

Christopher Gandrud

Reproducible Research with R and RStudio









Preface

The preface is incomplete.

This book would not have been possible without the advice and support of a great many people.

The developer and blogging community has been incredibly important for making this book possible. Foremost among these people is Yihui Xie. He is the developer of the *knitr* package (among others) and also an avid writer and reader of blogs. 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 bloggers include Carl Boettiger (who also developed the *knitcitations* package), Markus Gesmann (who developed *GoogleVis*), Jeromy Anglim.

The vibrant and very helpful communities at Stack Overflow <http://stackoverflow.com/> and Stack Exchange <http://stackexchange.com/> are always very helpful for finding answers to problems that plague any coder. Importantly they makes it easy for others to find the answers to questions that others have asked before.

My students at Yonsei University were also an important part of creating this book. 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 the Introduction to Social Science Data Analysis that I teach. I tested many of the explanations and examples with my students. Their feedback has been very helpful for making the book clearer and more useful.

Contents

I	Getting Started	1
1	Introducing Reproducible Research	3
1.1	What is reproducible research?	3
1.2	Why should research be reproducible?	5
1.2.1	For Science	5
1.2.2	For You	6
1.3	Who should read this book?	7
1.3.1	Academic Researchers	8
1.3.2	Students	8
1.3.3	Instructors	8
1.3.4	Editors	9
1.3.5	Private sector researchers	9
1.4	The Tools of Reproducible Research	9
1.5	Why use R, knitr, and RStudio for reproducible research? . .	10
1.5.1	Installing the Software	12
1.6	Book overview	13
1.6.1	What this book is not.	13
1.6.2	How to read this book	14
1.6.3	How this book was written	14
1.6.4	Contents overview	15
2	Getting Started with Reproducible Research	17
2.1	The Big Picture: A workflow for reproducible research	17
2.1.1	Reproducible Theory	20
2.2	Practical tips for reproducible research	20
2.2.1	Document everything!	20
2.2.2	Everything is a (text) file	22
2.2.3	All files should be human readable	23
2.2.4	Reproducible research projects are many files explicitly tied together	25
2.2.5	Have a plan to organize, store, & make your files avail- able	25

3	Getting Started with R, RStudio, and knitr	29
3.1	Using R: the basics	29
3.1.1	Objects	30
3.1.2	Component Selection	34
3.1.3	Subscripts	35
3.1.4	Functions and commands	37
3.1.5	Arguments	38
3.1.6	The Workspace & History	39
3.1.7	Installing new libraries and loading commands	40
3.2	Using RStudio	41
3.3	Using knitr: the basics	42
3.3.1	File extensions	43
3.3.2	Code Chunks	43
3.3.3	Global options	46
3.3.4	knitr package options	47
3.3.5	Hooks	48
3.3.6	knitr & RStudio	48
3.3.7	knitr & R	50
4	Getting Started with File Management	53
4.1	File paths & naming conventions	54
4.1.1	Root directories	54
4.1.2	Subdirectories	54
4.1.3	Working directories	55
4.2	Organizing your research project	55
4.3	Setting directories as RStudio Projects	57
4.4	R file manipulation commands	58
4.5	Unix-like shell commands for file management	59
4.6	File navigation in RStudio	61
II	Data Gathering and Storage	63
5	Storing, Collaborating, Accessing Files, Versioning	65
5.1	Saving data in reproducible formats	66
5.2	Storing your files in the cloud	67
5.2.1	Dropbox	67
5.2.2	Storage	68
5.2.2.1	Accessing Data	68
5.2.3	Collaboration	69
5.2.3.1	Version control	69
5.2.4	GitHub	70
5.2.4.1	Setting up GitHub: Basic	71
5.2.4.2	Storage on GitHub	72
5.2.4.3	Accessing on GitHub	72
5.2.4.4	Collaboration with GitHub	72

5.2.4.5	Version Control with GitHub	72
6	Gathering Data with R	75
6.1	Organize your data gathering: make files	75
6.2	Importing locally stored data sets	77
6.2.1	Importing a single locally stored file	77
6.2.2	Looping through multiple files	78
6.3	Importing data sets from the internet	78
6.3.1	Data from non-secure (http) URLs	78
6.3.2	Data from secure (https) URLs	79
6.3.3	Compressed data stored online	80
6.3.4	Data APIs & feeds	81
6.4	Basic web scraping	82
6.4.1	Gathering and parsing text from the web	82
6.4.2	Scraping tables	82
7	Preparing Data for Analysis	83
7.1	Cleaning data for merging	83
7.1.1	Renaming variables	83
7.1.2	Changing variables types	83
7.1.3	Creating ID Variables	83
7.1.4	Sorting & ordering data	83
7.2	Merging data sets	83
7.2.1	Binding	83
7.2.2	The merge command	83
III	Analysis and Results	85
8	Statistical Modelling and knitr	87
8.1	Incorporating analyses into the markup	87
8.1.1	Full code chunks	88
8.1.2	Showing code & results inline	88
8.1.2.1	LaTeX	89
8.1.2.2	Markdown	89
8.1.3	Dynamically including non-R code	90
8.2	Dynamically including segmented analysis files	91
8.2.1	Source from a local file	91
8.2.2	Source from a non-secure URL (http)	92
8.2.3	Source from a secure URL (https)	92
9	Showing Results with Tables	95
9.1	Table Basics	96
9.1.1	Tables in LaTeX	96
9.1.2	Tables in Markdown/HTML	96
9.2	Creating tables from R objects	96
9.2.1	xtable & texreg basics with supported class objects	96

9.2.1.1	<code>xtable</code> for LaTeX	96
9.2.1.2	<code>xtable</code> for Markdown	96
9.2.2	<code>xtable</code> with non-supported class objects	96
9.2.3	Basic <code>knitr</code> syntax for tables	98
9.3	Tables with <code>apsrtable</code>	99
10	Showing Results with Figures	101
10.1	Including graphics	101
10.2	Basic <code>knitr</code> figure options	101
10.3	Creating figures with <code>plot</code> and <code>ggplot2</code>	101
10.4	Animations	101
10.5	Motion charts and basic maps with <code>GoogleVis</code>	101
IV	Presentation Documents	103
11	Presenting with LaTeX	105
11.1	The Basics	105
11.1.1	Editors	105
11.1.2	The header & the body	106
11.1.3	Headings	106
11.1.4	Footnotes & Bibliographies	106
11.1.4.1	Footnotes	106
11.1.4.2	Bibliographies	106
11.2	Presentations with Beamer	109
12	Large LaTeX Documents: Theses, Books, & Batch Reports	111
12.1	Planning large documents	111
12.1.1	Planning theses and books	111
12.1.2	Planning batch reports	111
12.2	Combining Chapters	112
12.2.1	Parent documents	112
12.2.2	Child documents	112
12.3	Creating Batch Reports	113
12.3.1	<code>stich</code>	113
13	Presenting on the Web and Beyond with Markdown/HTML	115
13.1	The Basics	115
13.1.1	Headings	115
13.1.2	Footnotes and bibliographies with <code>MultiMarkdown</code>	115
13.1.3	Math	115
13.1.4	Drawing figures with <code>CSS</code>	115
13.2	Simple webpages	115
13.2.1	<code>RPubs</code>	115
13.2.2	Hosting webpages with <code>Dropbox</code>	115
13.3	Presentations with <code>Slidify</code>	115
13.4	Reproducible websites	115

	xi
13.4.1 Blogging with Tumblr	115
13.4.2 Jekyll-Bootstrap and GitHub	115
13.4.3 Jekyll and Github Pages	116
13.5 Using Markdown for non-HTML output with Pandoc	116
14 Going Beyond the Book	117
14.1 Licensing Your Reproducible Research	117
Index	123

Stylistic Conventions

I use the following conventions throughout this book to format computer code:

- **Abstract Variables**

Abstract variables, i.e. variables that do not represent specific objects in an example, are in ALL CAPS TYPWRITER 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*. Camelback is used for file and directory names.

- **Individual variable values**

Individual variable values mentioned in the text are in **bold**.

- **Objects**

Objects are printed in *italics*. Camelback (e.g. CamelBack) is used for object names.

- **Columns**

Columns are printed in *italics*

- **Packages**

R packages are printed in *italics*.

- **Windows**

Open windows are written in **bold** text.

- **Variable Names**

Variable names are printed in *italics*. Camelback is used for individual variable names.

Required R Packages

This book discusses how to use a number of user-written R packages for reproducible research. These are not included in the default R installation (see Section 1.5.1). They need to be installed separately. To install all of the user-written packages discussed in this book use the following code:

```
install.packages("apsrtable",  
                 "devtools",  
                 "formatR",  
                 "ggplot2",  
                 "knitr",  
                 "knitcitations",  
                 "markdown",  
                 "openair",  
                 "RCurl",  
                 "texreg",  
                 "tools",  
                 "xtable",  
                 "Zelig")
```

Once you enter this code, you may be asked to select a “mirror” to download the packages from. Simply select the mirror closest to you.

List of Figures

2.1	Example Workflow & Commands to Tie it Together	19
3.1	R Startup Console	30
3.2	RStudio Startup Panel	41
3.3	RStudio Source Code Pane Top Bars	43
3.7	RStudio Notebook Example	49
3.8	Folding Code Chunks in RStudio	51
4.1	Example Research Project File Tree	56
4.2	An Example RStudio Project Menu	57
4.3	The RStudio Files Pane	60
5.1	Part of this Book's GitHub Repository Webpage	74

List of Tables

2.1	A Selection of Commands for Tying Together Your Research Files	26
3.1	A Selection of <i>knitr</i> Code Chunk Options	47
8.1	Knitr engine Values	91
9.1	Coefficient Estimates Predicting Examination Scores in Swiss Cantons (1888) Found Using Bayesian Normal Linear Regression	98



xx





Part I

Getting Started



1

Introducing Reproducible Research

Research is often presented in a very abridged package: a conference presentation, journal article, book, or maybe even a website. These presentation documents announce a project’s findings and try to convince us that the results are correct (Mesirov, 2010). However, 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” (Buckheit and Donoho, 1995; Donoho, 2010, 385). Separating the research from its advertisement makes it difficult for others to verify the research’s 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 your 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—to present your findings,
- the *knitr* package and other tie commands, which dynamically tie your data gathering, analysis, 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?

Research results are replicable if there is sufficient information available for independent researchers to make the same findings using the same procedures (King, 1995, 444). For research that relies on experiments, this can mean a

researcher not involved in the original research being able to rerun the experiment and validate that the new results match the original ones. In computational and quantitative empirical sciences results are replicable if independent researchers can recreate results 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 limited resources.¹ So as a next-best standard we can aim for “really reproducible research” (Peng, 2011, 1226).² In computational sciences³ 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 (Ball and Medeiros, 2011). 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.

In particular you will learn tools that allow us to dynamically “*knit*”⁴ the data and the source code you used for data gathering and analysis into 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 our results. This will make your computational research easily reproducible.

¹In this book we will actually aim for replicable research, even if we don’t always achieve it. New technologies make it possible to replicate a data set, especially if the data is available over the internet.

²The idea of really reproducible computational research was originally thought of and implemented by Jon Claerbout and the Stanford Exploration Project beginning in the 1980s and early 1990s (Fomel and Claerbout, 2009; Donoho et al., 2009). 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 (Buckheit and Donoho, 1995).

³Reproducibility is important for both quantitative and qualitative research (King et al., 1994). Nonetheless, we will focus mainly on methods for reproducibility in quantitative computational research.

⁴Much 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 Knuth, 1992, 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 and is used more generally to describe weaving with a variety of markup languages. Because of this, I use the term knit rather than weave in this book.

