Christopher Gandrud

Reproducible Research with R and RStudio (Third Edition)

To my wife,

who is currently at the movies with our son, so that I can finish the third edition.

Contents

List of Tables

List of Figures

Preface

My motivation

This book has its genesis in my PhD research at the London School of Economics. I started the degree with questions about the 2008/09 financial crisis and planned to spend most of my time researching capital adequacy requirements. But I quickly realized that I would actually spend a large proportion of my time learning the day-to-day tasks of data gathering, analysis, and results presentation. After plodding through for a while with Word, Excel, and Stata, my breaking point came while reentering results into a regression table after I had tweaked one of my statistical models, yet again. Surely there was a better way to *do* research that would allow me to spend more time answering my research questions. Making research reproducible for others also means making it better organized and efficient for yourself. My search for a better way led me straight to the tools for reproducible computational research.

The reproducible research community is very active, knowledgeable, and helpful. Nonetheless, I often encountered holes in this collective knowledge, or at least had no resource organize it all together as a whole. That is my intention for this book: to bring together the skills I have picked up for actually doing and presenting computational research. Hopefully, the book, along with making reproducible research more widely used, will save researchers hours of googling, so they can spend more time addressing their research questions.

Changes to the Third Edition

To WRITE

- The book is created using *bookdown* (**?**), a format that builds on *rmarkdown* to compile books into many different formats.

Changes to the Second Edition

The tools of reproducible research have developed rapidly since the first edition of this book was published just two years ago. The second edition has been updated to incorporate the most important of these advancements, including discussions of:

- The *rmarkdown* package, which allows you to create reproducible research documents in PDF, HTML, and Microsoft Word formats using the simple and intuitive Markdown syntax.
- Improvements and changes to RStudio's interface and capabilities, such as its new tools for handling R Markdown documents.
- Expanded *knitr* R code chunk capabilities.
- The `kable()` function in the *knitr* package and the *texreg* package for dynamically creating tables to present your data and statistical results.
- An improved discussion of file organization allowing you to take full advantage of relative file paths so that your documents are more easily reproducible across computers and systems.
- The *dplyr*, *magrittr*, and *tidyr* packages for fast data manipulation.
- Numerous changes to R syntax in user-created packages.
- Changes to GitHub's and Dropbox's interfaces.

Acknowledgments

I would not have been able to write this book without many people's advice and support. Foremost is John Kimmel, acquisitions editor at Chapman and Hall. He approached me in Spring 2012 with the general idea and opportunity for this book. Other editors at Chapman and Hall and Taylor and Francis have greatly contributed to this project, including Marcus Fontaine. I would also like to thank all of the book's reviewers whose helpful comments have greatly improved it. The first edition's reviewers include:

- Jeromy Anglim, Deakin University
- Karl Broman, University of Wisconsin, Madison
- Jake Bowers, University of Illinois, Urbana-Champaign
- Corey Chivers, McGill University

- Mark M. Fredrickson, University of Illinois, Urbana-Champaign
- Benjamin Lauderdale, London School of Economics
- Ramnath Vaidyanathan, McGill University

and there have been many other annonymous reviewers who have provided great feedback over the years.

The developer and blogging community has also been incredibly important for making this book possible. Foremost among these people is Yihui Xie. He is the main developer behind the *knitr* package, co-developer of *rmarkdown*, and also an avid blog writer and commenter. Without him the ability to do reproducible research would be much harder and the blogging community that spreads knowledge about how to do these things would be poorer. Other great contributors to the reproducible research community include Carl Boettiger, Karl Broman, Markus Gesmann (who developed *googleVis*), Rob Hyndman, and Hadley Wickham (who has developed numerous very useful R packages). Thank you also to Victoria Stodden and Michael Malecki for helpful suggestions. And, of course, thank you to everyone at RStudio (especially JJ Allaire) for creating an increasingly useful program for reproducible research.

The second edition has benefited immensely from first edition readers' comments and suggestions. For a list of their valuable contributions, please see the book's GitHub Issues page https://GitHub.com/christophergandrud/Rep-Res-Book/issues and the first edition's Errata page http://christophergandrud.GitHub.io/RepResR-RStudio/errata.htm.

My students at Yonsei University were an important part of making the first edition. One of the reasons that I got interested in using many of the tools covered in this book, like using **knitr} in slideshows, was to improve a course I taught there: Introduction to Social Science Data Analysis. I tested many of the explanations and examples in this book on my students. Their feedback has been very helpful for making the book clearer and more useful. Their experience with using these tools on Microsoft Windows computers was also important for improving the book's Windows documentation. Similarly, my students at the Hertie School of Governance inspired and tested key sections of the second edition.

The vibrant community at Stack Overflow http://stackoverflow.com/ and Stack Exchange http://stackexchange.com/ are always very helpful for finding answers to problems that plague any computational researcher. Importantly, the sites make it easy for others to find the answers to questions that have already been asked.

My wife, Kristina Gandrud, has been immensely supportive and patient with me throughout the writing of this book (and pretty much my entire academic career). Certainly this is not the proper forum for musing about marital relations, but I'll do a musing anyways. Having a person who supports your

interests, even if they don't completely share them, is immensely helpful for a researcher. It keeps you going.

Stylistic Convensions

I use the following conventions throughout this book:

- **Abstract variables**: Abstract variables, i.e. variables that do not represent specific objects in an example, are in `ALL CAPS TYPEWRITER TEXT`.
- **Clickable buttons**: Clickable Buttons are in `typewriter text`.
- **Code**: All code is in `typewriter text`.
- **Filenames and directories**: Filenames and directories more generally are printed in *italics*. I use CamelBack for file and directory names.
- **File extensions**: Like filenames, file extensions are *italicized*.
- **Individual variable values**: Individual variable values mentioned in the text are in *italics*.
- **Objects**: Objects are printed in *italics*. I use CamelBack for object names.
- **Object columns**: Data frame object columns are printed in *italics*.
- **Function names** are followed by parentheses (e.g., `stats::lm()`)
- **Packages**: **R** packages are printed in *italics*.
- **Windows and RStudio panes**: Open windows and RStudio panes are written in *italics*.
- **Variable names**: Variable names are printed in **bold**. I use CamelBack for individual variable names.

Additional Resources

Additional resources that supplement the examples in this book can be freely downloaded and experimented with. These resources include longer examples discussed in individual chapters and a complete short reproducible research project.

Chapter Examples

Longer examples discussed in individual chapters, including files to dynamically download data, code for creating figures, and markup files for creating presentation documents, can be accessed at: https://github.com/christophergandrud/Rep-Res-Examples. Please see Chapter **??** for more information on downloading files from GitHub, where the examples are stored.

Short Example Project

To download a full (though very short) example of a reproducible research project created using the tools covered in this book go to: https://github.com/christophergandrud/Rep-Res-ExampleProject1. Please follow the replication instructions in the main *README.md* file to fully replicate the project. It is probably a good idea to hold off looking at this complete example in detail until after you have become acquainted with the individual tools it uses. Become acquainted with the tools by reading through this book and working with the individual chapter examples.

The following two figures give you a sense of how the example's files are organized. Figure **??** shows how the files are organized in the file system. Figure **??** illustrates how the main files are dynamically tied together. In the *Data* directory we have files to gather raw data from the (**?**) on fertilizer consumption and from (**?**) on countries' levels of democracy. They are tied to the

data through the `WDI()` and `download.file()` functions. A *Makefile* can run *Gather1.R* and *Gather2.R* to gather and clean the data. It runs *MergeData.R* to merge the data into one data file called *MainData.csv*. It also automatically generates a variable description file and a *README.md* recording the session info.

The *Analysis* folder contains two files that create figures presenting this data. They are tied to *MainData.csv* with the `read.csv()` function. These files are run by the presentation documents when they are knitted. The presentation documents tie to the analysis documents with *knitr* and the `source()` function.

Though a simple example, hopefully these files will give you a complete sense of how a reproducible research project can be organized. Please feel free to experiment with different ways of organizing the files and tying them together to make your research really reproducible.

