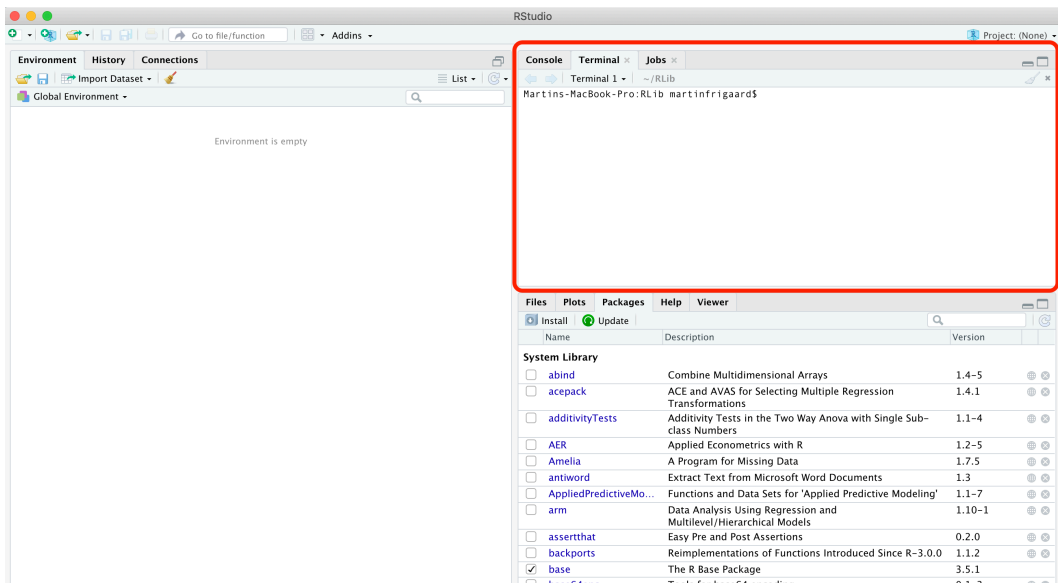
What Terminal looks like

Below is an image of what the terminal application looks like on macOS with Homebrew syntax highlighting.



The Terminal is a command line interface application for Mac users. Terminal is available as an application (on Mac go to *Applications > Utilities > Terminal*) or as a pane in RStudio.

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



Some common Terminal commands

FAIR WARNING—command line interfaces can be frustrating. Most of the technologies we interact with daily 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 you'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!!**) perform commands as **root** user

Terminal emulators and shells

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.

On Macs, the Terminal application runs a [bash shell](#). This is the most commonly used shell, but there are other options too (see [Zsh](#), [tcsh](#), and [sh](#)). *in fact, bash is a pun for Bourne-again shell.*

Why am I learning the command line?

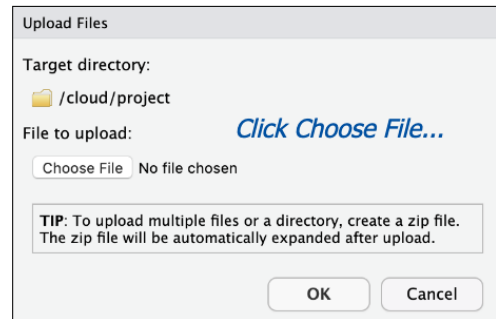
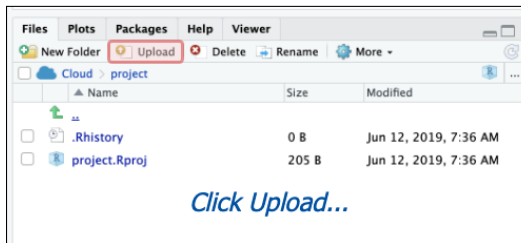
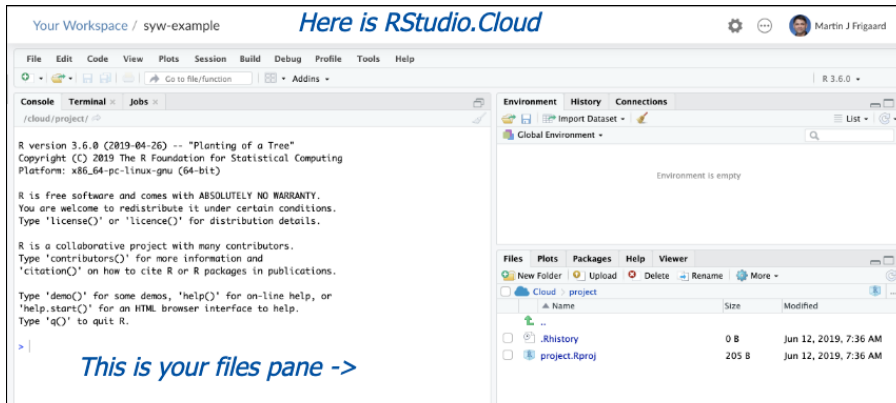
The key features that make the Unix programs so powerful are 1) specificity, and 2) modularity.