1.2 Why should research be reproducible?

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

1.2.1 For Science

Replicability has been a key part of scientific enquiry from perhaps the 1200s (Bacon, 1859; Nosek et al., 2012). It has even been called the “demarcation between science and non-science” (Braude, 1979, 2). Why is replication so important for scientific inquiry?

Standard to judge scientific claims

Replication, or at the least reproducibility, 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 these scientific claims is whether or not the claims can be replicated (Peng, 2011; Kelly, 2006). Research findings cannot even really be considered “genuine contribution[s] to human knowledge” until they have been verified through replication (Stodden, 2009b, 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”.⁵

Avoiding effort duplication & encouraging cumulative knowledge development

Not only is reproducibility crucial for evaluating scientific claims, it can also help enable the cumulative growth of future scientific knowledge (Kelly, 2006; King, 1995). Reproducible research cuts down on the amount of time scientists have to spend gathering data or 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 apply these data and procedures more quickly to building on established findings and developing new knowledge.

⁵See the American Physical Society's website at http://www.aps.org/policy/statements/99_6.cfm. See also Fomel and Claerbout (2009).

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 something like reading this book. But beyond the clear benefits for science, why should you make this effort? Using research reproducible 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 you data and source code up to a higher level of quality than you might if you “thought ‘no one was looking’” (Donoho, 2010, 386). This forces you to root out errors—a ubiquitous part of computational research—earlier in the research process (Donoho, 2010, 385). Clear documentation also makes it easier to find errors.⁶

Reproducible research needs to be stored so that other researchers can actually access the data and source code. By taking steps to make you research accessible for others you are also making it easier for you to find your data and methods when you revise your work or begin new projects. You avoid effort duplication; allowing you to cumulatively build on your 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. 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 (Donoho, 2010, 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 research*. Almost no actual research process is completely linear. You almost never gather data, run analyses, and present you results without going backwards to add variables, make changes to our statistical models, create new graphs, alter results tables in light of new findings, and so on. Often times you will 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

⁶Of course, reproducibility is “neither necessary nor sufficient to prevent mistakes” (Stodden, 2009a).

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 make these changes.

Creating dynamically reproducible documents in particular can make these changes 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 into 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 our research. If when you started the project you made it easier for others to reproduce your research, you also made it easier for to reproduce yourself. You will have taken steps to have a “better relationship with [your] future [self]” (Bowers, 2011).

Improve research impact

Reproducible research is more likely to be useful for other researchers than non-reproducible research. Useful research is cited more frequently (Vandewalle, 2012). Research that is fully reproducible contains more information, i.e. more reasons to use and cite, than research that merely presents findings. Independent researchers may use the data or code to look at other, often unanticipated, questions. For this 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” (2007, 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? (2009, 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 and the practical state-of-the-art computer tools to put it into practice. This includes 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 book will be helpful for people who already have general experience using technologies such as the R statistical programming language and LaTeX. The more researchers that start to incorporate the tools of reproducibility the better. So I also include enough information for people who have very limited experience with these tools. You will be able to start incorporating them into your workflow right away.

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 coherent 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 do. Forcing yourself to clearly document the steps you took will encourage you to think more clearly about what you are doing and reinforce what you are learning. It will also hopefully give you a greater appreciation of research accountability and integrity early in your careers (Barr, 2012; Ball and Medeiros, 2011, 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 courses they not only build students' understanding of research best practice, but are also better able to evaluate and provide meaningful feedback on students work (Ball and Medeiros, 2011, 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 (Peng, 2011). Hopefully, this book will be useful for editors at academic publishers who want to be better at evaluating reproducible research, edit it, and develop systems for making it more widely available. The journal *Biostatistics* is a good example of a publication that is encouraging (and requiring) reproducible research. From 2009 the journal has had an editor for reproducibility that ensures replication files are available and that results can actually be replicated using these files (Peng, 2009). 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. First, even if public reproducibility is ruled out to guard proprietary information,⁷ 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.

Also, 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

⁷There are ways to enable some public reproducibility without revealing confidential information. See Vandewalle et al. (2007) for a discussion of one approach.

that present the results and is easily linked to the reproducible research environment (Mesirov, 2010, 415).

In this book we will focus on learning how to use the widely available and highly flexible reproducible research environment—R/RStudio. R/RStudio can be linked to numerous reproducible research publishers such as LaTeX and Markdown with Yihui Xie’s *knitr* package (2012b). The main tools covered in this book include:

- **R:** a programming language primarily for statistics and graphics. It can also be used for data gathering and creating presentation documents.
- ***knitr*:** an R package for literate programming, i.e. it allows us to combine our statistical analysis and the presentation of the results into one document. It works 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 and Markdown.
- **RStudio:** an integrated developer environment (IDE) for R that tightly integrates R, *knitr*, and markup languages.
- **Cloud storage & versioning:** Services such as Dropbox and Github that can store data, code, and presentation files, save previous versions of these files, and make this information widely available.
- **Unix-like shell programs:** These tools are useful for working with large research projects.⁸ They also allow us to use command line tools including Pandoc, a program for converting documents from one markup language to another.

1.5 Why use R, knitr, and RStudio for reproducible research?

Why R?

Why use a statistical programming language like R (R Core Team, 2012) for reproducible research? R has a very active development community that is constantly expanding what R is capable of. As we will see in this book this enables researchers across a wide range of disciplines to gather data and run statistical analyses. Using the *knitr* package, you can connect your R-based analyses to presentation documents created with markup languages

⁸In this book I cover the Bash shell for Linux and Mac as well as Windows PowerShell

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 SPSS⁹ 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 2.

Why knitr?

Literate programming is a crucial part of reproducible quantitative research.¹⁰ Being able to directly link your analyses, your results, and the code you used to produce the results makes tracing your steps much easier. There are many different literate programming tools for a number of different programming languages. Previously, one of the most common tools for researchers using R and the LaTeX markup language was Sweave (Leisch, 2002). The package I am going to focus on in this book is newer and is called *knitr*. Why are we going to use *knitr* in this book and not Sweave or some other tool?

The simple answer is that *knitr* has the same capabilities as Sweave plus more. It can work with markup languages other than LaTeX¹¹ and can even work with programming languages other than R. It highlights R code in presentation documents making it easier for your readers to follow. Sweave doesn't. It gives you better control over the inclusion of graphics and can cache code chunks. It has the ability to understand Sweave-like syntax, so it will be easy to convert backwards to Sweave if you want to. You also have the choice to use much simpler and more straightforward syntax with *knitr*.

Why RStudio?

Why use the RStudio (RStudio, 2012) integrated development environment for reproducible research? R by itself has the capabilities necessary to gather

⁹I 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 just to write scripts that run analyses, create graphics, and deal with matrices.

¹⁰Donald 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 (Knuth, 1992).

¹¹It works with LaTeX, Markdown and reStructuredText. We cover the first two in this book.

data, analyse it, and, with a little help from *knitr* and markup languages, present results in a way that is highly reproducible. RStudio allows us to do all of these things, but simplifies many of them and allows you to navigate them in a more easily. It 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 stand alone editor for writing documents with LaTeX and Markdown. In LaTeX documents it can, for example, insert common commands like `\section*{}` for unnumbered sections (see Chapter 11).¹² 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.¹³ It's syntax highlighting can even distinguish between R code and markup commands in the same document. It can spell check LaTeX & Markdown documents. It handles *knitr* code chunks beautifully (see Chapter 3). Basically, RStudio makes it easy to create and navigate through complex documents.

Finally, RStudio not only has tight integration with various markup languages, it also has capabilities for using other tools such as 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 5). This is important for reproducible research since version control programs can document many of your research steps.

1.5.1 Installing the Software

Before you read this book you should install the software. All of the software programs covered in this book are open source and can be easily downloaded for free. They are available for Windows, Mac, and Unix. 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:** <http://www.r-project.org/>,
- **RStudio:** <http://rstudio.org/download/>.

The download webpages for these programs have comprehensive information on how to install them, so please refer to those pages for more information.

¹²If 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: <http://www.lyx.org/>.

¹³Syntax highlighting uses different colors and fonts to distinguish different types of text. For example in the PDF version of this book R commands are highlighted in **maroon**, while character strings are in **lavender**. Syntax highlighting can make source code easier to read.

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 see page xv.

Installing markup languages

If you are planning to create LaTeX documents you need to install a LaTeX distribution. They are available for Windows, Mac, and Unix. They can be found at: <http://www.latex-project.org/ftp.html>. Please refer to that site for more installation information.

If you want to create markdown documents you will need to install the *markdown* package in R. You can do this the same way that you install any package in R, with the `install.packages` command.¹⁴

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.

1.6.1 What this book is not.

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 introduction to R, RStudio, *knitr*, GitHub, the command line, 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 books that cover more of the nitty-gritty of R and the command line.

- Michael J. Crawley's encyclopaedic R book, appropriately titled, **The R Book** published by Wiley.
- Norman Matloff's tour through the programming language aspects of R called **The Art of R Programming: A Tour of Statistical Design Software** published by No Starch Press.
- For an excellent introduction to the command line in Linux and Mac, though with pretty clear implications for Windows users if they are running PowerShell (see Chapter 2) see William E. Shotts Jr.'s book **The**

¹⁴The exact command is: `install.packages("markdown")`.

Linux Command Line: A Complete Introduction also published by No Starch Press.

- The RStudio website (<http://rstudio.org/docs/>) has a number of useful tutorials on how to use *knitr* with LaTeX and Markdown.

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). I hope to save you from this wasted time by either providing a brief explanation of these possibly frustrating and mysterious conventions and/or pointing you in the direction of a good explanation.

1.6.2 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 maybe called *ResearchPaper* 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.

If you are experienced with R in particular you may want to skip over the first two sections of Chapter 3: Getting Started with R/RStudio. But don't skip over the whole chapter. The latter part contains important basic information on the *knitr* package.

1.6.3 How this book was written

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.

During the writing of this book, the repository is private and cannot be accessed publicly.

1.6.4 Contents overview

The book is broken into four parts. The first part of the book (chapters 2, 3, and 4) 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 guide you through the specific skills you will need for each part of the reproducible research process. The second part of the book (chapters 5, 6, and 7) covers the data gathering and file storage process. The third part (chapters 8, 9, and 10) teaches you how to dynamically incorporate your statistical analysis, results figures and tables into your presentation documents. The final part (chapters 11, 12, and 13) covers how to create reproducible presentation documents including LaTeX articles, books, slideshows and batch reports as well as Markdown webpages and slideshows.

2

Getting Started with Reproducible Research

Researchers often start thinking about making their research reproducible near the end of the research process when they write up the results. Or maybe later when a journal requires the data to 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 strewn across multiple folders on the researcher's computer. 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. Keeping your eye on reproducibility from the beginning of the research process and continuing to follow a few simple guidelines throughout your research can help solve these problems. Remember “reproducibility is not an afterthought—it is something that must be built into the project from the beginning” (Donoho, 2010, 386).