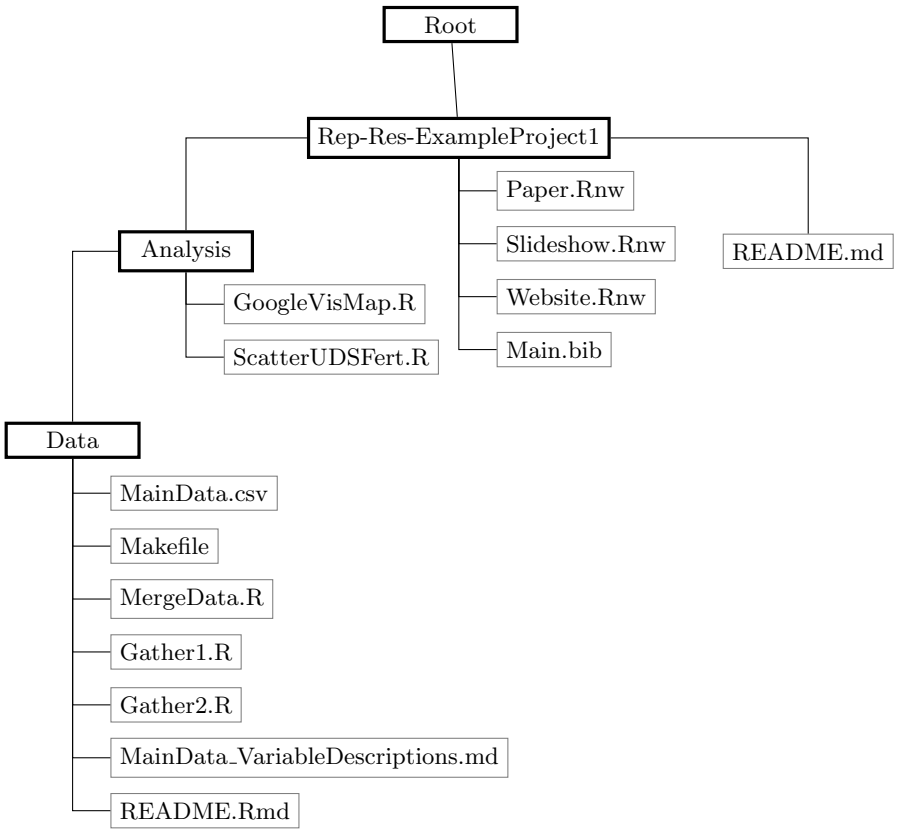

FIGURE 1: Short Example Project File Tree

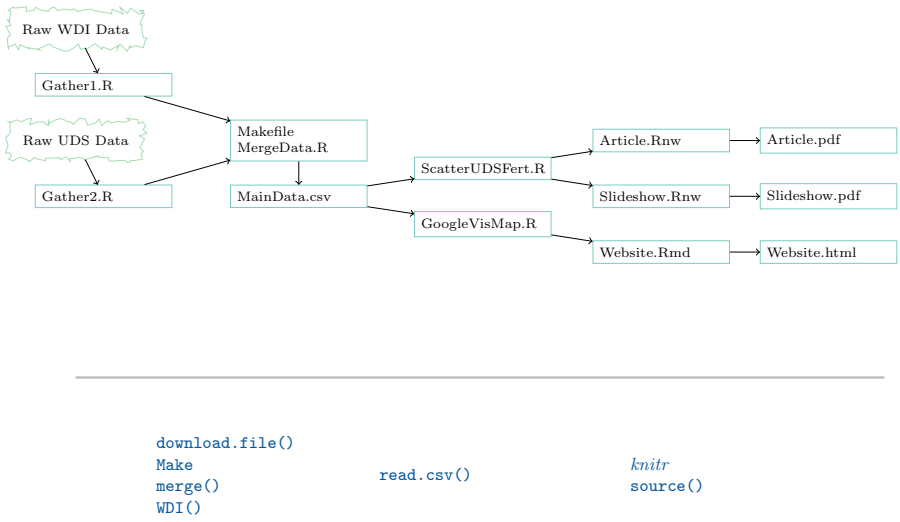

FIGURE 2: Short Example Main File Ties

Updates

Many of the reproducible research tools discussed in this book are improving rapidly. Because of this, I will regularly post updates to the content covered in the book at: https://github.com/christophergandrud/Rep-Res-Book.

Corrections

If you notice any corrections that should be made to fix typos, broken URLs, and so on, you can report them at: https://github.com/christophergandrud/Rep-Res-Book/issues. I'll post notifications of changes to an Errata page at: http://christophergandrud.GitHub.io/RepResR-RStudio/errata.htm.

Part I

Getting Started

1

Introducing Reproducible Research

Research is often presented in very selective containers: slideshows, journal articles, books, or maybe even websites. These presentation documents announce a project's findings and try to convince us that the results are correct (**?**). It's important to remember that these documents are not the research. Especially in the computational and statistical sciences, these documents are the "advertising". The research is the "full software environment, code, and data that produced the results" (**??**, 385). When we separate the research from its advertisement we are making it difficult for others to verify the findings by reproducing them.

This book gives you the tools to dynamically combine your research with the presentation of your findings. The first tool is a workflow for reproducible research that weaves the principles of reproducibility throughout your entire research project, from data gathering to the statistical analysis, and the presentation of results. You will also learn how to use a number of computer tools that make this workflow possible. These tools include:

- the **R** statistical language that will allow you to gather data and analyze it;
- the **LaTeX** and **Markdown** markup languages that you can use to create documents–slideshows, articles, books, and webpages–for presenting your findings;
- the *knitr* and *rmarkdown* **packages** for R and other tools, including **command-line programs** like GNU Make and Git version control, for dynamically tying your data gathering, analysis, and presentation documents together so that they can be easily reproduced;
- **RStudio**, a program that brings all of these tools together in one place.

1.1 What Is Reproducible Research?

Though there is some debate over what are the necessary and sufficient conditions for a replication (**?**, 2), research results are generally considered[1] *replicable* if there is sufficient information available for independent researchers to make the same findings using the same procedures with new data.[2] For research that relies on experiments, this can mean a researcher not involved in the original research being able to rerun the experiment, including sampling, and validate that the new results are comparable to the original ones. In computational and quantitative empirical sciences, results are replicable if independent researchers can recreate findings by following the procedures originally used to gather the data and run the computer code. Of course, it is sometimes difficult to replicate the original data set because of issues such as limited resources to gather new data or because the original study already sampled the full universe of cases. So as a next-best standard we can aim for "*really reproducible research*" (**?**, 1226).[3] In computational sciences[4] this means:

> the data and code used to make a finding are available and they are sufficient for an independent researcher to recreate the finding.

In practice, research needs to be *easy* for independent researchers to reproduce (**?**). If a study is difficult to reproduce it's more likely that no one will reproduce it. If someone does attempt to reproduce this research, it will be difficult for them to tell if any errors they find were in the original research or problems they introduced during the reproduction. In this book you will learn how to avoid these problems.

[1] **?**, 3-4 note that some disciplines, e.g. computing machinery and meteorology, give "replicable" and "reproducible" the exact opposite meanings from they way they are used in this book and many other disciplines such as biology, economics, and epidemiology.

[2] This is close to what (**?**) calls "operational replication".

[3] The really reproducible computational research originates in the 1980s and early 1990s with Jon Claerbout and the Stanford Exploration Project (**??**). Further seminal advances were made by Jonathan B. Buckheit and David L. Donoho who created the Wavelab library of MATLAB routines for their research on wavelets in the mid-1990s (**?**).

[4] Reproducibility is important for both quantitative and qualitative research (**?**). Nonetheless, we will focus mainly on on methods for reproducibility in quantitative computational research.

In particular you will learn tools for dynamically "*knitting*"[5] the data and the source code together with your presentation documents. Combined with well-organized source files and clearly and completely commented code, independent researchers will be able to understand how you obtained your results. This will make your computational research easily reproducible.

1.2 Why Should Research Be Reproducible?

Reproducible research is one of the main components of science. If that's not enough reason for you to make your research reproducible, consider that the tools of reproducible research also have direct benefits for you as a researcher.

1.2.1 For science

Replicability has been a key part of scientific inquiry from perhaps the 1200s (**??**). It has even been called the "demarcation between science and non-science" (**?**, 2). Why is replication so important for scientific inquiry?

Standard to judge scientific claims

Replication opens claims to scrutiny, allowing us to keep what works and discard what doesn't. Science, according to the American Physical Society, "is the systematic enterprise of gathering knowledge …organizing and condensing that knowledge into testable laws and theories". The "ultimate standard" for evaluating scientific claims is whether or not the claims can be replicated (**??**). Research findings cannot even really be considered "genuine contributions to human knowledge" until they have been verified through replication (**?**, 38). Replication "requires the complete and open exchange of data, procedures, and materials". Scientific conclusions that are not replicable should be abandoned or modified "when confronted with more complete or reliable …evidence".[6]

[5] Much of the reproducible computational research and literate programming literatures have traditionally used the term "weave" to describe the process of combining source code and presentation documents (see **?**, 101). In the R community weave is usually used to describe the combination of source code and LaTeX documents. The term "knit" reflects the vocabulary of the *knitr* R package (knit + R). It is used more generally to describe weaving with a variety of markup languages. The term is used by RStudio if you are using the *rmarkdown* package, which is similar to *knitr*. We also cover the *rmarkdown* package in this book. Because of this, I use the term knit rather than weave in this book.

[6] See the American Physical Society's website at http://www.aps.org/policy/statements/99_6.cfm. See also (**?**).

Reproducibility enhances replicability. If other researchers are able to clearly understand how a finding was originally made, then they will be better able to conduct comparable research in meaningful attempts to replicate the original findings. Sometimes strict replicability is not feasible, for example, when it is only possible to gather one data set on a population of interest. In these cases reproducibility is a "minimum standard" for judging scientific claims (**?**).

It is important to note that though reproducibility is a minimum standard for judging scientific claims, "a study can be reproducible and still be wrong" (**?**). For example, a statistically significant finding in one study may remain statistically significant when reproduced using the original data/code, but when researchers try to replicate it using new data and even methods, they are unable to find a similar result. The original finding could simply have been noise, even though it is fully reproducible.

Avoiding effort duplication & encouraging cumulative knowledge development

Not only is reproducibility important for evaluating scientific claims, it can also contribute to the cumulative growth of scientific knowledge (**??**). Reproducible research cuts down on the amount of time scientists have to spend gathering data or developing procedures that have already been collected or figured out. Because researchers do not have to discover on their own things that have already been done, they can more quickly build on established findings and develop new knowledge.

1.2.2 For you

Working to make your research reproducible does require extra upfront effort. For example, you need to put effort into learning the tools of reproducible research by doing things such as reading this book. But beyond the clear benefits for science, why should you make this effort? Using reproducible research tools can make your research process more effective and (hopefully) ultimately easier.

Better work habits

Making a project reproducible from the start encourages you to use better work habits. It can spur you to more effectively plan and organize your research. It should push you to bring your data and source code up to a higher level of quality than you might if you "thought 'no one was looking'" (**?**, 386). This forces you to root out errors–a ubiquitous part of computational

research–earlier in the research process (**?**, 385). Clear documentation also makes it easier to find errors.[7]

Reproducible research needs to be stored so that other researchers can actually access the data and source code. By taking steps to make your research accessible for others you are also making it easier for yourself to find your data and methods when you revise your work or begin new a project. You are avoiding personal effort duplication, allowing you to cumulatively build on your own work more effectively.

Better teamwork

The steps you take to make sure an independent researcher can figure out what you have done also make it easier for your collaborators to understand your work and build on it. This applies not only to current collaborators, but also future collaborators. Bringing new members of a research team up to speed on a cumulatively growing research project is faster if they can easily understand what has been done already (**?**, 386).

Changes are easier

A third person may or may not actually reproduce your research even if you make it easy for them to do so. But, *you will almost certainly reproduce parts or even all of your own research*. No actual research process is completely linear. You almost never gather data, run analyses, and present your results without going backwards to add variables, make changes to your statistical models, create new graphs, alter results tables in light of new findings, and so on. You will probably try to make these changes long after you last worked on the project and long since you remembered the details of how you did it. Whether your changes are because of journal reviewers' and conference participants' comments or you discover that new and better data has been made available since beginning the project, designing your research to be reproducible from the start makes it much easier to change things later on.

Dynamic reproducible documents in particular can make changing things much easier. Changes made to one part of a research project have a way of cascading through the other parts. For example, adding a new variable to a largely completed analysis requires gathering new data and merging it with existing data sets. If you used data imputation or matching methods you may need to rerun these models. You then have to update your main statistical analyses, and recreate the tables and graphs you used to present the results. Adding a new variable essentially forces you to reproduce large portions of

[7] Of course, it's important to keep in mind that reproducibility is "neither necessary nor sufficient to prevent mistakes" (**?**).

your research. If when you started the project you used tools that make it easier for others to reproduce your research, you also made it easier to reproduce the work yourself. You will have taken steps to have a "better relationship with your future self" (**?**, 2).