- **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.

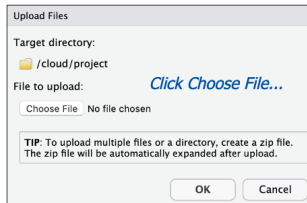
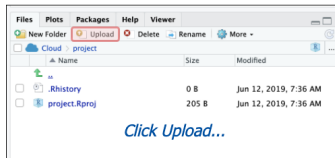
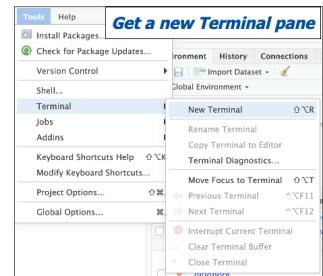
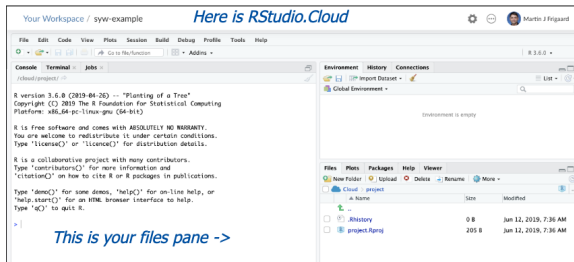
Task 1: Listing your files

The files in the **example project folder** that you downloaded in chapter 2. After unzipping these files, open the Terminal pane in RStudio.

If you’re using RStudio.Cloud, you will need to create a New Project, and upload the `syw-example-master.zip` file.



We are going to use the Terminal pane in RStudio to explore the contents of this folder, starting with `ls` to list the files.



\$ ls

project.Rproj syw-example-master

This shows the RStudio project file, and the folder we uploaded. We actually want to bring these files into the **root folder**. The root folder is the **parent folder** to the syw-example-master folder.

The 'project' folder is the parent folder to the 'syw-example-master' folder



We are going to copy the files into the root folder from the syw-example-master using the following commands.

```
cp -a /syw-example-master/. ./
```

Its not necessary that you totally understand what these commands are doing, but be sure to *type them into the Terminal*.

```
$ cp -a syw-example-master/. .
```

Now we can check for the files again with `ls`

```
$ ls
README.md docs      show-your-work-example.Rproj syw-example-master
data      project.Rproj src
```

Ok, but we don't need the old folder, `syw-example-master`, so we will remove it with `rm` and combine it with two flags `i` and `R`.

Type the following into the Terminal to learn more about the `rm` command.

Getting help

```
$ info rm
```

If we scroll down we learn the following about the `i` flag.

- i Request confirmation before attempting to remove each file, regardless of the file's permissions, or whether or not the standard input device is a terminal. The `-i` option overrides any previous `-f` options.

What does the `R` do?

After you've answered that question, type the following into the terminal pane and hit return/enter.

```
$ rm -iR syw-example-master
```

The terminal is going to ask you if you want to delete each file. This can be tedious, but it's better than deleting everything before reviewing the files.

First you'll be asked if you want to descend into directory 'syw-example-master?', and we do, so we type *y*. Then we are asked if we want to head into the `docs` folder (we do), then we get asked if we want to remove regular file in the `docs` folder, `2012-10-62-ican-manuscript-revision-v02.docx`, and we do so we enter *y* and hit enter or return. After we've deleted all of the files, we can check the files in the root folder using `ls` again.

```
$ ls
README.md data docs project.Rproj src syw-example.Rproj
```

Hm, this looks like a list of the files and folders, but not the files *in* the folders. Is there a way to get a nice [folder tree](#) that shows the entire project?

The bash shell on macOS comes with a whole host of packages you 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)

Installing packages in Terminal

You can install the `tree` package with homebrew, then enter the following commands to get the `tree` package.

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

This gives us the following output:

```
.
├── README.md
├── data
│   └── IcanBP.csv
├── docs
│   └── 2012-10-62-ican-manuscript-revision-v02.docx
├── src
│   ├── 01-import.R
│   └── 02-wrangle.R
└── syw-example.Rproj
```

Combining tools with pipes

We can imagine a situation where an output like this would be helpful, but it would be great if we could store it somewhere with these project files. It's probably never a bad idea to store the original folder contents somewhere as a backup.