This chapter first gives you a brief big picture 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 projects are:

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

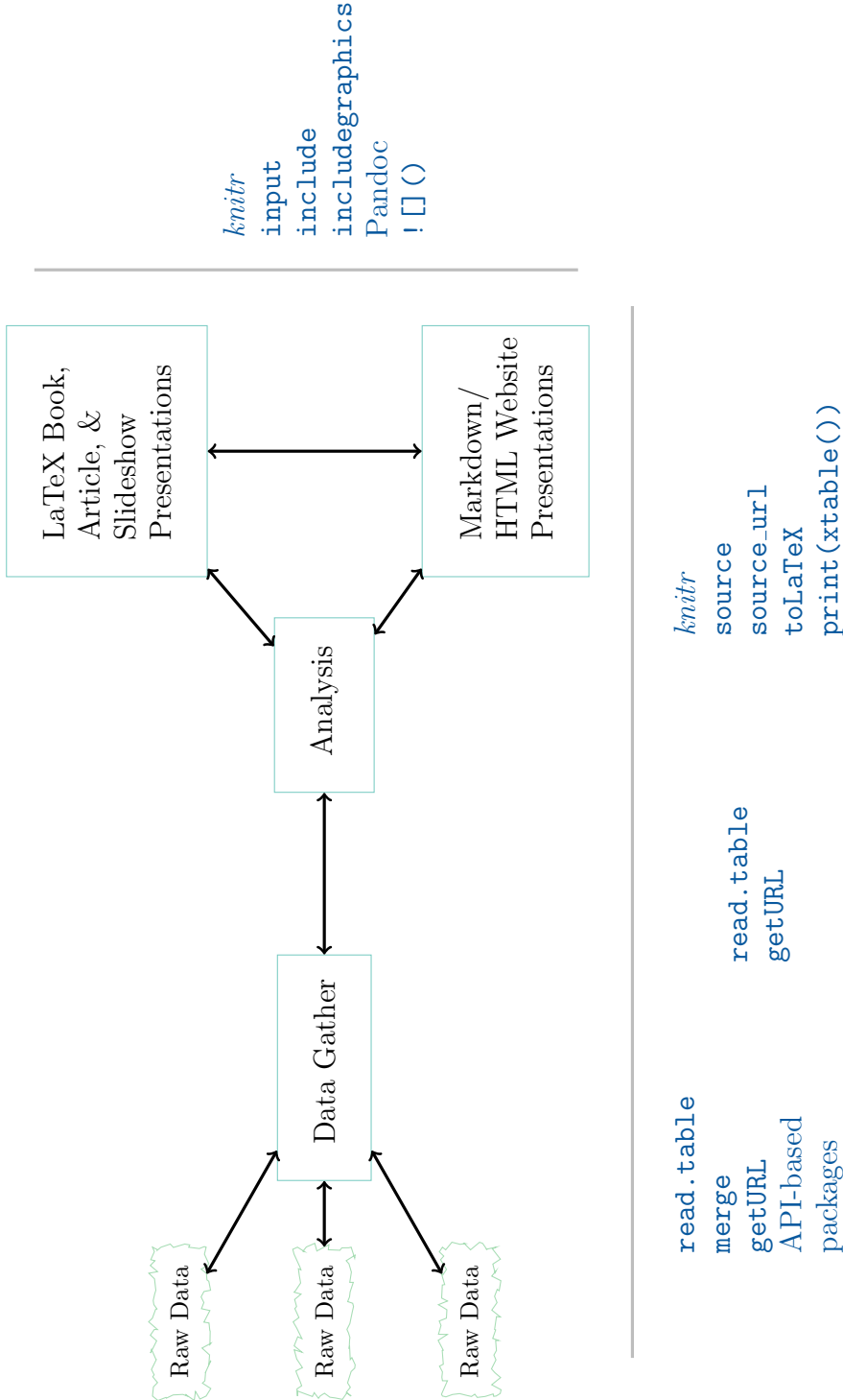
Each stage is part of the reproducible research workflow covered in this book. Tools for reproducibly gathering data are covered in Part II. Part III teaches tools for tying the data we gathered to our statistical analyses and

presenting the results with tables and figures. Part IV discusses how to tie these findings into documents you can use to advertise your findings.

If you're like me, you will probably be tempted to start using the individual tools of reproducible research with R/RStudio as soon as you learn them. However, I would recommend briefly stepping back and considering how the stages of reproducible research *tie* together overall. This will make your reproducible research workflow more coherent from the beginning and save you a lot of backtracking later on. Figure 2.1 illustrates the workflow. Notice that the arrows connecting the workflow's parts point in both directions, indicating that you should always be thinking 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 commands you will learn to make it easier to go forwards and backwards through the process. These commands tie your research together. For example, you can use API-based R packages to gather data from the internet. You can use the `merge` command to combine data gathered from different sources into one data set. The `and` and `getURL` commands can be used to bring this data set into your statistical analyses. The `knitr` package then ties your analyses into your presentation documents. This includes the code you used, the figures you created, and, with the help of commands such as `print` and the `xtable` 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 `` commands, respectively. This helps you maintain a consistent presentation of results across multiple documents types. We'll cover these commands in detail throughout the book. See Table 2.1 for a brief, but more complete overview of what I'll call *tie commands*.

FIGURE 2.1
Example Workflow & Commands to Tie it Together



2.1.1 Reproducible Theory

An important part of the research process that I do not actively discuss in this book is the theoretical stage. Ideally, if you are using a deductive research design, the bulk of this work will precede and guide the data gathering process. 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" (Vandewalle et al., 2007, 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 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 is useful to cover a few broad tips that will help you organize our research process and put these skills in perspective. The tips are:

1. Document everything!,
2. Everything is a file,
3. All files should be human readable,
4. Reproducible research projects are many files explicitly tied 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 its 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. Also, the way R functions and especially R packages are handled may vary across different operating systems. Finally, you may have R set to load packages by default (see page ?? for information about packages). These packages might be necessary to run your code, but other people might not what packages and what versions of the packages were loaded from just looking at your source code. The `sessionInfo` command in R prints a record of all of these things. The information from the session I used to create this book is:

```
sessionInfo()

## R version 2.15.1 (2012-06-22)
## Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)
##
## 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] splines    tools      stats      graphics  grDevices
## [6] utils      datasets  methods    base
##
## other attached packages:
## [1] survival_2.36-14    MSBVAR_0.7-2
## [3] mvtnorm_0.9-9992    bit_1.1-8
## [5] KernSmooth_2.23-8   reshape2_1.2.1
## [7] reshape_0.8.4       plyr_1.7.1
## [9] MCMCpack_1.2-4      coda_0.15-2
## [11] lattice_0.20-10     foreign_0.8-50
## [13] Zelig_3.5.5         boot_1.3-5
## [15] MASS_7.3-22         xtable_1.7-0
## [17] texreg_1.06         RCurl_1.95-0.1
## [19] bitops_1.0-4.1      openair_0.7-0
## [21] markdown_0.5.2      knitr_0.8
## [23] bibtex_0.3-2        knitr_0.8
## [25] ggplot2_0.9.2.1     extrafont_0.11
## [27] devtools_0.8        apsrtable_0.8-8
##
```

```
## loaded via a namespace (and not attached):
## [1] cluster_1.14.2      codetools_0.2-8
## [3] colorspace_1.1-1    dichromat_1.2-4
## [5] digest_0.5.2        evaluate_0.4.2
## [7] formatR_0.6         grid_2.15.1
## [9] gtable_0.1.1        httr_0.1.1
## [11] labeling_0.1         Matrix_1.0-9
## [13] memoise_0.1         mgcv_1.7-21
## [15] munsell_0.4         nlme_3.1-104
## [17] parallel_2.15.1     pkgmaker_0.8
## [19] proto_0.3-9.2       RColorBrewer_1.0-5
## [21] Rttf2pt1_1.1        scales_0.2.2
## [23] stringr_0.6.1       whisker_0.3-1
## [25] XML_3.95-0.1
```

Chapter 4 gives specific details about how to create files with dynamically included session information. Throughout the book you will learn how to fully document your research.

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 format¹ is the humble, but versatile, text file.²

Text files are extremely nimble. They can hold your data in, for example, comma-separated values (`.csv`) format. They can contain your analysis code in `.R` files. And they can be the basis for your presentations as markup documents like `.tex` or `.md`, 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`) documents change regularly and may not be compatible 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

¹Depending 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.

²Plain text files are usually given the file extension `.txt`.

files and, aided by frequent comments (see below), be able to understand how we conducted our research (Bowers, 2011, 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 track changes using programs such as Git (see Chapter 5).

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 we want done to do.³ 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 your source code files accessible to other people you need to *comment frequently* (Bowers, 2011, 3) and *format your code using a style guide* (Nagler, 1995). For especially important pieces of code you should use *literate programming*—where the source code and the presentation text appear in the same 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 (number, pound, or sharp) character⁴ 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 # 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 as usual and finds the answer 4, 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 mark-

³Of course, if it does not understand it will usually give us an error message.

⁴This is known as R’s comment declaration character.

down/HTML comments are placed inside of `<!-- -->`. The hash character is used for comment declaration in shell scripts.

Nagler (1995, 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 written code.

He 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 other contributors

You may also want to include other information in the header such as what other files it depends on, what output files it produces, and what version of the programming language you are using.

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”:

```
#####  
# Source code file used to create Figure 3 in 'My Article'  
# Created by Christopher Gandrud  
# Updated 15 October 2012  
#####
```

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 need to always follow a style guide. People could probably figure out what you are saying. But it would be 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 others to understand,
- it’s easier for you to understand.

There are a number of R style guides. Most of them are similar to the Google R Style Guide.⁵ Hadley Wickham also has a nicely presented R style guide.⁶ You can also use the *formatR* (Xie, 2012a) 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 (Knuth, 1992).

2.2.4 Reproducible research projects are many files explicitly tied 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 slideshow or 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 document. It will be very difficult for future researchers to trace the results table or figure back to a particular statistical model and a particular data set. Therefore, it is important to make the links between your files explicit.

Tie commands are the most dynamic way to explicitly link your files together. These commands instruct the computer program you are using to use information from another file. In Table 2.1 I have compiled a selection of key tie commands 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, & make your files available

Finally, in order for independent researchers to reproduce your research 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

⁵See: <http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html>.

⁶You can find it at <https://github.com/hadley/devtools/wiki/Style>.

TABLE 2.1

A Selection of Commands for Tying Together Your Research Files

Command/Package	Language	Description	Chapters for Further Information
<i>knitr</i>	R	R package with commands for tying analysis code into presentation documents including those written in LaTeX and Markdown.	Used throughout See Table 3.1.
<code>read.table</code>	R	Reads a table into R. You can use this to import plain-text file formatted data into R.	6
<code>read.csv</code>	R	Same as <code>read.table</code> with default arguments set to import .csv formatted data files.	6
API-based packages	R	Various packages use APIs to gather data from the internet.	6
<code>merge</code>	R	Merges together data frames.	7
<code>source</code>	R	Runs an R source code file.	8
<code>source_url</code>	R	From the <i>devtools</i> package. Runs an R source code file from a secure (https) url like those used by GitHub	8
<code>print(xtable())</code>	R	Combining the <code>print</code> & <code>xtable</code> commands creates LaTeX & HTML tables from R objects	9
<code>toLaTeX</code>	R	Converts R objects to LaTeX	2
<code>input</code>	LaTeX	Includes LaTeX files inside of other LaTeX files	12
<code>include</code>	LaTeX	Similar to <code>input</code> , but puts page breaks on either side of the <code>included</code> -ed text. Usually it is used for including chapters.	12
<code>includegraphics</code>	LaTeX	Inserts a figure into a LaTeX document.	10
<code></code>	Markdown	Inserts a figure into a Markdown document.	13
Pandoc	Shell	A Shell program for converting files from one markup language to another. Allows you to tie presentation documents together.	12 & 13

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 operations modular. One source code file should be used to complete one task. Breaking your operations into discrete parts will also make it easier for you and others to find errors (Nagler, 1995, 490).

Chapter 4 discusses file organization in much more detail. Chapter 5 teaches you a number of ways to make your files accessible through cloud computing services like Dropbox and GitHub.