Higher research impact

Reproducible research is more likely to be useful for other researchers than non-reproducible research. Useful research is cited more frequently (**???**). Research that is fully reproducible contains more information, i.e. more reasons to use and cite it, than presentation documents merely showing findings. Independent researchers may use the reproducible data or code to look at other, often unanticipated, questions. When they use your work for a new purpose they will (should) cite your work. Because of this, Vandewalle et al. even argue that "the goal of reproducible research is to have more impact with our research" (**?**, 1253).

A reason researchers often avoid making their research fully reproducible is that they are afraid other people will use their data and code to compete with them. I'll let Donoho et al. address this one:

> *True. But competition means that strangers will read your papers, try to learn from them, cite them, and try to do even better. If you prefer obscurity, why are you publishing?* (**?**, 16)

1.3 Who Should Read This Book?

This book is intended primarily for researchers who want to use a systematic workflow that encourages reproducibility as well as practical state-of-the-art computational tools to put this workflow into practice. These people include professional researchers, upper-level undergraduate, and graduate students working on computational data-driven projects. Hopefully, editors at academic publishers will also find the book useful for improving their ability to evaluate and edit reproducible research.

The more researchers that use the tools of reproducibility the better. So I include enough information in the book for people who have very limited

experience with these tools, including limited experience with R, LaTeX, and Markdown. They will be able to start incorporating reproducible research tools into their workflow right away. The book will also be helpful for people who already have general experience using technologies such as R and LaTeX, but would like to know how to tie them together for reproducible research.

1.3.1 Academic researchers

Hopefully so far in this chapter I've convinced you that reproducible research has benefits for you as a member of the scientific community and personally as a computational researcher. This book is intended to be a practical guide for how to actually make your research reproducible. Even if you already use tools such as R and LaTeX you may not be leveraging their full potential. This book will teach you useful ways to get the most out of them as part of a reproducible research workflow.

1.3.2 Students

Upper-level undergraduate and graduate students conducting original computational research should make their research reproducible for the same reasons that professional researchers should. Forcing yourself to clearly document the steps you took will also encourage you to think more clearly about what you are doing and reinforce what you are learning. It will hopefully give you a greater appreciation of research accountability and integrity early in your career (**??**, 183).

Even if you don't have extensive experience with computer languages, this book will teach you specific habits and tools that you can use throughout your student research and hopefully your careers. Learning these things earlier will save you considerable time and effort later.

1.3.3 Instructors

When instructors incorporate the tools of reproducible research into their assignments they not only build students' understanding of research best practice, but are also better able to evaluate and provide meaningful feedback on students' work (**?**, 183). This book provides a resource that you can use with students to put reproducibility into practice.

If you are teaching computational courses, you may also benefit from making your lecture material dynamically reproducible. Your slides will be easier to update for the same reasons that it is easier to update research. Making the methods you used to create the material available to students will give them

more information. Clearly documenting how you created lecture material can also pass information on to future instructors.

1.3.4 Editors

Beyond a lack of reproducible research skills among researchers, an impediment to actually creating reproducible research is a lack of infrastructure to publish it (**?**). Hopefully, this book will be useful for editors at academic publishers who want to be better at evaluating reproducible research, editing it, and developing systems to make it more widely available. The journal *Biostatistics* is a good example of a publication that is encouraging (actually requiring) reproducible research. From 2009 the journal has had an editor for reproducibility that ensures replication files are available and that results can be replicated using these files (**?**). The more editors there are with the skills to work with reproducible research the more likely it is that researchers will do it.

1.3.5 Private sector researchers

Researchers in the private sector may or may not want to make their work easily reproducible outside of their organization. However, that does not mean that significant benefits cannot be gained from using the methods of reproducible research.

Even if a company has only one person doing research, they benefit from using reproducible research methods. Just with academic research this person actually does have a collaborator: their future self. As discussed above, reproducible research makes this collaboration easier.

Companies with more than one researcher do (or likely should) act as a research community, even if public reproducibility is ruled out to guard proprietary information.[8] Making your research reproducible to members of your organization can spread valuable information about how analyses were done and data was collected. This will help build your organization's knowledge and avoid effort duplication. Just as a lack of reproducibility hinders the spread of information in the scientific community, it can hinder it inside of a private organization. Using the sort of dynamic automated processes run with clearly documented source code we will learn in this book can also help create robust data analysis methods that help your organization avoid errors that may come from cutting-and-pasting data across spreadsheets.[9]

[8] There are ways to enable some public reproducibility without revealing confidential information. See (**?**) for a discussion of one approach.

[9] See this post by David Smith about how the J.P. Morgan "London Whale" problem may have been prevented with the type of processes covered in this book:

The tools of reproducible research covered in this book enable you to create professional standardized reports that can be easily updated or changed when new information is available. In particular, you will learn how to create batch reports based on quantitative data.

1.4 The Tools of Reproducible Research

This book will teach you the tools you need to make your research highly reproducible. Reproducible research involves two broad sets of tools. The first is a **reproducible research environment** that includes the statistical tools you need to run your analyses as well as "the ability to automatically track the provenance of data, analyses, and results and to package them (or pointers to persistent versions of them) for redistribution". The second set of tools is a **reproducible research publisher**, which prepares dynamic documents for presenting results and is easily linked to the reproducible research environment (**?**, 415).

In this book we will focus on learning how to use the widely available and highly flexible reproducible research environment–R/RStudio (**??**).[10] R/RStudio can be linked to numerous reproducible research publishers such as LaTeX and Markdown with Yihui Xie's *knitr* package (**?**) or the related *rmarkdown* package (**?**). The main tools covered in this book include:

- **R**: a programming language primarily for statistics and graphics. It can also be useful for data gathering and creating presentation documents.
- ***knitr* and *rmarkdown***: related R packages for literate programming. They allow you to combine your statistical analysis and the presentation of the results into one document. They work with R and a number of other languages such as Bash, Python, and Ruby.
- **Markup languages**: instructions for how to format a presentation document. In this book we cover LaTeX, Markdown, and a little HTML.
- **RStudio**: an integrated developer environment (IDE) for R that tightly combines R, *knitr*, *rmarkdown*, and markup languages.
- **Cloud storage & versioning**: Services such as Dropbox and Git/GitHub that can store data, code, and presentation files, save previous versions of these files, and make this information widely available.

http://blog.revolutionanalytics.com/2013/02/did-an-excel-error-bring-down-the-london-whale.html (posted 11 February 2013).

[10] The book was created with R version and developer builds of RStudio version 0.99.370.

- **Unix-like shell programs**: These tools are useful for working with large research projects.[11] They also allow us to use command-line tools including GNU Make for compiling projects and Pandoc, a program useful for converting documents from one markup language to another.

1.4.1 Why Use R, *knitr*/*rmarkdown*, and RStudio for Reproducible Research?

Why R?

Why use a statistical programming language like R for reproducible research? R has a very active development community that is constantly expanding what it is capable of. As we will see in this book, R enables researchers across a wide range of disciplines to gather data and run statistical analyses. Using the *knitr* or *rmarkdown* package, you can connect your R-based analyses to presentation documents created with markup languages such as LaTeX and Markdown. This allows you to dynamically and reproducibly present results in articles, slideshows, and webpages.

The way you interact with R has benefits for reproducible research. In general you interact with R (or any other programming and markup language) by explicitly writing down your steps as source code. This promotes reproducibility more than your typical interactions with Graphical User Interface (GUI) programs like SPSS[12] and Microsoft Word. When you write R code and embed it in presentation documents created using markup languages, you are forced to explicitly state the steps you took to do your research. When you do research by clicking through drop-down menus in GUI programs, your steps are lost, or at least documenting them requires considerable extra effort. Also it is generally more difficult to dynamically embed your analysis in presentation documents created by GUI word processing programs in a way that will be accessible to other researchers both now and in the future. I'll come back to these points in Chapter **??**.

Why knitr and rmarkdown?

Literate programming is a crucial part of reproducible quantitative research.[13] Being able to directly link your analyses, your results, and the code you used

[11] In this book I cover the Bash shell for Linux and Mac as well as Windows PowerShell.

[12] I know you can write scripts in statistical programs like SPSS, but doing so is not encouraged by the program's interface and you often have to learn multiple languages for writing scripts that run analyses, create graphics, and deal with matrices.

[13] Donald Knuth coined the term literate programming in the 1970s to refer to a source file that could be both run by a computer and "woven" with a formatted presentation document (**?**).

to produce the results makes tracing your steps much easier. There are many different literate programming tools for a number of different programming languages.[14] Previously, one of the most common tools for researchers using R and the LaTeX markup language was *Sweave* (**?**). The packages I am going to focus on in this book are newer and have more capabilities. They are called *knitr* and *rmarkdown*. Why are we going to use these tools in this book and not *Sweave* or some other tool?

The simple answer is that they are more capable than *Sweave*. Both *knitr* and *rmarkdown* can work with markup languages other than LaTeX including Markdown and HTML. *rmarkdown* can even output Microsoft Word documents. They can work with programming languages other than R. They highlight R code in presentation documents making it easier for your readers to follow.[15] They give you better control over the inclusion of graphics and can cache code chunks, i.e. save the output for later. *knitr* has the ability to understand *Sweave*-like syntax, so it will be easy to convert backwards to *Sweave* if you want to.[16] You also have the choice to use much simpler and more straightforward syntax with *knitr* and *rmarkdown*.

knitr and *rmarkdown* have broadly similar capabilities and syntax. They both are literate programming tools that can produce presentation documents from multiple markup languages. They have almost identical syntax when used in Markdown. Their main difference is that they take different approaches to creating presentation documents. *knitr* documents must be written using the markup language associated with the desired output. For example, with *knitr*, LaTeX must be used to create PDF output documents and Markdown or HTML must be used to create webpages. *rmarkdown* builds directly on *knitr*, the key difference being that it uses the straightforward Markdown markup language to generate PDF, HTML, and MS Word documents.[17]

Because you write with the simple Markdown syntax, *rmarkdown* is generally easier to use. It has the advantage of being able to take the same markup document and output multiple types of presentation documents. Nonetheless, for complex documents like books and long articles or work that requires custom formatting, *knitr* LaTeX is often preferable and extremely flexible, though the syntax is more complicated.

---

[14] A very interesting tool that is worth taking a look at for the Python programming language is HTML Notebooks created with Jupyter. For more details see `http://jupyter.org/`. We will also discuss these in more detail in Section **??**.

[15] Syntax highlighting uses different colors and fonts to distinguish different types of text.

[16] Note that the Sweave-style syntax is not identical to actual *Sweave* syntax. See Yihui Xie's discussion of the differences between the two at: `http://yihui.name/knitr/demo/sweave/`. *knitr* has a function (`Sweave2knitr`) for converting *Sweave* to *knitr* syntax.