In order to accomplish this, we will 'pipe' the output from `tree` into a plain text file and call it `syw-folder-backup`. We will also include today's date.

```
$ tree > syw-folder-backup-$(date +%Y-%m-%d).txt
```

Now we can view the contents of this file using the `cat` command.

```
$ cat syw-folder-backup-2019-06-12.txt
.  
  README.md  
  data  
    IcanBP.csv  
  docs  
    2012-10-62-ican-manuscript-revision-v02.docx  
  src  
    01-import.R  
    02-wrangle.R  
  syw-example.Rproj  
  syw-folder-backup-2019-06-12.txt
```

This is a small taste of how these commands can be combined to create very efficient workflows and procedures. I can tether commands together, and move inputs and outputs around with a lot of flexibility (and a little reading).

Getting more help

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

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

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

Tracing your steps

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

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

Version control vs. track changes

13	Multiple logistic regression models were used to assess the strength of	
14	BMI-for-age and overall EBP risk (any systolic or diastolic readings greater than	- We can see the proposed changes, what they are, and who made them.
15	age, gender and height or an absolute value equal to or greater than 120/80 mmHg	- This revision history can track changes in a single document, but what about all the files used to create this single statement?
16	included sex, age and race/ethnicity (Table 3). Obesity significantly predicted	
17	1.43-18.66) and diastolic EBP (OR, 8.24; 2.71-25.05) risk. Overweight status significantly predicted	
18	diastolic EBP (OR 3.96; 1.24-12.60) risk. Obese students were 8.16 times more likely to present with a	
19	BP reading that was $\geq 90^{\text{th}}$ percentile ($P < 0.01$) compared to normal BMI category students. <u>When BMI</u>	
20	<u>status was dichotomized (underweight/normal weight vs. overweight/obese), the overweight/obese</u>	
21	<u>students were 3.70 times more likely to have a BP reading $\geq 90^{\text{th}}$ percentile ($P < 0.05$) compared to</u>	
22	<u>normal BMI category students (data not shown).</u> Age, sex, and race/ethnicity were not significant	Frigaard, Martin Deleted:
23	predictors of overall EBP risk.	
24	Discussion	We know any changes to this section means a lot of other files have to change, but how can we know which files (and what changes) just by looking at this document?

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

"Keep Knowledge in Plain Text"

In the classic text [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 track of your changes, notes, and any pertinent documentation about your project in plain text README files. The reasons for this will

become more apparent as we move through the example, but I wanted to outline a few here:

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

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

What *isn’t* plain text

Non-plain text files are usually binary files (the 0/1 kind of binary). Binary files (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.

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

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.

As Prof. Kieran Healy acknowledges the benefits (and downsides) of having collaborators working in binary file formats in his [text](#), “The Plain Person’s Guide to Plain Text Social Science,”

“...it is generally easier for you to use their software than vice versa, if only because you are likely to have a copy of Word on your computer already. In these circumstances you might also collaborate using Google Docs or some other service that allows for simultaneously editing the master copy of a document. This may not be ideal, but it

is better than not collaborating. There is little to be gained from plain-text dogmatism in a .docx world.”

The problem with some of these technologies is they don’t scale (imagine 100+ collaborators), and they come with a task-switching cost (see attentional residue from chapter 2).

Using markdown & Rmarkdown

A common type of plain text file is a markdown file, or .md file. Markdown has a straightforward syntax that is easy for both humans and computers to read, and it allows for some formatting options to aid with communication. I recommend reading up on R and RMarkdown because of how many different outputs this combination can be used to produce (see [Markdown Syntax Documentation](#) on John Gruber’s site, and [R Markdown: The Definitive Guide](#) for more information).

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.

Plain text + Git

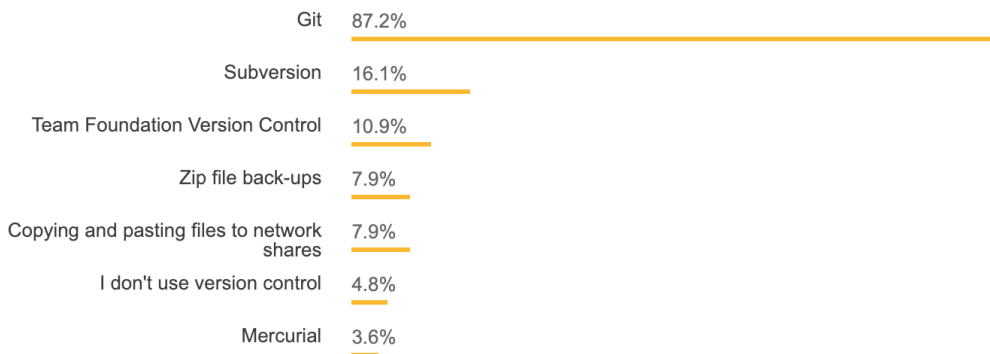
Git prefers plain text files because, until recently, software engineers and app developers were using programs like Git to track their source code (which they write in plain text)–another reason we recommend keeping your documentation and notes in a plain text file.