3

Getting Started with R, RStudio, and knitr

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 in this book. For more detailed introductions to R please refer to the resources I mentioned in Chapter 1.6.1. 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* package, how to use it in R, and how it is 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 including:

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

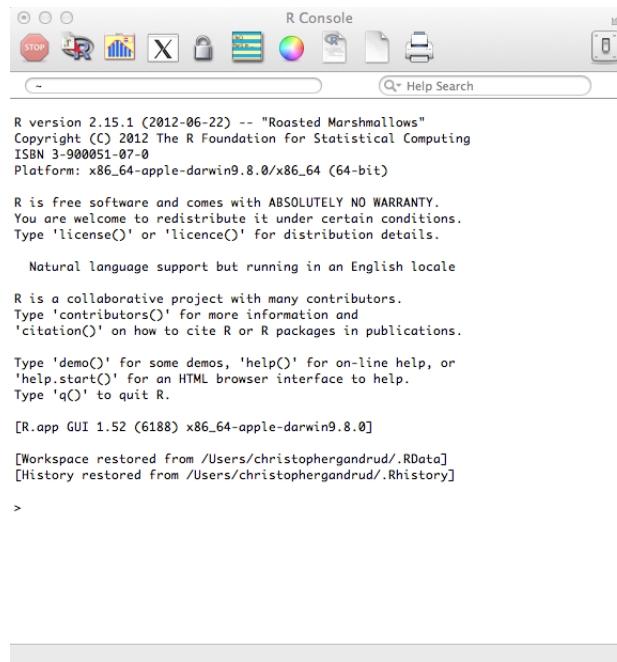
Before discussing each of these parts of the language let's open R and look around.¹ When you open R you should get a window that looks something like Figure 3.1.² This window is the **R console**. Under the startup information—information about what version of R you are using, license details, and so

¹Please see Chapter 1 for instructions on how to install R.

²This figure and almost all screenshots in this book were taken on a computer using the Mac OS 10.8 operating system.

on—you should see a `>`. This prompt is where you enter R code.³ To run R code that you have typed after the prompt hit the **Enter** or **Return** key. Now that we have a new R session open we can get started.

FIGURE 3.1
R Startup Console



3.1.1 Objects

If you’ve read a description of R, 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’ just means that R is focused on doing actions to objects. We will talk about the actions—commands and functions—later in this section. For now let’s create a few objects.

³If you are using a Unix-like system such as 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 `r` into the Terminal and then the **Enter** or **Return** key. This will begin a new R session. You know if a new R session has started if you get the same startup information is printed in the Terminal window.

Numeric & string objects

Objects can have a number of different data types. Let's make two simple objects. The first has a numeric type object. The other is a character. 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.⁴ Let's call our numeric object *Number*. To put something into the object we use the **assignment operator**⁵: `<-`. Let's assign the number 10 to our *Number* object.

```
Number <- 10
```

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

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

Let's briefly breakdown this output. 10 is clearly the contents of *Number*. The double hash (`##`) is included in the output here to tell you that this is output rather than R code.⁶ You probably did not get the double hash in your R output. Finally, `[1]` is the row number of the object that 10 is on. Clearly our object only has one row.

Creating a character object is very similar. The only difference is that you enclose the character string (letters in a word for example) inside of quotation marks (`"`). 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 we created earlier:

⁴It is common for people to use either periods (`.`) or capital letters (referred to as CamelBack) separate words in object names without using spaces. For example: *new.data* or *NewData* rather than *new data*.

⁵The assignment operator is sometimes also referred to as the 'gets arrow'.

⁶The double hash is generated automatically by *knitr*. It makes easier to copy and past code included in a presentation document by *knitr*.

```
mean(Words)

## Warning: argument is not numeric or logical: returning NA
## [1] NA
```

Trying to find the mean of our *Words* object gave us a warning message and returned the value NA: not applicable. You can also think of NA as meaning missing. To find out what type of object you have use the `class` command. 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.⁷ Clearly we want to be using objects—data sets—that have many strings and numbers. In R these are usually data frame type objects and are roughly equivalent 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 “workhorse” (Matloff, 2011). Knowing how to use vectors will be especially helpful when you clean up raw data in Chapter 7 and make tables in Chapter 9.⁸

Vectors:

Vectors are the “fundamental data type” in R (Matloff, 2011). They are simply an ordered group of numbers, character strings, and so on.⁹ It may be useful to think of basically all other data types as complicated forms 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.

⁷These might be called scalar objects, though in R scalars are just vectors with a length of 1.

⁸If you want information about other types of R objects such as lists and matrices, Chapter 1 of Norman Matloff’s book(2011) is a really good place to look.

⁹In 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.

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

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

# Show NumericVect's contents
NumericVect

## [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:

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

# Show CharacterVect's contents
CharacterVect

## [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 *NumericVect* and *CharacterVect* into a new object with the `cbind` function. This function binds the two vectors together side-by-side as columns.¹⁰

```
StringNumObject <- cbind(CharacterVect, NumericVect)

# Show StringNumObject's contents
StringNumObject

##      CharacterVect NumericVect
## [1,] "Albania"      "2.8"
## [2,] "Botswana"    "2"
## [3,] "Cambodia"    "14.8"
```

By binding these two objects together we've created a new matrix object.¹¹

¹⁰If you want to combine objects as if they were rows of the same column(s) use the `rbind` function.

¹¹Matrices are vectors with columns as well as rows.

You can see that the numbers in the *NumericVect* column are between quotation marks. Matrices, like vectors can only have one data type.

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` command to combine the *NumericVect* and *CharacterVect* objects.

```
StringNumObject <- data.frame(CharacterVect, NumericVect)

# Display contents of StringNumObject data frame
StringNumObject

##   CharacterVect NumericVect
## 1      Albania         2.8
## 2      Botswana         2.0
## 3      Cambodia        14.8
```

There are two 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. You will also notice that the strings in the *CharacterVect* object are no longer in quotation marks. This does not mean that they are somehow now numeric data. To prove this try to find the mean of *CharacterVect* by running it through the `mean` command like this:

```
mean(StringNumObject$CharacterVect)

## Warning: argument is not numeric or logical: returning NA

## [1] NA
```

3.1.2 Component Selection

The last bit of code will probably be confusing. Why do we have a dollar sign (\$) inbetween the name of our data frame object and the *CharacterVect* vector? The dollar sign is called the component selector.¹² It basically extracts a part

¹²It's also known as the element name operator.

of an object. In the previous example it extracts the *CharacterVect* column from the *StringNumObject* and feeds this to the `mean` command, which tries (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 *StringNumObject* and want an object with only the information in the numbers column. Let's use the following code:

```
NewNumeric <- StringNumObject$NumericVect  
  
# Display contents of NewNumeric  
NewNumeric  
  
## [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 9.

Using the component selector can lead to long repetitive code. You have to write the object name, a dollar sign every time you want to select a component. You can streamline your code by using commands such as `attach` and `with`. For examples in this book I largely avoid using these commands, instead using the component selector. Though it creates longer code, I find it easier to follow, because it's always clear which object we are selecting a component from.

3.1.3 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 cells. 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 the `head` command 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 commands:

```
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
```

3.1.4 Functions and commands

If objects are the nouns of the R language functions then commands¹³ are the verbs. They do things to objects. Let's use the `mean` command as an example. This command takes the mean of a numeric vector object. Remember our *NumericVect* object from before:

```
# Show contents of NumericVect  
NumericVect  
  
## [1]  2.8  2.0 14.8
```

To find the mean of this object simply type:

```
mean(x = NumericVect)  
  
## [1] 6.533
```

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

```
MeanNumericVect <- mean(x = NumericVect)
```

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

¹³For the purposes of this book I treat the two as the same.

3.1.5 Arguments

Arguments modify what commands do. In our most recent example we gave the `mean` command one argument (`x = NumericVect`) telling it that we wanted to find the mean of *NumericVect*. Arguments use the `ARGUMENTLABEL = VALUE` syntax.¹⁴

To find all of the arguments that an argument can accept look at the **Arguments** section of the command's help file. To access the help file type: `?COMMAND`. For example,

```
?mean
```

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

You have to be careful with the syntax for your argument's values. Arguments for logical arguments must be written as `TRUE` or `FALSE`.¹⁵ Arguments that are character strings should be in quotation marks.

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

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

## [1] 6.5
```

You can see that arguments are separated by commas.

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

```
# Find mean of NumericVect
mean(NumericVect)

## [1] 6.533
```

¹⁴Note: 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.

¹⁵They can be abbreviated `T` and `F`.

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

Finally, you can stack arguments inside of other arguments. To have R find the mean of `NumericVect` and round it to one decimal place use:

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

## [1] 6.5
```

3.1.6 The Workspace & History

All of the objects you create become part of your workspace. Use the `ls` command to list all of the objects in your current workspace.

```
ls()

## [1] "CharacterVect" "Data" "DeleteFiles"
## [4] "doInstall" "MeanNumericVect" "NBModel"
## [7] "NBModelSum" "NBSumDataFrame" "NBTable"
## [10] "NewNumeric" "Number" "NumericVect"
## [13] "PackagesCited" "ParentDirectory" "Plot"
## [16] "StringNumObject" "temp" "toInstall"
## [19] "url" "URL" "UrlAddress"
## [22] "Words"
```

To save the workspace into an `.RData` file use the `save.image` command. The main argument of the `save.image` command is the file path you would like the file saved into. If you don't specify the file path it will save in your current working directory (see Chapter 4).

You should generally avoid saving your workspace. 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.¹⁶ The only time when

¹⁶For 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 commands.

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 it is still useful to clean up the workspace before saving it by using the `rm` command to remove the other objects. For example, to remove the `CharacterVect` and `Words` objects type:

```
rm(CharacterVect, Words)
```

When you enter a command into R they become part of your history. To see the most recent commands in your history use the `(history)` command. 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.

3.1.7 Installing new libraries and loading commands

Commands are stored in R libraries. R automatically loads a number of basic libraries by default. One of the great things about R is the many user-created libraries¹⁷ that greatly expand the number of commands we can use. To install commands that do not come with base R you need to install the add-on packages that contain them. To do this use the `install.packages` command. By default this command downloads and installs the packages from the Comprehensive R Archive Network (CRAN).

For the code you need to install all of the package libraries used in this book see page xv. When you run that code on xv, you will likely 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` command to load a package library. Use the following code to load the `ggplot2` library that we use in Chapter 10 to create figures.