[17] It does this by relying on a tool called Pandoc (**?**).

Why RStudio?

Why use the RStudio integrated development environment for reproducible research? R by itself has the capabilities necessary to gather data, analyze it, and, with a little help from *knitr*/*rmarkdown* and markup languages, present results in a way that is highly reproducible. RStudio allows you to do all of these things, but simplifies many of them and allows you to navigate through them more easily. It also is a happy medium between R's text-based interface and a pure GUI.

Not only does RStudio do many of the things that R can do but more easily, it is also a very good standalone editor for writing documents with LaTeX and Markdown. For LaTeX documents it can, for example, insert frequently used commands like `\section{}` for numbered sections (see Chapter **??**).[18] There are many LaTeX editors available, both open source and paid. But RStudio is currently the best program for creating reproducible LaTeX and Markdown documents. It has full syntax highlighting. Its syntax highlighting can even distinguish between R code and markup commands in the same document. It can spell check LaTeX and Markdown documents. It handles *knitr*/*rmarkdown* code chunks beautifully (see Chapter **??**).

Finally, RStudio not only has tight integration with various markup languages, it also has capabilities for using other tools such as C++, CSS, JavaScript, and a few other programming languages. It is closely integrated with the version control programs Git and SVN. Both of these programs allow you to keep track of the changes you make to your documents (see Chapter **??**). This is important for reproducible research since version control programs can document many of your research steps. It also has a built-in ability to make HTML slideshows from *knitr*/*rmarkdown* documents. Basically, RStudio makes it easy to create and navigate through complex reproducible research documents.

1.5 Installing the main software

Before you read this book you should install the main software. All of the software programs covered in this book are open source and can be easily

[18] If you are more comfortable with a what-you-see-is-what-you-get (WYSIWYG) word processor like Microsoft Word, you might be interested in exploring Lyx. It is a WYSIWYG-like LaTeX editor that works with *knitr*. It doesn't work with the other markup languages covered in this book. For more information see: https://www.lyx.org/. I give some brief information on using Lyx with *knitr* in Chapter 3's Appendix.

downloaded for free. They are available for Windows, Mac, and Linux operating systems. They should run well on most modern computers.

You should install R before installing RStudio. You can download the programs from the following websites:

- **R**: https://www.r-project.org/,
- **RStudio Desktop (Open Source License)**: https://www.rstudio.com/products/rstudio/download/.

The webpages for downloading these programs have comprehensive information on how to install them. Please refer to those pages for more information.

After installing R and RStudio you will probably also want to install a number of user-written packages that are covered in this book. To install all of these user-written packages, please this chapter's Appendix.

1.5.1 Installing markup languages

You will need to install the R package *rmarkdown* (**?**) to turn your markdown documents into polished output that can be presented (e.g. as a website or PDF). To do this in R use:

```
install.packages("rmarkdown")
```

If you plan to render your RMarkdown documents from the console without RStudio you will need to install Pandoc. For instructions see Pandoc's download page: https://pandoc.org/installing.html.

If you want to create LaTeX (PDF) documents you can install a TeX distribution.[19] The simplest way to get all of the LaTeX capabilities you will need for this book is to use the *tinytex* (**?**) R package:

```
install.packages('tinytex')
tinytex::install_tinytex()
```

If you want a full LaTeX distribution see http://www.latex-project.org/ftp.html for installation information.

[19] LaTeX is is really a set of macros for the TeX typesetting system. It is included in all major TeX distributions.

1.5.2 GNU Make

If you are using a Linux computer you already have GNU Make **??** installed.[20] Mac users will need to install the command-line developer tools. There are two ways to do this. One is go to the App Store and download Xcode (it's free). Once Xcode is installed, install command-line tools, which you will find by opening Xcode then clicking on `Preference` `Downloads`. However, Xcode is a very large download and you only need the command-line tools for Make. To install just the command-line tools, open the Terminal and try to run Make by typing `make` and hitting return. A box should appear asking you if you want to install the command-line developer tools. Click `Install`. Windows users will have Make installed if they have already installed *Rtools* (see this Chapter's Appendix). Mac and Windows users will need to install this software not only so that GNU Make runs properly, but also so that other command-line tools work well.

1.5.3 Other Tools

We will discuss other tools such as Git that can be a useful part of a reproducible research workflow. Installation instructions for these tools will be discussed below.

1.6 Book Overview

The purpose of this book is to give you the tools that you will need to do reproducible research with R and RStudio. This book describes a workflow for reproducible research primarily using R and RStudio. It is designed to give you the necessary tools to use this workflow for your own research. It is not designed to be a complete reference for R, RStudio, *knitr*/*rmarkdown*, Git, or any other program that is a part of this workflow. Instead it shows you how these tools can fit together to make your research more reproducible. To get the most out of these individual programs I will along the way point you to other resources that cover these programs in more detail.

To that end, I can recommend a number of resources that cover more of the nitty-gritty:@label(OtherBooks)

[20] To verify this, open the Terminal and type: `make -version` (I used version 3.81 for this book). This should output details about the current version of Make installed on your computer.

- Michael J. Crawley's (**?**) encyclopaedic R book, appropriately titled ***The R Book***, published by Wiley.
- Hadley Whickham (**?**) has a great new book out from Chapman and Hall on ***Advanced R***.
- Yihui Xie's (**?**) book ***R Markdown: The Definitive Guide***, published by Chapman and Hall, is needless to say the definitive guide on R Markdown syntax. It's a good complement to this book's generally more research project–level focus.
- Cathy O'Neil and Rachel Schutt (**?**) give a great introduction the field of data science generally in ***Doing Data Science***, published by O'Reilly Media Inc.
- For many real-world examples of reproducible research in action see Kitzes et al (**?**) collection of case studies ***The Practice of Reproducible Research***.
- For an excellent introduction to the command-line in Linux and Mac, see William E. Shotts Jr.'s (**?**) book ***The Linux Command-line: A Complete Introduction*** also published by No Starch Press. It is also helpful for Windows users running PowerShell (see Chapter **??**). Sean Kross' (**?**) ***The Unix Workbench*** is also a great freely available online introduction to the topic.
- The RStudio website (`http://www.rstudio.com/ide/docs/`) has a number of useful tutorials on how to use *knitr* with LaTeX and Markdown. They also have very good documentation for *rmarkdown* at `https://rmarkdown.rstudio.com/`.

That being said, my goal is for this book to be *self-sufficient*. A reader without a detailed understanding of these programs will be able to understand and use the commands and procedures I cover in this book. While learning how to use R and the other programs I personally often encountered illustrative examples that included commands, variables, and other things that were not well explained in the texts that I was reading. This caused me to waste many hours trying to figure out, for example, what the `$` is used for (preview: it's the component selector, see Section **??**. I hope to save you from this wasted time by either providing a brief explanation of possibly frustrating and mysterious things and/or pointing you in the direction of good explanations.

1.6.1 How to read this book

This book gives you a workflow. It has a beginning, middle, and end. So, unlike a reference book, it can and should be read linearly as it takes you through

an empirical research processes from an empty folder to a completed set of documents that reproducibly showcase your findings.

That being said, readers with more experience using tools like R or LaTeX may want to skip over the nitty-gritty parts of the book that describe how to manipulate data frames or compile LaTeX documents into PDFs. Please feel free to skip these sections.

More-experienced R users

If you are an experienced R user you may want to skip over the first section of Chapter **??**: Getting Started with R, RStudio, and *knitr*/*rmarkdown*. But don't skip over the whole chapter. The latter parts contain important information on the *knitr*/*rmarkdown* packages. If you are experienced with R data manipulation you may also want to skip all of Chapter **??**.

More-experienced LaTeX users

If you are familiar with LaTeX you might want to skip the first part of Chapter **??**. The second part may be useful as it includes information on how to dynamically create BibTeX bibliographies with *knitr* and how to include *knitr* output in a Beamer slideshow.

Less-experienced LaTeX/Markdown users

If you do not have experience with LaTeX or Markdown you may benefit from reading, or at least skimming, the introductory chapters on these top topics (chapters **??** and **??**) before reading Part III.

1.6.2 Reproduce this book

This book practices what it preaches. It can be reproduced. I wrote the book using the programs and methods that I describe. Full documentation and source files can be found at the book's GitHub repository. Feel free to read and even use (within reason and with attribution, of course) the book's source code. You can find it at: https://github.com/christophergandrud/Rep-Res-Book. This is especially useful if you want to know how to do something in the book that I don't directly cover in the text.

If you notice any errors or places where the book can be improved please report them on the book's GitHub Issues page: https://github.com/christophergandrud/Rep-Res-Bookissues. Corrections will be posted at: http://christophergandrud.GitHub.io/RepResR-RStudio/errata.htm.

1.6.3 Contents overview

The book is broken into four parts. The first part (chapters **??**, **??**, and **??**) gives an overview of the reproducible research workflow as well as the general computer skills that you'll need to use this workflow. Each of the next three parts of the book guides you through the specific skills you will need for each part of the reproducible research process. Part two (chapters **??**, **??**, and **??**) covers the data gathering and file storage process. The third part (chapters **??**, **??**, and **??**) teaches you how to dynamically incorporate your statistical analysis, results figures, and tables into your presentation documents. The final part (chapters **??**, **??**, and **??**) covers how to create reproducible presentation documents including LaTeX articles, books, slideshows, and batch reports as well as Markdown webpages and slideshows.

Appendix: Additional R Setup

Some setup is required to reproduce this book. Here are key R packages you should consider installing and specific instructions for Windows and Linux users.

R Packages

In this book I discuss how to use a number of user-written R packages for reproducible research. Many of these packages are not included in the default R installation. They need to be installed separately.

Note: in general you should aim to minimize the number of packages that your research depends on. Doing so will lessen the possibility that your code will "break" when a package is updated. This book depends on relatively many packages because of its special and unusual purpose of illustrating a variety of tools that you can use for reproducible research.

To install key user-written packages discussed in this book, copy the following code and paste it into your R console:

```
# Packages to install
pkg_to_install <- c("brew", "bookdown", "rio", "xfun")


# Check if the packages are installed, if not install them
```

```
lapply(
    pkg_to_install,
    function(pkg) {
        if (system.file(package = pkg) == "") {
            install.packages(pkg,
                repos = "http://cran.us.r-project.org"
            )
        }
    }
)
```

Note that I specified a US based R Project CRAN "mirror" to download the packages from.[21] There are many others to choose from. See: https://cran.r-project.org/mirrors.html.