Why use Git?

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

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

Git has become the most common version control system used by [programmers](#).



74,298 responses; select all that apply

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

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 'commits,' 'pushes,' 'forks,' and 'repos.'

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

Installing Git

1. Download and install [Git](#).
2. Create a [Github](#) account.

Configuring Git with `git config`

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

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

```
git config --system
```

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

```
git config --global
```

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

```
git config
```

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

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

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

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

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

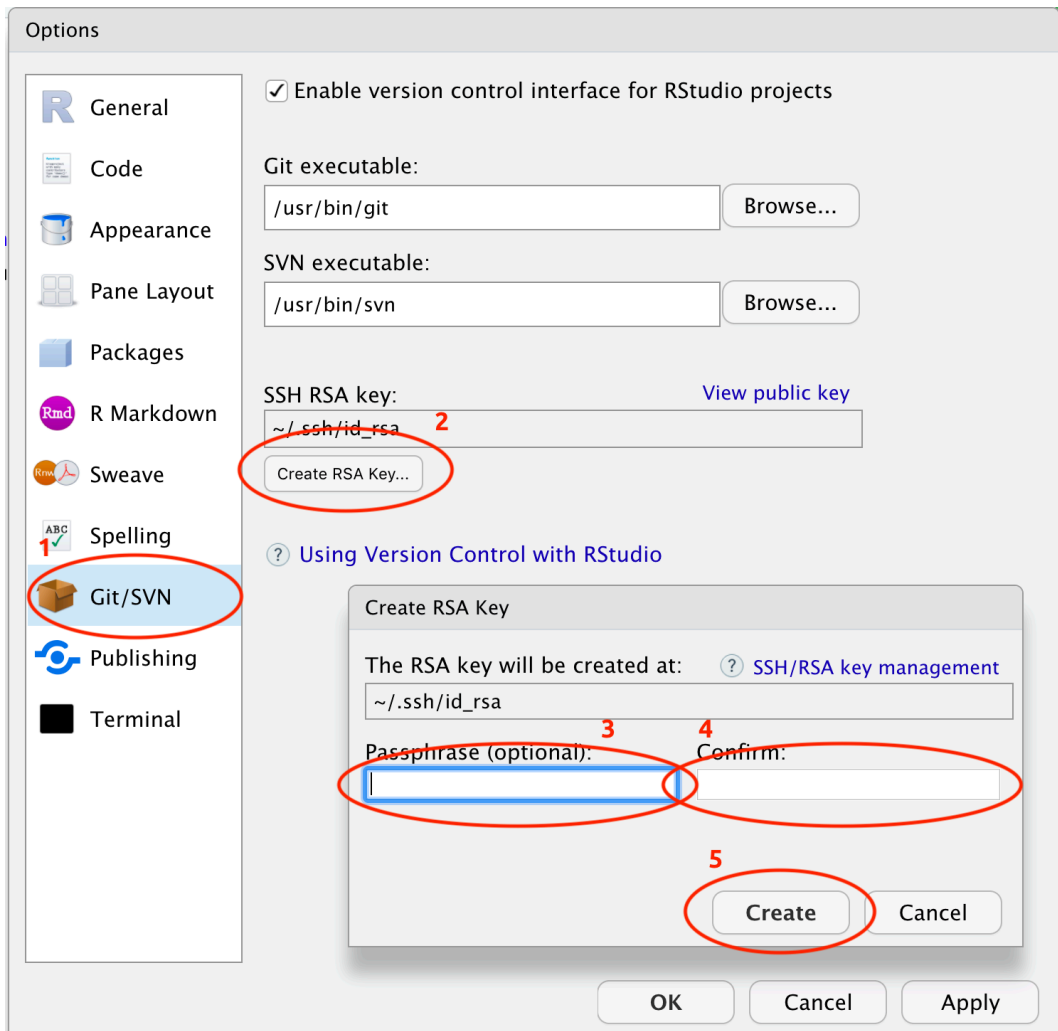
```
user.name=Martin Frigaard  
user.email=mjfrigaard@gmail.com
```

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

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

Synchronize RStudio and Git/Github

[Jenny Bryan](#) 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. I echo a lot of this information below (with copious screenshots).



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

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

The result should look something like this:

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

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

Adding a key SSH in Terminal

In the Terminal, I enter the following commands.

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

The response tells me I'm an Agent.

Agent pid 007

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

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

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

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

Enter passphrase **for** /Users/username/.ssh/id_rsa:

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

Create the `~/.ssh/config` file

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

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

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

Then I create this config file with `touch`

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

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

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

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

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

Finally, I can check config with cat

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

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

6. Conclusion

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