```
library(ggplot2)
```

Please note for the examples in this book I only specify the library a command is in if the library is not loaded by default when you start an R session.

¹⁷For the latest list see: http://cran.r-project.org/web/packages/available_packages_by_name.html

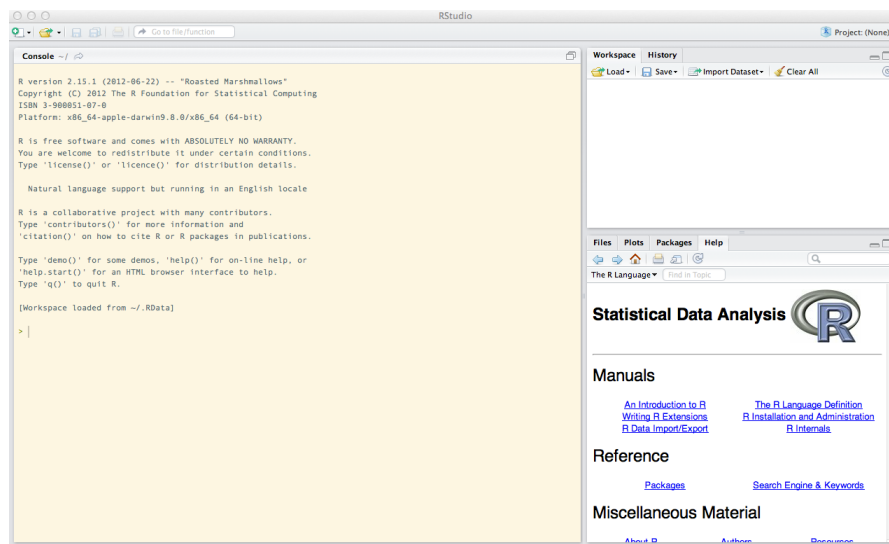
3.2 Using RStudio

As I mentioned in Chapter 1, 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.

The default window

When you first open RStudio you should see a default window that looks like Figure 3.2. 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 *Workspace/History* panes, usually in the upper right-hand corner. The Workspace pane shows you all of the objects in your current 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 commands you have run. It allows you to rerun code and insert it into a source code file.

FIGURE 3.2
RStudio Startup Panel



In the lower right-hand corner you will see the *Files/Plots/Packages/Help* pane. We will discuss the Files pane in more detail in Chapter 4. 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 shows the packages you have installed, allows you to load individual packages by clicking on the dialog box next to them, access their manual files (click on the package name), update the packages, and even install new packages. Finally, the Help pane shows you help files. You can search for help files and search within help files using this pane.

The source pane

There is an important pane that does not show up when you open RStudio for the first time. This is the *Source* pane. The Source pane is where you create, edit, and run your source code files. It also functions as an editor for your markup files. It is the center of reproducible research in RStudio. Let's first look at how to use the Source pane with regular R files. We will cover how to use the Source pane with literate programming file formats—e.g. R Markdown and R LaTeX—in more detail after first discussing the *knitr* basics in the next section.

R source code files have the file extension `.R`. You can create a new source code document, which will open a new Source pane, by going to the menu bar and clicking on **File** → **New**. In this drop down menu you have the option to create a variety of different source code documents. Select the **R Source** option. You should now see a new pane with a bar across the top that looks like the first image in Figure 3.3. To run the R code you have in your source code file simply highlight it¹⁸ and click the **Run** icon on the top bar. This sends the code to the console where it is executed. The icon to the right of **Run** simply runs the code above where you have highlighted. The **Source** icon next to this runs all of the code in the file using R's `source` command. The icon next to **Source** is for compiling RStudio Notebooks. We will look at RStudio Notebooks later in this chapter.

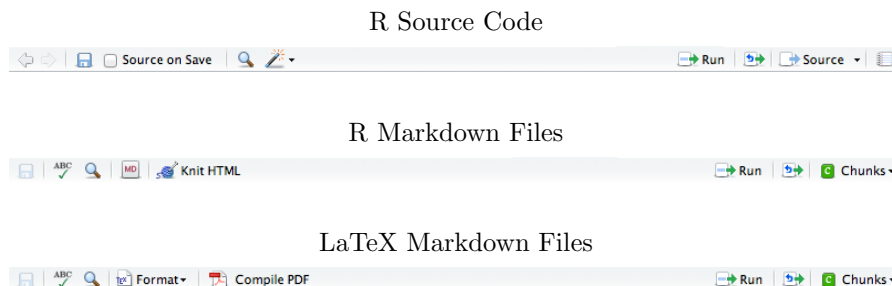
3.3 Using knitr: the basics

To get started with *knitr* in R or RStudio we need to learn some of the basic concepts and syntax. The concepts are the same regardless of the markup language we are knitting R code with, but much of the syntax varies.

¹⁸If you are only running one line of code you don't need to highlight the code, you can simply put your cursor on that line.

FIGURE 3.3

RStudio Source Code Pane Top Bars



3.3.1 File extensions

When you save a knitable file use a file extension that indicates (a) that it is knitable and (b) what markup language it is using. You can use a number of file extensions for R Markdown files including: `.Rmd` and `.Rmarkdown`. LaTeX documents that include *knitr* code chunks are generally called R Sweave files and have the file extension `.Rnw`. This terminology is a little confusing. It is a holdover from *knitr*'s main literate programming predecessor *Sweave* (Leisch, 2002). You can also use the less confusing file extension `.Rtex`, as regular LaTeX files have the extension `.tex`. However, the syntax for `.Rtex` files is different from that used with `.Rnw` files. We'll look at this issue in more detail below.

3.3.2 Code Chunks

When we want to include R code into our markup presentation documents we place them in a code chunk. Code chunk syntax differs depending on the markup language we are using to write our documents. Let's see the syntax for R Markdown and R LaTeX files.

R Markdown

In R Markdown files we begin a code chunk by writing the head: ````{r}`. A code chunk is closed—ended—simply with: `````. For example:

```
```{r}
Example of a R Markdown code chunk
```



```
StringNumObject <- cbind(CharacterVect, NumericVect)
```

```

R LaTeX

There are two different ways to delimit code chunks in R LaTeX documents. One way largely emulates the established *Sweave* syntax.¹⁹ *Knitr* also supports files with the *.Rtex* extension, though the code chunk syntax is different. I will cover both types of syntax for code chunks in LaTeX documents, though in throughout the book I use the older and more established *Sweave* style syntax.

Sweave-style

Traditional Sweave-style code chunks begin with the following head: `<<>>=`. The code chunk is closed with an at sign (`@`).

```
<< >>=
# Example of a Sweave-style code chunk
StringNumObject <- cbind(CharacterVect, NumericVect)
@
```

Rtex-style

Sweave-style code chunks may seem fairly baroque. Another option for LaTeX files is the Rtex-style syntax. To begin a code chunk simply type double percent signs followed by `begin.rcode`, i.e. `%% begin.rcode`. To close the chunk you use double percent signs: `%%`. Each line in the code chunk needs to begin with a single percent sign. For example:

```
%% begin.rcode
% # Example of a Rtex-style code chunk
% StringNumObject <- cbind(CharacterVect, NumericVect)
```

¹⁹The syntax has its genesis in a literate programming tool called *noweb* (Leisch, 2002; Ramsey).

```
%%
```

Code chunk labels

Each chunk has a label. When a code chunk creates a plot or the output is cached—stored for future use—*knitr* uses the chunk label for the new file’s name. If you do not explicitly give the chunk a label it will be assigned one like: **unnamed-chunk-i**. *i* is the chunk’s number.

To explicitly assign chunk labels in R Markdown documents place the label name inside of the braces after the **r**. If we wanted to use the label **ChunkLabel** we would simply type:

```
```{r ChunkLabel}  
Example chunk label
```
```

The same general format applies to the two types of LaTeX chunks. In Sweave-style chunks we would type: `<<ChunkLabel>>=`. In Rtex-style we use: `%
begin.rcode ChunkLabel`.

Try not to use spaces or periods in your label names. And all chunk labels *must* be unique.

Code chunk options

There are many times when we want to change how our code chunks are knitted and presented. Maybe we only want to show the code and not the results or perhaps we don’t want to show the code at all but just a figure that it produces. Maybe we want the figure to be formatted in a certain way. To make these changes, and many others we can specify code chunk options.

Like chunk labels, you specify options in the chunk head. Place them after the chunk label, separated by a comma. Chunk options are written following pretty much the same rules as regular R command arguments. They have a similar **OPTIONLABEL=VALUE** structure as arguments. The option values must be written in the same way that argument values are. Character strings need to be inside of quotation marks. The logical **TRUE** and **FALSE** operators cannot be written “true” and “false”. For example, imagine we have a Markdown code chunk called **ChunkLabel**. If we only want to have *knitr* include the code in our document, but not actually run it we use the option **eval=FALSE**. This option tells *knitr* not to evaluate (run) the code chunk.

```
```{r ChunkLabel, eval=FALSE}
Example of a non-evaluated code chunk
StringNumObject <- cbind(CharacterVect, NumericVect)
```
```

Note that all labels and code chunk options must be on the same line. Options are separated by commas. The syntax for *knitr* options is the same regardless of the markup language. Here is the same chunk option in Rtex-style syntax:

```
%% begin.rcode ChunkLabel, eval=FALSE
% # Example of a non-evaluated code chunk
% StringNumObject <- cbind(CharacterVect, NumericVect)
%%
```

Throughout this book we will look at a number of different code chunk options. All of the chunk options we will use in this book are listed in Table 3.1. For the full list of *knitr* options see the *knitr* chunk options page maintained by *knitr*'s creator Yihui Xie: http://yihui.name/knitr/options#package_options.

Note:
this table will be expanded as the later chapters are expanded.

3.3.3 Global options

So far we have only looked at how to set local options in *knitr* code chunks, i.e. options for only one specific chunk. If we want an option to apply to all of the chunks in our document we can set **global chunk options**. Options are 'global' in the sense that they apply to the entire document. Setting global chunk options helps us create documents that are formatted consistently without having to repetitively specify the same option every time we create a new code chunk. For example, in this book I center almost all of the figures. Instead of using the `fig.align='center'` option in each code chunk that creates a figure, I set the option globally.

To set a global option first create a new code chunk at the beginning of your document²⁰ You will probably want to set the option `echo=FALSE` so that *knitr* doesn't echo the code. Inside the code chunk use `opts_chunk$set`. You can set any chunk option as an argument to `opts_chunk$set`. The option will be applied across your document, unless you set a different local option.

Here is an example of how we can center align all of the figures in a

²⁰In Markdown, you can put global chunk options at the very top of the document. In LaTeX they should be placed after the `\begin{document}` command (see Chapter 11 for more information on how LaTeX documents are structured).

TABLE 3.1A Selection of *knitr* Code Chunk Options

| Chunk Option Label | Type | Description |
|-------------------------|-----------|---|
| <code>eval</code> | Logical | Whether or not to run the chunk. |
| <code>echo</code> | Logical | Whether or not to include the code in the presentation document. |
| <code>error</code> | Logical | Whether or not to include errors. |
| <code>engine</code> | Character | Set the programming language for <i>knitr</i> to evaluate the code chunk with |
| <code>warning</code> | Logical | Whether or not to include warnings. |
| <code>message</code> | Logical | Whether or not to include message messages. |
| <code>fig.align</code> | Character | Aligns figures. |
| <code>fig.height</code> | Numeric | Sets figures' height. |
| <code>fig.width</code> | Numeric | Sets figures' width. |

Markdown document created *knitr* code chunks. We place the following code at the beginning of the document:

```
```{r GlobalFigOpts, echo=FALSE}
Center align all knitr figures
opts_chunk$set(fig.align='center')
```
```

3.3.4 knitr package options

knitr package options affect how the *knitr* package itself runs. For example, the `progress` option can be set as either `TRUE` or `FALSE`²¹ depending on whether or not we want a progress bar to be displayed when we knit a code chunk.²² You can use `base.dir` to set the directory where you want all of your figures to be saved to (see Chapter 4) or the `child.path` option to specify where child documents are located (see Chapter 12).

²¹It's set as `TRUE` by default.

²²The *knitr* progress bar looks like this `|>>>>>| 100%` and indicates how much of a code chunk has been run.

We set package options in a similar way to global chunk options with `opts_knit$set`. For example, to turn off the progress bar when knitting Markdown documents include this code at the beginning of the document:

```
```{r GlobalFigOpts, echo=FALSE}
Turn off knitr progress bar
opts_knit$set(progress=FALSE)
```
```

3.3.5 Hooks

You can also set chunk hooks. Hooks come in two types: chunk hooks and output hooks. Chunk hooks run a function before or after a code chunk. Output hooks change how the raw output is formatted. I don't cover hooks in much detail in this book. For more information on hooks, please see Yihui Xie's webpage: <http://yihui.name/knitr/hooks>.

3.3.6 knitr & RStudio

RStudio is highly integrated with *knitr* and the markup languages *knitr* works with. Because of this integration it is easier to create and compile *knitr* documents than doing so in plain R. Most of the RStudio/*knitr* features are accessed in the Source pane. The Source pane's appearance and capabilities changes depending on the type of file you have open in it. RStudio uses the file extension to automatically determine what type of file you have open.²³ We have already seen some of the features the Source pane has for R source code files. Let's now look at how to use *knitr* with R source code files as well as the markup formats we cover in this book: R Markdown, and R LaTeX.

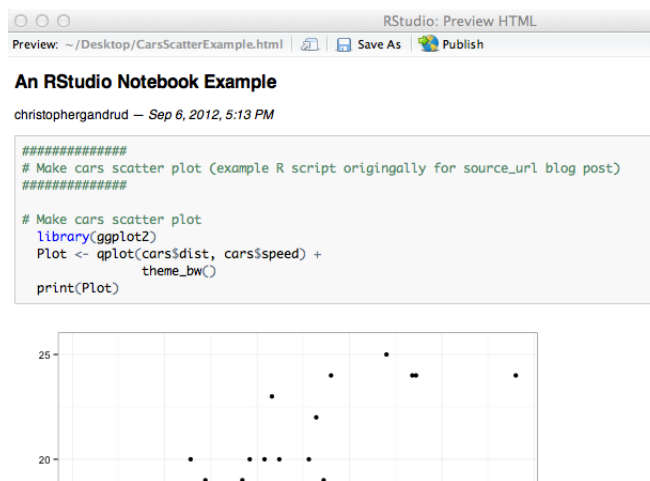
Compiling R source code notebooks

If you want a quick well formatted account of the code that you ran and the results that you got you can use RStudio's "Compile Notebook" capabilities. RStudio uses *knitr* to create a standalone HTML file that includes all of the code from an R source file as well as the output. This can be useful for recording the steps you took to do an analysis. You can see an example RStudio Notebook in Figure 3.7.

If you want to create a Notebook from an open R source code file simply

²³You can manually set how you want the Source pane to act by selecting the file type using the drop down menu in the lower right-hand corner of the Source pane.

FIGURE 3.7
RStudio Notebook Example



click the **Compile Notebook** icon in the Source pane's top bar (see Figure 3.3).²⁴ Then click the **Compile** button in the window that pops up. In Figure 3.7 you can see near the top center right a small globe icon next to the word "Publish". Clicking this allows you to publish your Notebook to RPubS (<http://www.rpubs.com/>). RPubS allows you to share your Notebooks over the internet. You can publish not only Notebooks, but also any *knitr* Markdown document you compile in RStudio.

R Markdown

The second image in figure 3.3 is what the Source pane's top bar looks like when you have an R Markdown file open. You'll notice the familiar Run button for running R code. At the far right you can see a new **Chunks** drop down menu. In this menu you can select **Insert Chunk** to insert the basic syntax required for a code chunk. There is also an option to **Run Current Chunk**—i.e. the chunk where your cursor is located—**Run Next Chunk**, and **Run All** chunks. You can navigate to a specific chunk using a drop down menu in the bottom left-hand side of the Source pane (not shown). This can be very useful if you are working with a long document. To knit your file click the **Knit HTML** icon on the left side of the Source pane's top bar. This will create a knitted HTML file as well as a regular Markdown file with highlighted code, output, and figures in your R Markdown's directory. Other useful features in the R

²⁴Alternatively, **File** → **Compile Notebook...**

Markdown Source pane’s top bar include the **ABC** spell check icon and **MD** icon, which gives you a Markdown syntax reference file in the Help pane.

Another useful RStudio *knitr* integration feature is that RStudio can properly highlight both the markup language syntax and the R code in the Source pane. This makes your source code much easier to read and navigate. RStudio can also fold code chunks. This makes navigating through long documents, with long code chunks, much easier. In the first image in Figure 3.8 you can see a small downward facing arrow at line 25. If you click this arrow the code chunk will collapse, like in the second image in Figure 3.8. To unfold the chunk, just click on the arrow again.

You may also notice that there are code folding arrows on lines 27 and 34 in the first image. These allow us to fold parts of the code chunk. To enable this option create a comment line with at least one hash before the comment text and at least four after like this:

```
#### An RStudio Foldable Comment ####
```

You will be able to fold all of the text after this comment up until the next similarly formatted comment (or the end of the chunk).

R LaTeX

You can see in the final image in Figure 3.3 that many of the Source pane options for R LaTeX files are the same as R Markdown files. The key differences being that there is a **Compile PDF** icon instead of **Knit HTML**. Clicking this icon knits the file and creates a PDF file in your R LaTeX file’s directory. There is also a **Format** icon next instead of **MD**. This actually inserts LaTeX formatting commands into your document for things such as section headings and bullet lists. These can be very tedious to type out by hand.

Change default .Rnw knitter

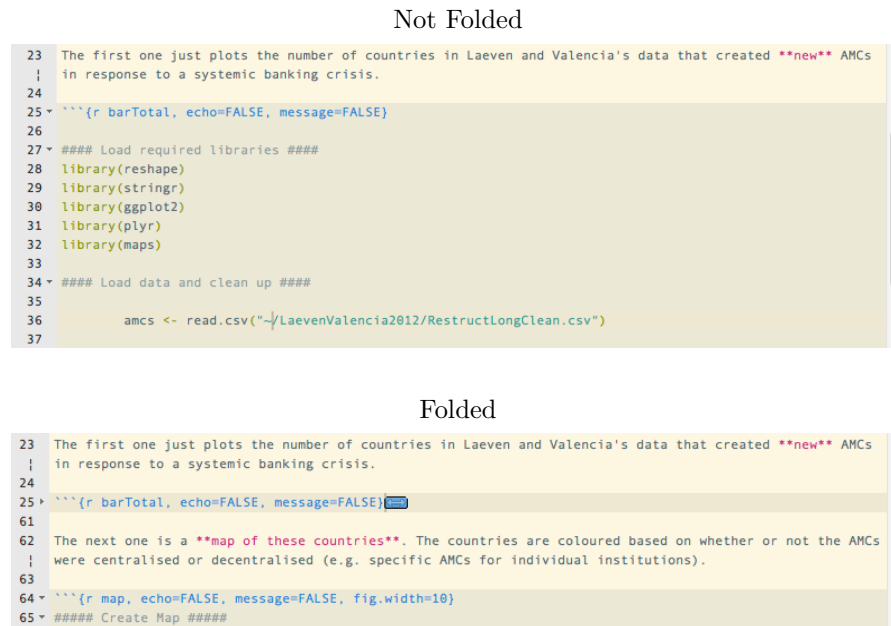
By default RStudio is set up to use *Sweave* for compiling LaTeX documents. To use *knitr* instead of *Sweave* when knitting .Rnw files in RStudio you should go to the **Options** window. Click on the **Sweave** button. Select **knitr** from the drop down menu for “Weave files using:”. Finally, click **Apply**. The **Options** window can be opened by clicking on **Tools** in the menu bar then **Options**.

3.3.7 knitr & R

As *knitr* is a regular R package, you can of course knit documents in regular R (or using the console in RStudio). All of the *knitr* in your markup document is the same as before. But instead of clicking a **Compile PDF** or **knit HTML** button use the **knit** command. To knit our example Markdown file *Example.Rmd*

FIGURE 3.8

Folding Code Chunks in RStudio



we first set the working directory (see Chapter 4) to the the folder where your *Example.Rmd* file is located with the `setwd` command. In this example I have it on my desktop.²⁵

```
setwd("~/Desktop/")
```

Then I knit my file:

```
knit(input = "Example.Rmd", output = "Example.md")
```

You use the same steps for all other knitable document types. Note that if you do not specify the output file *knitr* will determine what the file name and

²⁵Using the directory name `~/Desktop/` is for Mac computers. Please use alternative syntax discussed in Chapter 4 on other types of systems.

extension should be. In this example it would come up with the same name and location as I gave it.

If you try this example, you find that the *knit* command only created a Markdown file and not an HTML file like clicking the RStudio **knit** HTML did. Likewise, if you use **knit** on a *.Rnw* file you will only end up with a basic LaTeX *.tex* file and not a compiled PDF. To convert the Markdown file into HTML you need to further run the *.md* file through the **markdownToHTML** command from the *markdown* package, i.e.

```
markdownToHTML(file = "Example.md", output = "Example.html")
```

This is a bit tedious. Luckily, there is a command in the *knitr* package that combines **textttmarkdownToHTML** and **knit**. It is called **knit2html**. You use it like this:

```
knit2html(file = "Example.Rmd", output = "Example.html")
```

If we want to compile a *.tex* file in R we run it through the **texi2pdf** command in the *tools* package. This package will run both LaTeX and to create a PDF with a bibliography (see Chapter 11 for more details on using for bibliographies). Here is a **texi2pdf** example:

```
# Load tools package
library(tools)