The *xfun* package (**?**) contains a function called `pkg_attach2()`. When supplied with a vector of package names like those in `pkg_to_install` above, will install all non-installed packages. `p_load()` from the *pacman* package (**?**) works in a similar way. These functions are much less verbose than the example above, but they do require the user to install the package separately before `pkg_attach2()` or `p_load()` can be used. The example above relies only on functions available in the basic **R** installation.

Special issues for Windows and Linux Users

If you are using Windows, you will also need to install *Rtools*. You can install *Rtools* from: http://cran.r-project.org/bin/windows/Rtools/. Please use the recommended installation to ensure that your system PATH is set up correctly. Otherwise your computer will not know where the tools are. Alternatively, use the `install.Rtools()` function from the *installr* (**?**) package to install it.

[**CONSIDER REMOVING IF NOT NEEDED]

On Linux you will need to install the *RCurl* (**?**) and *XML* (**?**) packages separately. Use your Terminal to install these packages with the following code:

```
apt-get update

apt-get install libcurl4-gnutls-dev
apt-get install libxml2-dev
```

[21] CRAN stands for the Comprehensive R Archive Network.

```
apt-get install r-cran-rcurl-
apt-get install r-cran-xml
```

2

Getting Started with Reproducible Research

Researchers often start thinking about making their work reproducible near the end of the research process when they write up their results or maybe even later when a journal requires their data and code be made available for publication. Or maybe even later when another researcher asks if they can use the data from a published article to reproduce the findings. By then there may be numerous versions of the data set and records of the analyses stored across multiple folders on the researcher's computers. It can be difficult and time consuming to sift through these files to create an accurate account of how the results were reached. Waiting until near the end of the research process to start thinking about reproducibility can lead to incomplete documentation that does not give an accurate account of how findings were made. Focusing on reproducibility from the beginning of the process and continuing to follow a few simple guidelines throughout your research can help you avoid these problems. Remember "reproducibility is not an afterthought–it is something that must be built-into the project from the beginning" (**?**, 386).

This chapter first gives you a brief overview of the reproducible research process: a workflow for reproducible research. Then it covers some of the key guidelines that can help make your research more reproducible.

2.1 The Big Picture: A Workflow for Reproducible Research

The three basic stages of a typical computational empirical research project are:

- data gathering,
- data analysis,
- results presentation.

Each stage is part of the reproducible research workflow covered in this book. Tools for reproducibly gathering data are covered in Part II. Part III teaches

tools for tying the data we gathered to our statistical analyses and presenting the results with tables and figures. Part IV discusses how to tie these findings into a variety of documents you can use to advertise your findings.

Instead of starting to use the individual tools of reproducible research as soon as you learn them, I recommend briefly stepping back and considering how the stages of reproducible research *tie* together. This will make your workflow more coherent from the beginning and save you a lot of backtracking later on. Figure **??** illustrates the workflow. Notice that most of the arrows connecting the workflow's parts point in both directions, indicating that you should always be thinking about how to make it easier to go backwards through your research, i.e. reproduce it, as well as forwards.

Around the edges of the figure are some of the functions you will learn to make it easier to go forwards and backwards through the process. These functions tie your research together. For example, you can use API-based R packages to gather data from the internet. You can use R's `merge()` function to combine data gathered from different sources into one data set. The `getURL()` function from R's *RCurl* package (**?**) and the `read.table()` function in base R or the much more versatile `import()` function from the *rio* package (**?**) can be used to bring this data set into your statistical analyses. The *knitr* or *rmarkdown* package then ties your analyses into your presentation documents. This includes the code you used, the figures you created, and, with the help of tools such as the `kable()` function in the *knitr* package, tables of results. You can even tie multiple presentation documents together. For example, you can access the same figure for use in a LaTeX article and a Markdown-created website with the `includegraphics` and `![]()` functions, respectively. This helps you maintain a consistent presentation of results across multiple document types. We'll cover these functions in detail throughout the book. See Table **??** for a brief but more complete overview of the main *tie functions*.

2.1.1 Reproducible theory

An important part of the research process that I do not discuss in this book is theoretical stage. Ideally, if you are using a deductive research design, the bulk of this work will precede and guide the data gathering and analysis stages. Just because I don't cover this stage of the research process doesn't mean that theory building can't and shouldn't be reproducible. It can in fact be "the easiest part to make reproducible" (**?**, 1254). Quotes and paraphrases from previous works in the literature obviously need to be fully cited so that others can verify that they accurately reflect the source material. For mathematically based theory, clear and complete descriptions of the proofs should be given.

Though I don't actively cover theory replication in depth in this book, I do

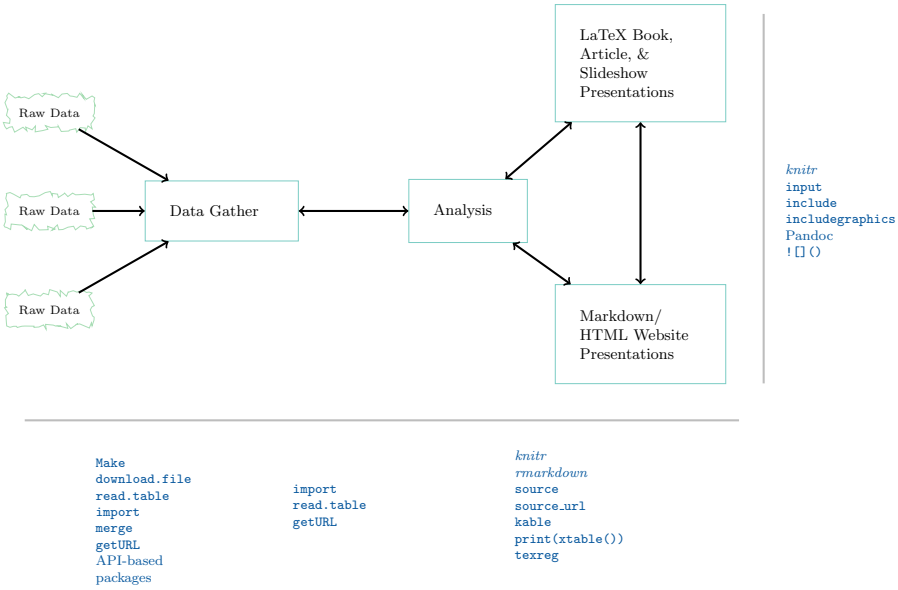

FIGURE 2.1: Example Workflow and a Selection of Functions to Tie It Together

touch on some of the ways to incorporate proofs and citations into your presentation documents. These tools are covered in Part IV.

2.2 Practical Tips for Reproducible Research

Before we start learning the details of the reproducible research workflow with R and RStudio, it's useful to cover a few broad tips that will help you organize your research process and put these skills in perspective. The tips are:

1. Document everything!
2. Everything is a (text) file.
3. All files should be human readable.
4. Explicitly tie your files together.
5. Have a plan to organize, store, and make your files available.

Using these tips will help make your computational research really reproducible.

2.2.1 Document everything!

In order to reproduce your research, others must be able to know what you did. You have to tell them what you did by documenting as much of your research process as possible. Ideally, you should tell your readers how you gathered your data, analyzed it, and presented the results. Documenting everything is the key to reproducible research and lies behind all of the other tips in this chapter and tools you will learn throughout the book.

Document your R session info

Before discussing the other tips it's important to learn a key part of documenting with R. You should *record your session info*. Many things in R have stayed the same since it was introduced in the early 1990s. This makes it easy for future researchers to recreate what was done in the past. However, things can change from one version of R to another and especially from one version of an R package to another. Also, the way R functions and how R packages are handled can vary across different operating systems, so it's important to note what system you used. Finally, you may have R set to load packages by default (see Section **??** for information about packages). These packages might be necessary to run your code, but other people might not know what packages and what versions of the packages were loaded from just looking at your source code. The `sessionInfo()` function in R prints a record of all of these things. The information from the session I used to create this book is:

```
# Print R session info
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS  10.14
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/C/en_GB.UTF-8/en_GB.UTF-8
##
## attached base packages:
```

```
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
## [1] ggplot2_3.0.0 magrittr_1.5
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.19     compiler_3.5.1   pillar_1.3.0
##  [4] plyr_1.8.4       highr_0.7        bindr_0.1.1
##  [7] tools_3.5.1      digest_0.6.18    evaluate_0.12
## [10] tibble_1.4.2     gtable_0.2.0     pkgconfig_2.0.2
## [13] rlang_0.2.2      rstudioapi_0.8   yaml_2.2.0
## [16] xfun_0.3         bindrcpp_0.2.2   withr_2.1.2
## [19] stringr_1.3.1    dplyr_0.7.7      knitr_1.20
## [22] rprojroot_1.3-2  grid_3.5.1       tidyselect_0.2.5
## [25] glue_1.3.0       R6_2.3.0         rmarkdown_1.10
## [28] bookdown_0.7     purrr_0.2.5      backports_1.1.2
## [31] scales_1.0.0     htmltools_0.3.6  assertthat_0.2.0
## [34] colorspace_1.3-2 stringi_1.2.4    lazyeval_0.2.1
## [37] munsell_0.5.0    crayon_1.3.4
```

Chapter **??** gives specific details about how to create files with dynamically included session information. If you use non-R tools you should also record what versions of these tools you used.

2.2.2 Everything is a (text) file

Your documentation is stored in files that include data, analysis code, the write-up of results, and explanations of these files (e.g. data set codebooks, session info files, and so on). Ideally, you should use the simplest file format possible to store this information. Usually the simplest file format is the humble, but versatile, text file.[1]

Text files are extremely nimble. They can hold your data in, for example, comma-separated values () format. They can contain your analysis code in files. And they can be the basis for your presentations as markup documents like or , for LaTeX and Markdown files, respectively. All of these files can be opened by any program that can read text files.

One reason reproducible research is best stored in text files is that this helps *future-proof* your research. Other file formats, like those used by Microsoft Word (`.docx`) or Excel (`.xlsx`), change regularly and may not be compatible

[1] Plain text files are usually given the file extension `.txt`. Depending on the size of your data set it may not be feasible to store it as a text file. Nonetheless, text files can still be used for analysis code and presentation files.

with future versions of these programs. Text files, on the other hand, can be opened by a very wide range of currently existing programs and, more likely than not, future ones as well. Even if future researchers do not have R or a LaTeX distribution, they will still be able to open your text files and, aided by frequent comments (see below), be able to understand how you conducted your research (**?**, 3).

Text files are also very easy to search and manipulate with a wide range of programs–such as R and RStudio–that can find and replace text characters as well as merge and separate files. Finally, text files are easy to version and changes can be tracked using programs such as Git (see Chapter **??**).

Learn from the text file: keep it simple

Text files are simple. Their simplicitly increases the probability of baseline usefulness in the future to researchers who will reproduce our work. We can extend the logic of the simple text file to all of the tools we use: keep it simple. Avoid adding dependencies you don't need to actually gather your data, analyze it, and present the results. For example, I have been tempted to make my presentation slides look nicer with new fonts. I was later burned when I wanted to make minor changes to slides a year after I first presented them (and a day before teaching an upcoming class) only to find that the custom fonts were no longer available. This broke my slides and forced me to spend considerable time reworking writing my source documents. If I, the creator of the slides, found this time consuming and annoying, imagine how an outside researcher would find it.

2.2.3 All files should be human readable

Treat all of your research files as if someone who has not worked on the project will, in the future, try to understand them. Computer code is a way of communicating with the computer. It is 'machine readable' in that the computer is able to use it to understand what you want to do.[2] However, there is a very good chance that other people (or you six months in the future) will not understand what you were telling the computer. So, you need to make all of your files 'human readable'. To make them human readable, you should comment on your code with the goal of communicating its design and purpose (**?**). With this in mind it is a good idea to *comment frequently* (**?**, 3) and *format your code using a style guide* (**?**). For especially important pieces of code you should use *literate programming*–where the source code and the presentation text describing its design and purpose appear in the same

[2] Of course, if the computer does not understand it will usually give an error message.

document. Doing this will make it very clear to others how you accomplished a piece of research.

Commenting

In R, everything on a line after a hash character—-(also known as number, pound, or sharp) is ignored by R, but is readable to people who open the file. The hash character is a comment declaration character. You can use the to place comments telling other people what you are doing. Here are some examples:

```
# A complete comment line
2 + 2 # A comment after R code
```

```
## [1] 4
```

On the first line the (hash) is placed at the very beginning, so the entire line is treated as a comment. On the second line the is placed after the simple equation `2 + 2`. R runs the equation and finds the answer , but it ignores all of the words after the hash.

Different languages have different comment declaration characters. In LaTeX everything after the percent sign is treated as a comment, and in Markdown/HTML comments are placed inside of . The hash character is used for comment declaration in command-line shell scripts.

Nagler (**?**, 491) gives some advice on when and how to use comments:

- write a comment before a block of code describing what the code does,
- comment on any line of code that is ambiguous.

In this book I follow these guidelines when displaying code. Nagler also suggests that all of your source code files should begin with a comment header. *At the least* the header should include:

- a description of what the file does,
- the date it was last updated,
- the name of the file's creator and any contributors.

You may also want to include other information in the header such as what files it depends on, what output files it produces, what version of the programming language you are using, sources that may have influenced the code, and how the code is licensed. Here is an example of a minimal file header for an R source code file that creates the third figure in an article titled 'My Article':

```
############################
# R Source code file used to create Figure 3 in My 'Article'
# Created by Christopher Gandrud
# MIT License
############################
```

Feel free to use things like the long series of hash marks above and below the header, white space, and indentations to make your comments more readable.

Style guides

In natural language writing you don't necessarily have to follow a style guide. People could probably figure out what you are trying to say, but it is a lot easier for your readers if you use consistent rules. The same is true when writing computer code. It's good to follow consistent rules for formatting your code so that it's easier for you and others to understand.

There are a number of R style guides. Most of them are similar to the Google R Style Guide.[3] Hadley Wickham also has a nicely presented R style guide.[4] You may want to use the *styler* (**?**) package to automatically reformat your code so that it is easier to read.

Literate programming

For particularly important pieces of research code it may be useful to not only comment on the source file, but also display code in presentation text. For example, you may want to include key parts of the code you used for your main statistical models and an explanation of this code in an appendix following your article. This is commonly referred to as literate programming (**?**).

2.2.4 Explicitly tie your files together

If everything is just a text file, then research projects can be thought of as individual text files that have a relationship with one another. They are tied together. A data file is used as input for an analysis file. The results of an analysis are shown and discussed in a markup file that is used to create a PDF document. Researchers often do not explicitly document the relationships between files that they used in their research. For example, the results of an analysis–a table or figure–may be copied and pasted into a presentation

[3] See: http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html.
[4] You can find it at http://adv-r.had.co.nz/Style.html.

document. It can be very difficult for future researchers to trace the table or figure back to a particular statistical model and a particular data set without clear documentation. Therefore, it is important to make the links between your files explicit.

Tie functions are the most dynamic way to explicitly link your files together. These functions instruct the computer program you are using to use information from another file. In Table **??** I have compiled a selection of key tie functions you will learn how to use in this book. We'll discuss many more, but these are some of the most important.

2.2.5 Have a plan to organize, store, and make your files available

Finally, in order for independent researchers to reproduce your work, they need to be able access the files that instruct them how to do this. Files also need to be organized so that independent researchers can figure out how they fit together. So, from the beginning of your research process you should have a plan for organizing your files and a way to make them accessible.

One rule of thumb for organizing your research in files is to limit the amount of content any one file has. Files that contain many different operations can be very difficult to navigate, even if they have detailed comments. For example, it would be very difficult to find any particular operation in a file that contained the code used to gather the data, run all of the statistical models, and create the results figures and tables. If you have a hard time finding things in a file you created, think of the difficulties independent researchers will have!

Because we have so many ways to link files together, there is really no need to lump many different operations into one file. So, we can make our files modular. One source code file should be used to complete one or just a few tasks. Breaking your operations into discrete parts will also make it easier for you and others to find errors (**?**, 490).

Chapter **??** discusses file organization in much more detail. Chapter **??** teaches you a number of ways to make your files accessible through the cloud computing services like GitHub.

TABLE 2.1: A Selection of Functions/Packages/Programs for Tying Together Your Research Files

Function/Package/ Program	Language	Description	Chapters Discussed
knitr	R	R package with commands for tying analysis code into presentation documents including those written in LaTeX and Markdown.	Throughout
rmarkdown	R	R package that builds on *knitr*. It allows you to use Markdown to output to HTML, PDFs compiled with LaTeX or Microsoft Word.	Throughout
`download.file`	R	Downloads a file from the internet.	??
`read.table`	R	Reads a table into R. You can use this to import a plain-text file formatted data into R.	??
`read.csv`	R	Same as `read.table` with default arguments set to import `.csv` formatted data files.	??
`import`	R	Reads a table stored locally or on the internet into R. You can use it to import a wide variety of plain-text data formats into R from secure (https) URLs.	??
API-based packages	R	Various packages use APIs to gather data from the internet.	??
`merge`	R	Merges together data frames.	??
`source`	R	Runs an R source code file.	??
`source_url`	R	From the *devtools* package. Runs an R source code file from a secure (`https`) url like those used by GitHub.	??
`kable`	R	Creates tables from data frames that can be rendered using Markdown or LaTeX.	??
`toLaTeX`	R	Converts R objects to LaTeX.	??
`input`	LaTeX	Includes LaTeX files inside of other LaTeX files.	??
`include`	LaTeX	Similar to `input`, but puts page breaks on either side of the included text. Usually it is used for including chapters.	??
`includegraphics`	LaTeX	Inserts a figure into a LaTeX document.	??
`![]()`	Markdown	Inserts a figure into a Markdown document.	??
Pandoc	shell	A shell program for converting files from one markup language to another. Allows you to tie presentation documents together.	?? & ??
Make	shell	A shell program for automatically building many files at the same time.	??

3

Getting Started with R, RStudio, and knitr/rmarkdown

If you have rarely or never used R before, the first section of this chapter gives you enough information to be able to get started and understand the R code I use throughout the book. For more detailed introductions on how to use R please refer to the resources I mentioned in Chapter **??** (Section **??**). Experienced R users might want to skip the first section. In the second section I'll give a brief overview of RStudio. I highlight the key features of the main RStudio panel (what appears when you open RStudio) and some of its key features for reproducible research. Finally, I discuss the basics of the *knitr* and *rmarkdown* packages, how to use them in R, and how they are integrated into RStudio.

3.1 Using R: The Basics

To get you started with reproducible research, we'll cover some very basic R syntax–the rules for talking to R. I cover key parts of the R language including:

- objects & assignment,
- component selection,
- functions,
- arguments,
- the workspace and history,
- packages.

Before discussing each of these in detail, let's open R and look around.[1] When you open the R GUI program by clicking on the R icon you should get a

[1] Please see Chapter **??** for instructions on how to install R.

window that looks something like Figure **??**.[2] This window is the **R console**. Below the startup information–information about what version of R you are using, license details, and so on–you should see a > (greater-than sign). This prompt is where you enter R code.[3] To run R code that you have typed after the prompt, hit the or key. Now that we have a new R session open we can get started.

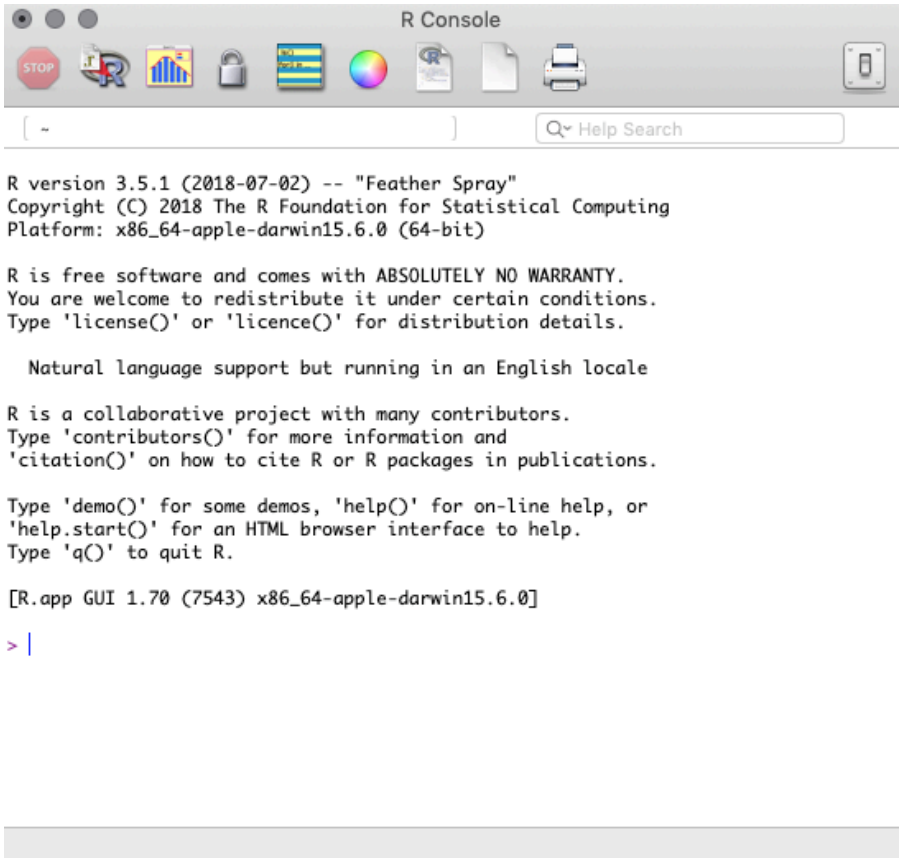

FIGURE 3.1: R Console at Startup

[2]This figure and almost all screenshots in this book were taken on a computer using the Mac OS 10.10 operating system.

[3]If you are using a Unix-like system such as Linux Ubuntu or Mac OS 10, you can also access R via an application called the Terminal. If you have installed R on your computer you can type into the Terminal and then the or key. This will begin a new R session. You will know you are in a new R session because the same type of startup information as in Figure **??** will be printed in your Terminal.

3.1.1 Objects

If you've read a description of R before, you will probably have seen it referred to as an 'object-oriented language'. What are objects? Objects are like the R language's nouns. They are things, like a vector of numbers, a data set, a word, a table of results from some analysis, and so on. Saying that R is object-oriented means that R is focused on doing actions to objects. We will talk about the actions–functions and functions–later in this section.[4] Now let's create a few objects.

Numeric & string objects

Objects can have a number of different types. Let's make two simple objects. The first is a numeric-type object. The other is a character object. We can choose almost any name we want for our objects as long as it begins with an alphabetic character and does not contain spaces.[5] Let's call our numeric object *number*. It is a good idea to give each object a unique name to avoid conflicts and confusion. Also make sure that object names are different from the names of their components, e.g. individual variable names. This will avoid many complications like accidentally overwriting an object or confusing R about what object or component you are referring to.

To put something into the object we use the assignment operator[6] (`<-`). Let's assign the number 10 to our *number* object.

```
number <- 10
```

To see the contents of our object, type its name.

```
number
```

```
## [1] 10
```

Let's briefly breakdown this output. `10` is clearly the contents of *number*. The double hash (`##`) is included here to tell you that this is output rather than R code.[7] If you run functions in your R console, you will not get the double

[4] Somewhat confusingly, functions and functions are themselves objects. In this chapter I treat them as distinct from other object types to avoid confusion.

[5] It is common for people to use either periods (`.`) or capital letters (referred to as CamelBack) to separate words in object names instead of using spaces. For example: *new.data* or *NewData* rather than *new data*. For more information on R naming conventions see `?`.

[6] The assignment operator is sometimes also referred to as the 'gets arrow'.

[7] The double hash is generated automatically by *knitr*. They make it easier to copy and paste code into R from a document created by *knitr* because R will ignore everything after a hash.

hash in your output. Finally, `[1]` is the row number of the object that 10 is on. Clearly our object only has one row.

Creating an object with words and other characters–a character object–is very similar. The only difference is that you enclose the character string (letters in a word for example) inside of single or double quotation marks (`''`, or `""`).[8] To create an object called *words* that contains the character string "Hello World":

```
words <- "Hello World"
```

An object's type is important to keep in mind as it determines what we can do to it. For example, you cannot take the mean of a character object like the *words* object:

```
mean(words)
```

```
## Warning in mean.default(words): argument is not numeric
## or logical: returning NA
```

```
## [1] NA
```

Trying to find the mean of our *words* object gives us a warning message and returns the value NA: not applicable. You can also think of `NA` as meaning "missing". To find out an object's type, use the `class()` function. For example:

```
class(words)
```

```
## [1] "character"
```

Vector & data frame objects

So far we have only looked at objects with a single number or character string.[9] Clearly we often want to use objects that have many strings and numbers. In R these are usually data frame-type objects and are roughly equivalent to the data structures you would be familiar with from using a program such as Microsoft Excel. We will be using data frames extensively throughout the book. Before looking at data frames it is useful to first look at the simpler objects that make up data frames. These are called vectors. Vectors are R's

[8] Single and double quotation marks are interchangeable in R for this purpose. In this book I always use double quotes, except for *knitr* code chunk options.

[9] These might be called scalar objects, though in R scalars are just vectors with a length of 1.

"workhorse" (**?**). Knowing how to use vectors will be especially helpful when you cleanup raw data in Chapter **??** and make tables in Chapter **??**.[10]

3.1.1.0.1 Vectors

Vectors are the "fundamental data type" in R (**?**). They are simply an ordered group of numbers, character strings, and so on.[11] It may be useful to think of basically all R objects as composed of vectors. For example, data frames are basically multiple vectors of the same length–i.e. they have the same number of rows–attached together to form columns.

Let's create a simple numeric vector containing the numbers 2.8, 2, and 14.8. To do this we will use the `c()` (combine) function:

```
numeric_vector <- c(2.8, 2, 14.8)

# Show numeric_vector's contents
numeric_vector
```

```
## [1]  2.8  2.0 14.8
```

Vectors of character strings are created in a similar way. The only major difference is that each character string is enclosed in quotation marks like this:

```
character_vector <- c("Albania", "Botswana", "Cambodia")

# Show character_vector's contents
character_vector
```

```
## [1] "Albania"  "Botswana" "Cambodia"
```

To give you a preview of what we are going to do when we start working with real data sets, let's combine the two vectors *numeric_vector* and *character_vector* into a new object with the `cbind()` function. This function binds the two vectors together side-by-side as columns.[12]

[10] If you want information about other types of R objects such as lists and matrices, Chapter 1 of Norman Matloff's (**?**) book is a really good place to look.

[11] In a vector, every member of the group must be of the same type. If you want an ordered group of values with different types you can use lists.

[12] If you want to combine objects as if they were rows of the same column(s), use the `rbind()` function.

```
string_num_object <- cbind(character_vector, numeric_vector)

# Show string_num_object's contents
string_num_object
```

```
##      character_vector numeric_vector
## [1,] "Albania"        "2.8"
## [2,] "Botswana"       "2"
## [3,] "Cambodia"       "14.8"
```

By binding these two objects together we've created a new matrix object.[13] You can see that the numbers in the *numeric_vector* column are between quotation marks. Matrices, like vectors, can only have one data type.

3.1.1.0.2 Data frames

If we want to have an object with rows and columns and allow the columns to contain data with different types, we need to use data frames. Let's use the `data.frame` function to combine the *numeric_vector* and *character_vector* objects.

```
string_num_object <- data.frame(character_vector, numeric_vector)

# Display contents of string_num_object data frame
string_num_object
```

```
##   character_vector numeric_vector
## 1          Albania            2.8
## 2         Botswana            2.0
## 3         Cambodia           14.8
```

There are a few important things to notice in this output. The first is that because we used the same name for the data frame object as the previous matrix object, R deleted the matrix object and replaced it with the data frame. This is something to keep in mind when you are creating new objects. In general it is a better idea to assign elements to new objects rather than overwriting old ones. This will help avoid accidentally using an object you had not intended to use. It also allows you to more easily change previously run source code.

[13] Matrices are vectors with columns as well as rows.

You can see the data frame's names attribute.[14] It is the column names. You can use the `names()` function to see any data frame's names:[15]

```
names(string_num_object)
```

```
## [1] "character_vector" "numeric_vector"
```

You will also notice that the first column of the data set has no name and is a series of numbers. This is the row.names attribute. Data frame rows can be given any name as long as each row name is unique. We can use the `row.names()` function to set the row names from a vector. For example,

```
# Reassign row.names
row.names(string_num_object) <- c("First", "Second", "Third")

# Display new row.names
row.names(string_num_object)
```

```
## [1] "First"  "Second" "Third"
```

You can see in this example how `row.names()` can also be used to print the row names.[16] The *row.names* attribute does not behave like a regular data frame column. You cannot, for example, include it as a variable in a regression. You can use the `row.names()` function to assign the *row.names* values to a regular column (for an example see Section **??**.

You will notice in the output for *string_num_object* that the strings in the **character_vector** column are no longer in quotation marks. This does not mean that they are now numeric data. To prove this, try to find the mean of **character_vector** by running it through the `mean()` function:

```
mean(string_num_object$character_vector)
```

```
## Warning in
## mean.default(string_num_object$character_vector):
## argument is not numeric or logical: returning NA

## [1] NA
```

[14] Matrices can also have a names attribute.

[15] You can also use `names()` to assign names for the entire data frame. For example, `names(string_num_object) <- c(variable_1, variable_2)`

[16] Note that this is really only useful for data frames with few rows.

Component selection}

The last bit of code we just saw will probably be confusing. Why do we have a dollar sign (`$`) between the name of our data frame object name and the `character_vector` variable? The dollar sign is called the component selector. It's also sometimes called the element name operator. Either way, it extracts a part–component–of an object. In the previous example it extracted the **character_vector** column from the *string_num_object* and fed it to the `mean()` function, which tried (in this case unsuccessfully) to find its mean.

We can, of course, use the component selector to create new objects with parts of other objects. Imagine that we have the *string_num_object* and want an object with only the information in the numbers column. Let's use the following code:

```
numeric_extract <- string_num_object$numeric_vector

# Display contents of numeric_extract
numeric_extract
```

```
## [1]  2.8  2.0 14.8
```

Knowing how to use the component selector will be especially useful when we discuss making tables for presentation documents in Chapter **??**.

`attach` and `with`

Using the component selector can create long repetitive code if you want to select many components. You have to write the object name, a dollar sign, and the component name every time you want to select a component. You can streamline your code by using functions such as `attach()` and `with()`.

`attach()` attaches a database to R's search path.[17] R will then search the database for variables you specify. You don't need to use the component selector to tell R again to look in a particular data frame after you have attached it. For example, let's attach the *cars* data that comes with R. It has two variables, **speed** and **dist**.[18]

```
# Attach cars to search path
attach(cars)
```

[17] You can see what is in your current search path with the `search` function. Just type `search()` into your R console.

[18] For more information on this data set, type `?cars` into your R console.

```
# Display speed
head(speed)
```

```
## [1] 4 4 7 7 8 9
```

```
# Display dist
head(dist)
```

```
## [1]  2 10  4 22 16 10
```

```
# Detach cars
detach(cars)
```

We used the `head()` function to see just the first few values of each variable. It is a good idea to `detach()` a data frame after you are done using it, to avoid confusing R.

Similarly, you can use `with` when you run functions using a particular database (see Section **??** for more details about functions). For example, we can find the mean of *numeric_vector* `with()` the *string_num_object* data frame:

```
with(string_num_object, {
        mean(numeric_vector)
    }
)
```

```
## [1] 6.533
```

You can see that in the `with()` call the data frame object goes first and then the `mean()` function[19] goes second in curly brackets (`{}`).

In this book I largely avoid using the `attach()` and `with()` functions. Instead I tend to use the component selector. Though it creates longer code, I find that code written with the component selector is less ambiguous It's always clear which object we are selecting a component from. Nonetheless, **attach()** and **with()** are can be useful for streamlining your R code.

[19] Using R terminology, the second "argument" value–the code after the comma–of the **with** function is called an "expression", because it can contain more than one R function or statement. See Section **??** for a more comprehensive discussion of R function arguments.

3.1.2 Subscripts

Another way to select parts of an object is to use subscripts. You have already seen subscripts in the output from our examples so far. They are denoted with square braces (`[]`). We can use subscripts to select not only columns from data frames but also rows and individual values. As we began to see in some of the previous output, each part of a data frame has an address captured by its row and column number. We can tell R to find a part of an object by putting the row number/name, column number/name, or both in square braces. The first part denotes the rows and separated by a comma (`,`) are the columns.

To give you an idea of how this works let's use the *cars* data set that comes with R. Use `head()` to get a sense of what this data set looks like.

```
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

We can see a data frame with information on various cars' speeds (**speed**) and stopping distances (**dist**). If we want to select only the third through seventh rows we can use the following subscript functions:

```
cars[3:7, ]
```

```
##   speed dist
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
## 7    10   18
```

The colon (`:`) creates a sequence of whole numbers from 3 to 7. To select the fourth row of the **dist** column we can type:

```
cars[4, 2]
```

```
## [1] 22
```

An equivalent way to do this is:

```
cars[4, "dist"]
```

```
## [1] 22
```

Finally, we can even include a vector of column names to select:

```
cars[4, c("speed", "dist")]
```

```
##   speed dist
## 4     7   22
```

3.1.3 Functions

If objects are the nouns of the R language, functions are the verbs. They do things to objects. Let's use the `mean` function as an example. This function takes the mean of a numeric vector object. Remember our *numeric_vector* object from before:

```
# Show contents of numeric_vector
numeric_vector
```

```
## [1]  2.8  2.0 14.8
```

To find the mean of this object simply type:

```
mean(x = numeric_vector)
```

```
## [1] 6.533
```

We use the assignment operator to place a function's output into an object. For example:

```
numeric_vector_mean <- mean(x = numeric_vector)
```

Notice that we typed the function's name then enclosed the object name in parentheses immediately afterwards. This is the basic syntax that all functions use, i.e. `FUNCTION(ARGUMENTS)`. If you don't want to explicitly include an argument, *you still need to type the parentheses after the function.*

3.1.4 Arguments

Arguments modify what functions do. In our most recent example we gave the `mean` function one argument (`x = numeric_vector`) telling it that we wanted to find the mean of *numeric_vector*. Arguments use the `ARGUMENT_LABEL = VALUE` syntax.[20] In this case **x** is the argument label.

To find all of the arguments that a function can accept, look at the **Arguments** section of the functions's help file. To access the help file type: `?FUNCTION`. For example,

```
?mean
```

The help file will also tell you the default values that the arguments are set to. Clearly, you do not need to explicitly set an argument if you want to use its default value.

You do have to be fairly precise with the syntax for your argument's values. Values for logical arguments must written as `TRUE` or `FALSE`.[21] Arguments that accept character strings require quotation marks.

Let's see how to use multiple arguments with the `round()` function. This function rounds a vector of numbers. We can use the `digits` argument to specify how many decimal places we want the numbers rounded to. To round the object *numeric_vector_mean* to one decimal place type:

```
round(x = numeric_vector_mean, digits = 1)
```

```
## [1] 6.5
```

Note that arguments are separated by commas.

Some arguments do not need to be explicitly labeled. For example, we could have written:

```
# Find mean of numeric_vector
mean(numeric_vector)
```

```
## [1] 6.533
```

R will do its best to figure out what you want and will only give up when it can't. This will generate an error message. However, to avoid any misunderstandings between yourself and R it can be good practice to label most of

[20] Note: you do not have to put spaces between the argument label and the equals sign or the equals sign and the value. However, having spaces can make your code easier for other people to read.

[21] They can be abbreviated `T` and `F`.

your arguments. This will also make your code easier for other people to read, i.e. it will be more reproducible.

You can stack arguments inside of other arguments. To have R find the mean of *numeric_vector* and round it to one decimal place use:

```
round(mean(numeric_vector), digits = 1)
```

```
## [1] 6.5
```

Stacking functions inside of each other can create code that is difficult to read. Another option that potentially makes more easily understandable code is piping using the pipe function (`%>%`) that you can access from the *magrittr* package (`?`). The basic idea behind the pipe function is that the output of one function is set as the first argument of the next. For example, to find the mean of **numeric_vector** and then round it to one decimal place use:

```
# Load magrittr package
library(magrittr)

# Find mean of numeric_vector and round to 1 decimal place
mean(numeric_vector) %>%
    round(digits = 1)
```

```
## [1] 6.5
```

3.1.5 The workspace & history

All of the objects you create become part of your workspace, alternatively known as the current working environment. Use the `ls()` function to list all of the objects in your current workspace.[22]

```
ls()
```

```
## [1] "character_vector"    "number"
## [3] "numeric_extract"     "numeric_vector"
## [5] "numeric_vector_mean" "pkg_to_install"
## [7] "string_num_object"   "words"
```

You can remove specific objects from the workspace using the `rm()` function. For example, to remove the `character_vector` and `words` objects type:

[22] Note: your workspace will probably include different objects than this example. These are objects created to knit the book.

```
rm(character_vector, words)
```

To save the entire workspace into a binary–not plain-text–RData file use `save.image()`**??**. The main argument of `save.image()` is the location and name of the file in which you want to save the workspace. If you don't specify the file path it will be saved into your current working directory (see Chapter **??** for information on files paths and working directories). For example, to save the current workspace in a file called *workspace_2018-10-28.RData* in the current working directory type:

```
save.image(file = "workspace_2018-10-28.RData")
```

Use `load()` to load a saved workspace back into R:

```
load(file = "workspace_2018-10-28.RData")
```

You should generally avoid having R automatically save your workspace when you quit and reload it when you start R again. Instead, when you return to working on a project, rerun the source code files. This avoids any complications caused when you use an object in your workspace that is left over from running an older version of the source code.[23] In general I also recommend against saving data in binary RData formatted files. Because they are not text files they are not human readable and are much less future-proof.

One of the few times when saving your workspace is very useful is when it includes an object that was computationally difficult and took a long time to create. In this case you can save only the large object with `save()`.[24]**??** For example, if we have a very large object called *comprehensive* we can save it to a file called *comprehensive.RData* like this:

```
save(comp, file = "comprehensive.RData")
```

3.1.5.0.1 R history

When you enter a function into R it becomes part of your history. To see the

[23] For example, imagine you create an object, then change the source code you used to create the object. However, there is a syntax error in the new version of the source code. The old object won't be overwritten and you will be mistakenly using the old object in future functions.

[24] `save.image()` is just a special case of `save()`.

most recent functions in your history use the `history()` function. You can also use the up and down arrows on your keyboard when your cursor is in the R console to scroll through your history.

3.1.6 Global R options

In R you can set global options with `options()`. This lets you set how R runs and outputs functions through an entire R session. For example, to have output rounded to one decimal place, set the `digits` argument:

```
options(digits = 1)
```

3.1.7 Installing new packages and loading functions

Commands are stored in R packages. The functions we have used so far were loaded automatically by default. One of the great things about R is the many user-created packages[25] that greatly expand the number of functions we can use. To install functions that do not come with the basic R installation you need to install the add-on packages**??** that contain them. To do this, use the function. By default this function downloads and installs the packages from the Comprehensive R Archive Network (CRAN).

For the code you need to install all of the packages used in this book, see page . When you install a package, you will likely be given a list of mirrors from which you can download the package. Simply select the mirror closest to you.

Once you have installed a package you need to load it so that you can use its functions. Use the `library()` function to load a package.[26] Use the following code to load the *ggplot2* package that we use in Chapter **??** to create figures.

```
library(ggplot2)
```

Please note that for the examples in this book I only specify what package a function is from if it is not loaded by default when you start an R session.

Finally, if you want to make sure R uses a function from a specific package

[25] For the latest list see: http://cran.r-project.org/web/packages/available_packages_by_name.html.

[26] You will probably see R packages referred to as "libraries", though this is a misnomer. See this blog post by Carlisle Rainey for a discussion: http://www.carlislerainey.com/2013/01/02/packages-v-libraries-in-r/?utm_source=rss&utm_medium=rss&utm_campaign=packages-v-libraries-in-r (posted 2 January 2013).

you can use the double-colon operator (`::`). For example, to make sure that we use the `qplot` function from the *ggplot2* package we type:

```
ggplot2::qplot(. . .)
```

We can use the double-colon to simplify our code as we don't need to include `library(. . .)`. Using the double-colon in this way ensures that R will use the function from the particular package you want and makes it clear to a source code reader what package a function comes from. Note that it does not load all of the functions in the package, just the one you ask for.

3.2 Using RStudio

As I mentioned in Chapter **??**, RStudio is an integrated development environment for R. It provides a centralized and well-organized place to do almost anything you want to do with R. As we will see later in this chapter, it is especially well integrated with literate programming tools for reproducible research. Right now let's take a quick tour of the basic RStudio window.

3.2.0.0.1 The default window

When you first open RStudio you should see a default window that looks like Figure **??**. In this figure you see three window panes. The large one on the left is the *Console*. This pane functions exactly the same as the console in regular R. Other panes include the *Environment/History* panes, in the upper right-hand corner. The *Environment* pane shows you all of the objects in your workspace and some of their characteristics, like how many observations a data frame has. You can click on an object in this pane to see its contents. This is especially useful for quickly looking at a data set in much the same way that you can visually scan a Microsoft Excel spreadsheet. The *History* pane records all of the functions you have run. It also allows you to rerun code and insert it into a source code file.

In the lower right-hand corner you will see the *Files/Plots/Packages/ Help/Viewer* panes. We will discuss the *Files* pane in more detail in Chapter **??**. Basically, it allows you to see and organize your files. The *Plots* pane is where figures you create in R appear. This pane allows you to see all of the figures you have created in a session using the right and left arrow icons. It also lets you save the figures in a variety of formats. The *Packages* pane