# Compile pdf
texi2pdf(file = "Example.tex")
```

Just like with **knit2html**, you can speed up this process by using the **knit2pdf** command to compile a PDF file from a *.Rnw* or *.Rtex* document.

4

Getting Started with File Management

Careful file management is crucial for reproducible research. Remember two of the guidelines from Chapter 2:

- Reproducible research projects are many files explicitly tied together,
- Have a plan to organize, store, and make your files available.

Apart from the fleeting situations where you have an email exchange (or even meet in person) with someone interested in reproducing your research, the main information independent researchers have about the procedures you used will be stored across many files: data files, analysis files, and presentation files. If these files are well organized and the links tying them together are clear, replication will be much easier. File management is also important for you as a researcher, because if your files are well organized you will be able to more easily make changes, collaborate with others, and so on.

Using tools such as R, *knitr*, and markup languages like LaTeX requires fairly detailed knowledge of where files are stored—their **file path**—in your computer. Handling files reproducibly also requires us to use command line tools to access and organize our files. R and Unix-like shell programs allow us to control files—creating, deleting, moving them—in powerful and really reproducible ways. By typing these commands we are documenting every step we take. This is a major advantage over graphical user interface type systems where we organize files by clicking and dragging them with the cursor. However, text commands require us to know our files’ specific address—it’s file path.

In this chapter we discuss how a reproducible research project may be organized and cover the basics of file path naming conventions in Unix, Mac, and Windows systems. We then learn organize them with RStudio Projects. Finally, we will cover some basic R and Unix-like shell commands for manipulating files as well as how to navigate through files in RStudio in the **Files** pane and. The skills we learn in this chapter will be heavily used in the next chapter (Chapter 5) and throughout the book.

In this chapter we work with locally stored files, i.e. files stored on your computer. In the next chapter we will discuss various ways to store and access files remotely in the cloud.

4.1 File paths & naming conventions

All of the operating systems covered in this book organize files in hierarchical directories (or file trees). To a large extent, ‘directories’ can be thought of as the folders you usually see on your Windows or Mac desktop.¹ They are called ‘hierarchical’ because directories are located inside of other directories, like we will see in Figure 4.1.

4.1.1 Root directories

A root directory is the first level in a disk, such as a hard drive. It is the root out of which the file tree ‘grows’. All other directories are subdirectories of the root directory.²

On Windows computers the root directory is given a drive letter assignment. If you use Windows regularly you will most likely be familiar with the `C:\` used to denote the C partition of the the hardrive. This is a root directory. On Unix-like systems, including Mac computers. The root directory is simply labeled `/` with nothing before it.

4.1.2 Subdirectories

You will probably not store all of your files the root directory. This would get very messy. Instead you will likely store your files in subdirectories of the root directory. Inside of these subdirectories may be further subdirectories and so on. The directory a subdirectory is inside of is called the parent directory.

On Windows computers separate subdirectories are indicated with a back slash (`\`). For example if we have a folder called *Data* inside of a folder called *ExampleProject* which is located in the C root directory it has the address `C:\ExampleProject\Data`.³ When you type Windows file paths into R you need to use two backslashes rather than one: `C:\\ExampleProject\\Data`. This is because the `\` is an escape character in R.⁴ Escape characters tell R to interpret the next character or sequence of characters differently. For example, on page 67 you’ll see how `\t` can be interpreted by R as a tab. To neutralize the escape character simply add another escape character, i.e. escape the escape character. In R this simply means using a double backslash: `\\`. Another option for writing Windows file names is to use one forward slash (`/`).

On Unix-like systems, including Mac computers, directories are indicated

¹To simplify things, I use the terms ‘directory’ and ‘folder’ interchangeably in this book.

²On Windows computers you can have multiple root directories, one for each storage devise or partition of a storage device.

³For more information on Windows file path names see this helpful website: [http://msdn.microsoft.com/en-us/library/windows/desktop/aa365247\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365247(v=vs.85).aspx)

⁴As we will see in Part IV, it is also a LaTeX escape character.

with a forward slash (/). The file path of the *Data* file on a Unix-like system, for example: `/ExampleProject/Data`

In the book I switch between the two file system naming conventions to expose you to both.

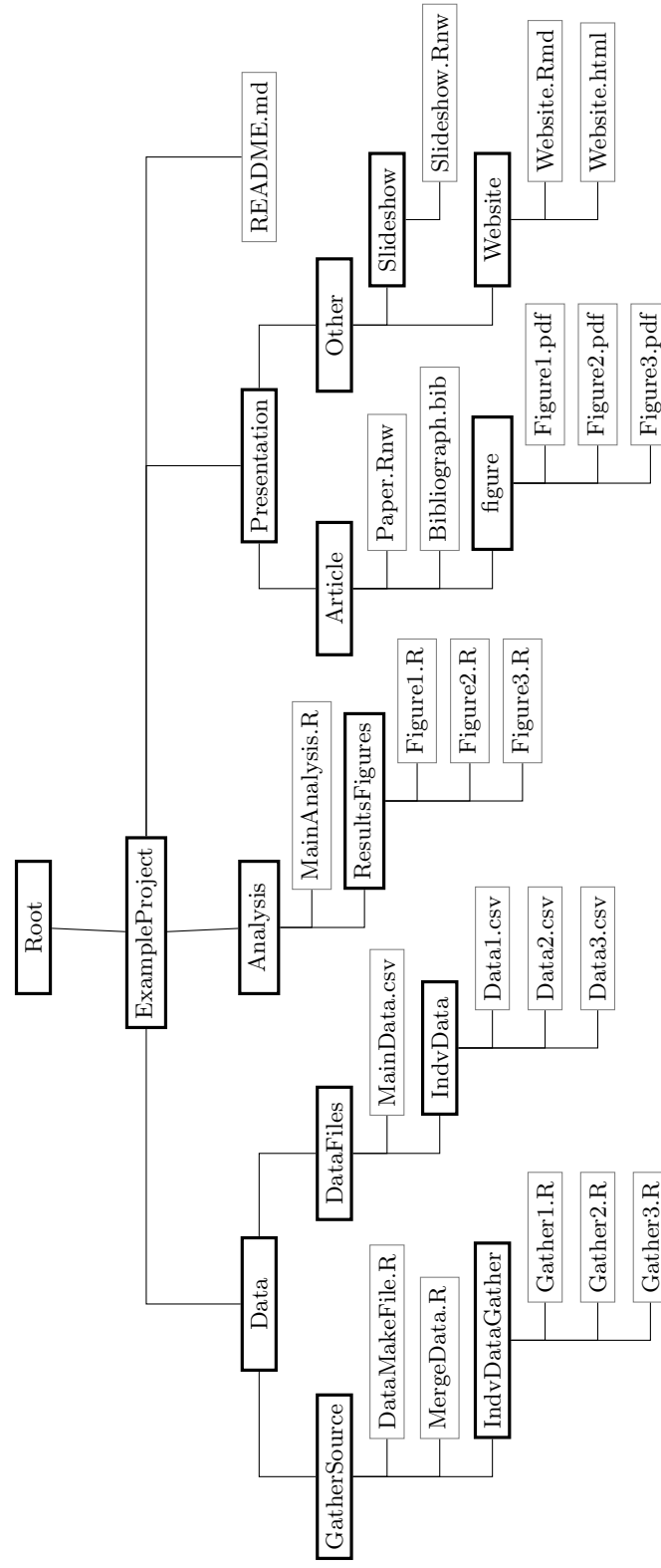
4.1.3 Working directories

When you use R, the command line, and markup languages it is important to keep in mind what your current working directory is. The working directory is the directory where the program automatically look for files and other directories. It is also where it will save files. Later in this chapter we will cover commands for handling the working directory.

4.2 Organizing your research project

Figure 4.1 gives an example of how the files in a simple reproducible research project could be organized. The project's main parent directory is called *Example Project*. Inside this directory are three subdirectories: a data gathering directory, an analysis directory, and a presentation directory. Each of these directories contain further subdirectories and files. The *Presentation* directory for example contains subdirectories for files that present the findings in article, slideshow, and website formats.

FIGURE 4.1
Example Research Project File Tree



In addition to this main folders you will probably notice a file called *README.md*. The *README.md* file gives an overview of all the files in the project. It should briefly describe the project including things such as its title, author(s), topic and so on. It should also indicate how the folders in the project are organized and give instruction for how to reproduce the project. The README file should be in the main project folder—in our example this is called *ExampleProject*—so that it is easy to find. If you are storing your project as a GitHub repository (see Chapter 5) and the file is called **README** its contents will automatically be displayed on the repository’s main page. If it is written using Markdown, it will also be properly formatted.

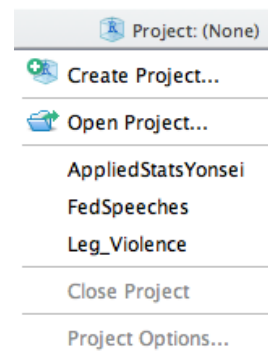
It is good practice to dynamically include the system information for the R session you used to create the project. To do this you can write your readme file with R Markdown (see Chapter 12). Simply include the `sessionInfo()` command in a code chunk in the R Markdown document. If you knit this file immediately after knitting your presentation document it will record the information for that session.

You can also dynamically include session info in a LaTeX document. To do this simply use the `toLatex` command in a code chunk. The code chunk should have the option `results='asis'`. The code is:

```
toLatex(sessionInfo())
```

FIGURE 4.2

An Example RStudio Project Menu



4.3 Setting directories as RStudio Projects

If you are using RStudio, you may want to organize your files in Projects. You can turn a normal directory into an RStudio Project by clicking on **Project** in the RStudio menu bar and selecting **Create Project...** A new window will pop up. Select the option **Existing Directory**. Find the directory you want to make into an RStudio Project by clicking on the **Browse** button. Finally, select **Create Project**. You will also notice in the Create Project pop up window that you can build new project directories and also create a project

from a directory already under version control (we'll do this in Chapter 5). When you create a new project you will see that RStudio has put a file with the extension `.Rproj` into the directory.

Making your research project directories RStudio Projects is useful in a number of ways:

- when you open the `.Rproj` file RStudio automatically sets the working directory to the Project's directory and loads the project workspace, history, and source code files you were last working on.
- you can set project specific options like whether PDF presentation documents should be compiled with Sweave or *knitr*.
- when you close the project your R workspace and history are saved in the project directory,
- helps you version control your files
- the project is listed in RStudio's Project menu where it can be opened easily (see Figure 4.2).

4.4 R file manipulation commands

R has an a range of commands for handling and navigating through files. Including these commands in our source code files allows you to more easily replicate your actions.

`getwd`

To find out what current working directory R use the `getwd` command:

```
getwd()

## [1] "/git_repositories/Rep-Res-Book/Source/Children/Chapter4"
```

The example here shows you the current working directory that was used while knitting this chapter.

`list.files`

Use the `list.files` command to see all of the files and subdirectories in the current working directory. You can list the files in other directories too by adding the directory path as an argument to the command.

setwd

The **setwd** command sets the current working directory. For example, if we are on a Mac or other Unix-like computer we can set the working directory to the *GatherSource* directory in our Example Project (see Figure 4.1) like this

```
setwd("/ExampleProject/Data/GatherSource")
```

Now R will automatically look in the *GatherSource* folder for files and will save new files into this folder, unless we explicitly tell it to do otherwise.

dir.create

Sometimes we may want to create a new directory. We can use the **dir.create** command to do this.⁵ For example to create a *ExampleProject* file in the root C directory on a Windows computer type:

```
dir.create("C:\\ExampleProject")
```

file.create

Similarly, you can create a new blank file with the **file.create** command. To add a blank R source code file called *SourceCode.R* to the *ExampleProject* directory on the C drive use:

```
file.create("C:\\ExampleProject\\SourceCode.R")
```

unlink

Finally, you can use the **unlink** command to delete a file, files, or directories.

4.5 Unix-like shell commands for file management

The remainder of this chapter is incomplete.

⁵Note: you will need the correct system permissions to be able to do this.

Though this book is mostly focused on using R for reproducible research it can be useful to use a Unix-like shell program to manipulate files in large projects. A command line shell program is simply a program that allows you to type commands to interact with your computer's operating system. We will especially return to shell commands near the end of the book when we discuss make files for compiling large documents, and batch reports (Chapter 12). The syntax discussed here is also similar to the used in command line git (Chapter 5) and Pandoc (chapter 12 and 13). We don't have enough space to properly get started with shell

programs. For good introductions for Unix and Mac OS 10 computers see William E. Shotts Jr.'s book on the Linux command-line(Shotts Jr, 2012). For Windows users, Microsoft maintains a tutorial on Windows PowerShell at <http://technet.microsoft.com/en-us/library/hh848793>.

The one piece of general instruction I will give now is to highlight an important difference in the syntax between R and shell commands. In shell commands you don't need to put parentheses around your arguments. For example if we want to change our working directory to my Mac Desktop in a shell using the `cd` command we simply type:

```
cd /Users/Me/Desktop
```

`cd`

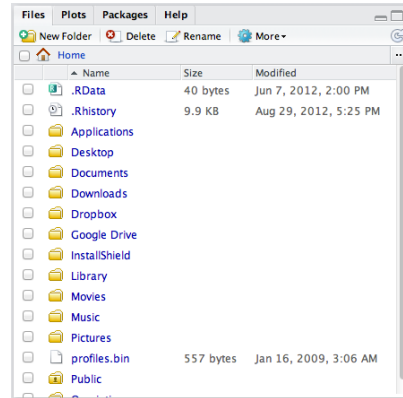
As we just saw, to change the working directory in the shell can just use the `cd` (change directory) command.

`rm`

The `rm` command is similar to R's `unlink` command. It deletes files or directories.

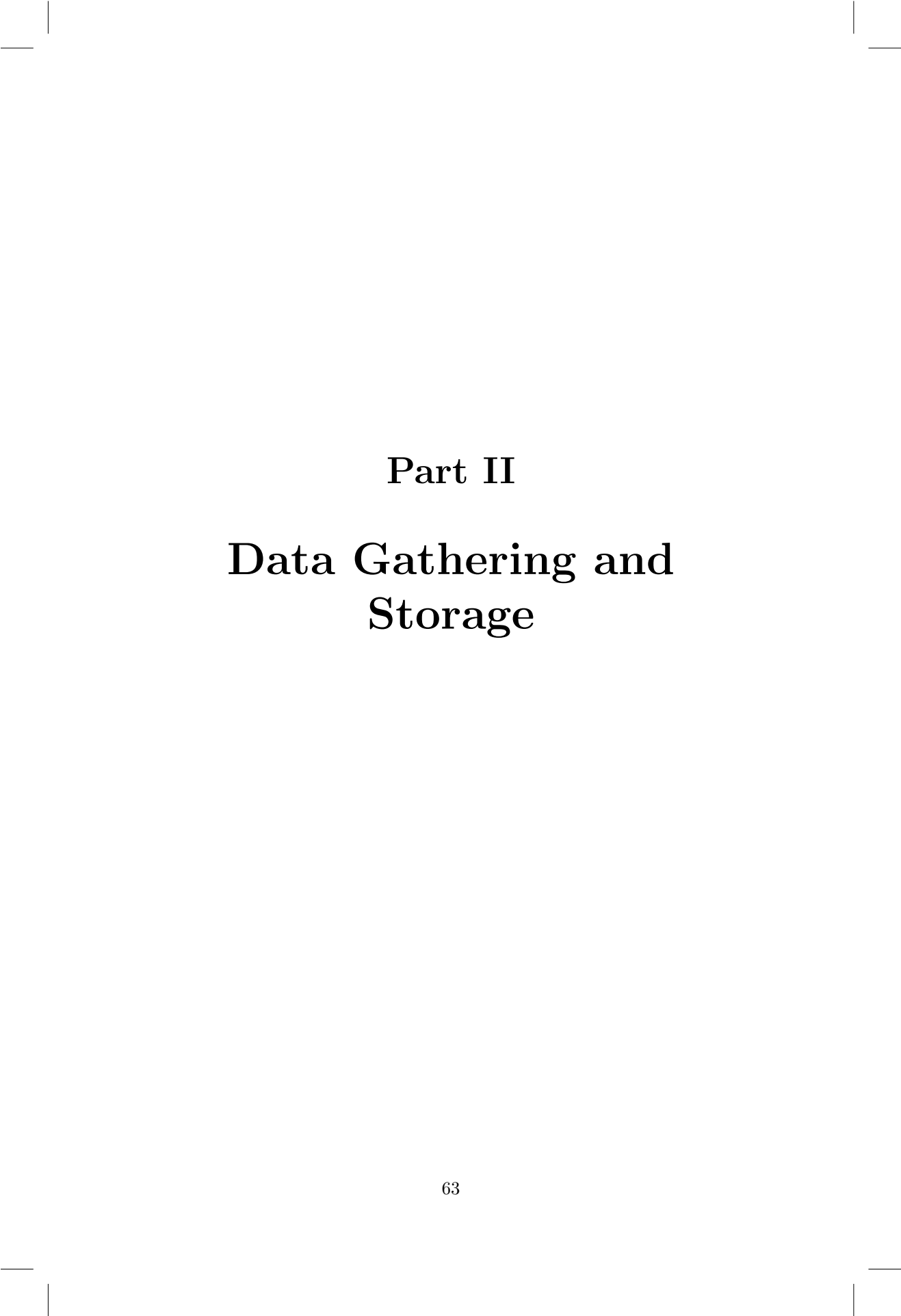
As we saw in Chapter 3, R also has an `rm` command. It is different because it removes objects from your R workspace rather than files from your working directory.

FIGURE 4.3
The RStudio Files Pane



4.6 File navigation in RStudio

The RStudio **Files** pane allows us to navigate and do some basic file manipulation. Figure 4.3 shows us what this pane looks like.



Part II

Data Gathering and Storage



5

Storing, Collaborating, Accessing Files, Versioning

In addition to being well organized, your research files need to be accessible for other researchers to be able to reproduce it. A useful way to make your files accessible is to store them on a cloud storage service (see Howe, 2012). This chapter describes in detail two different ‘cloud’ storage services—Dropbox and GitHub¹—that you can use to make your research files easily accessible to others. Not only do these services enable others to reproduce your research, they also have a number of benefits for your research workflow. Researchers often face a number of data management issues that, beyond making their research difficult to reproduce, can make doing the initial research difficult.

First, there is the problem of **storing** the data so that it is protected against computer failure—virus infections, spilling coffee on your laptop, and so on. Storing data locally—on your computer—or on a flash drive is generally more prone to loss than on remote servers in the cloud.

Second, we may work on a project with different computers and other devices. For example, we may use a computer at work to run computationally intensive analysis, while editing our presentation document on an tablet computer while riding the train to the office. So, we need to be able to **access** our files from multiple devices while in different locations. We often need a way for our **collaborators** to access and edit research files as well.

Finally, we almost never create a data set or write a paper perfectly all at once. We may make changes and then realize that we liked an earlier version, or parts of an earlier version better. This is a particularly important issue in data management where we may transform our data in unintended ways and want to go back to an earlier version. Also, when working on a collaborative project, one of the authors may accidentally delete something in a file that another author needed, for example. To deal with these issues we need to store our data in a system that has **version control**. Version control systems keep track of changes we make to our files and allow us to access previous versions if we like.

You can solve all of these problems in a couple of different ways using free or low cost cloud-based storage formats. In this chapter we will learn how to use Dropbox and GitHub for research file:

¹These services store your data on remote servers

- storage,
- accessing,
- collaboration,
- version control.

5.1 Saving data in reproducible formats

Before getting into the details of cloud-based data storage for all of our research files, let's just consider what type of formats you should actually save your data in. A key issue for reproducibility is that others be able to not only get ahold of the exact data you used in your analysis, but be able to understand and use the data now and in the future. Some file formats make this easier than others.

In general, for small to moderately-sized data sets² a plain-text format like comma-separated values (`.csv`) or tab-separated values³ (`.tsv`) can be a good way to store your data. These formats simply store a data set as a text file. A row in the data set is a line in the text file. Data is separated into columns with commas or tabs, respectively. These formats are not dependent on a specific program. Any program that can open text files can open them including a wide variety of statistical programs other than R. This helps future proof your research. Version control systems that track changes to text, like GitHub—are also very effective version control systems with these types of files.

To save data in a plain-text format with R use the `write.table` command. For example, to save a data frame called *Data* as a csv file called *MainData.csv* in our example *DataFiles* directory (see Figure 4.1):

```
write.table(Data, "/ExampleProject/Data/DataFiles/MainData.csv",
            sep = ",")
```

The `sep = ","` argument specifies that we want to use a comma to separate the values. For csv files you can use a modified version of this command called

²I don't cover methods for storing and handling very large data sets—with high hundreds of thousands and more observations. For information on large data and R, not just storage, one place to look is this blog post by from RDataMining: <http://rdatamining.wordpress.com/2012/05/06/online-resources-for-handling-big-data-and-parallel-computing-in-r/> (posted 6 May 2012).

³Sometimes this format is called tab-delimited values.

`write.csv`. This command simply makes it so that you don't have to write `sep = ","`.

If you want to save your data with rows separated by tabs, rather than commas, simply set the argument `sep = "\t"` and the file extension to `.tsv`.

R is also able to save data in a wide variety of other file formats, mostly through the *foreign* package in. Though, these formats may be less future proof than simple text-formatted data files.

5.2 Storing your files in the cloud

In this book we'll cover two (largely) free cloud storage services that allow you to store, access, collaborate on, and version control your research files. These services are Dropbox and GitHub.⁴ Though they both meet our basic storage needs, they do so in different ways and require different levels of effort to set up and maintain.

These two services are certainly not the only way to make your research files available. Research oriented services include the SDSC Cloud,⁵ the Dataverse Network Project,⁶ figshare⁷ and RunMyCode.⁸ These services include good built-in citation systems, unlike Dropbox and GitHub. They may be a very good place to store research files once the research is completed or close to completion. Some journals are beginning to require key reproducibility files be uploaded to these sites. However, these sites' ability to store, access, collaborate on, and version control files *during* the main part of the research process is mixed. Services like Dropbox and Github are very capable of being part of the research workflow from the beginning.

5.2.1 Dropbox

The easiest types of cloud storage for your research are services like Dropbox⁹ and Google Drive.¹⁰ Most of these services not only store your data in the cloud, but also provide some way to share files and even includes basic version control capabilities. I'm going to focus on Dropbox because it currently offers a complete set of features that allow you to store, version, collaborate, and

⁴Dropbox provides a minimum amount of storage for free, above which they charge a fee. GitHub lets you create publicly accessible repositories—kind of like project folders—for free, but they charge for private repositories.

⁵<https://cloud.sdsc.edu/hp/index.php>

⁶<http://thedata.org/>

⁷<http://figshare.com/>

⁸<http://www.runmycode.org/>

⁹<http://www.dropbox.com/>

¹⁰<https://drive.google.com/>

access your data. I will focus on how to use Dropbox on your computer. Some Dropbox functionality may be different on mobile devices.

5.2.2 Storage

When you sign up for Dropbox and install the program¹¹ it creates a directory on your computer's hard drive. When you place new files and folders in this directory and make changes to them, Dropbox automatically syncs the directory with a similar folder on a cloud-based server. Typically you can sign up for the service for free and receive a limited amount of storage space; usually a few gigabytes.

5.2.2.1 Accessing Data

There are two similar, but importantly different ways to access data stored on Dropbox. All files stored on Dropbox have a URL address through which they can be accessed from a computer connected to the internet. Some of these files can be easily loaded directly into R, while others must be manually (point-and-click) downloaded onto your computer and then loaded into R. Files in the *Public* folder can be downloaded directly into R. Files not in the *Public* folder have to be downloaded manually.¹²

Either way you find a file's URL address by first right-clicking on the file icon in your Dropbox folder. If the file is stored in the *Public* folder, you go to **Dropbox** in the menu that pops up, then click *Copy Public Link*. This copies the URL into your clipboard from where you can paste it into your R source code (or wherever). Once you have the URL you can load the file directly into R using the `read.table` command for dataframes (see Chapter 5) or the `source` command for source code files (see Chapter 8).

To give you a preview of how to download data directly into R from Dropbox, try downloading a data file from my Public folder. The full URL of the data set is: `http://dl.dropbox.com/u/12581470/code/Replicability_code/Fin_Trans_Replication_Journal/Data/public.fin.msm.model.csv`. I've used the URL shortening service bitly¹⁴ to make this link fit on the page.

```
# Download data stored in a Dropbox Public folder
# and save as an object named Data
Data <- read.table("http://bit.ly/PhjaPM",
```

¹¹See <https://www.dropbox.com/downloading> for downloading and installation instructions.

¹²This is not completely true. It could be possible to create a web scraper¹³ that could download data from a file not in your *Public* folder. However, this is kind of a hassle and not practical, especially since the accessing files from the *Public* folder is so easy.

¹⁴See <https://bitly.com/>.

```
sep = ",", header = TRUE)

# Show variables in Data
names(Data)

## [1] "idn"      "country"  "year"     "reg_4state"
```

If the file is not in your *Public* folder you also go to **Dropbox** after right-clicking on the file. Then choose **Get Link**. This will open a webpage in your default web browser from where you can download the file. You can copy and paste the page's URL from your browser's address bar. You can also get these URL links through the online version of your Dropbox. First log into the Dropbox website. If the file is in your *Public* folder, right-click on it and then select **Copy Public Link**. When you hover your cursor over a file or folder not in the *Public* Folder you will see a chain icon appear on the far right. Clicking on this icon will get you the link.

Storing files in the *Public* folder clearly makes replication easier because the files can be downloaded and run directly in R.

5.2.3 Collaboration

Though others can easily access your data and files through Dropbox URL links, you cannot save files through the link. You must save files in the Dropbox folder on your computer. If you would like collaborators to be able to modify the research files you will need to 'share' the Dropbox folder with them. You cannot share your *Public* folder, so you will need to keep the files you want collaborators to be able to modify in a non-public folder. Once you create this folder you can share it with your collaborators by right-clicking on the folder and selecting **Invite to folder** on the Dropbox website or **Dropbox → Share This Folder...** on the locally stored folder. Enter your collaborator's email address when prompted. They will be sent an email that will allow them to accept the share request and, if they don't already have an account, sign up for Dropbox.

5.2.3.1 Version control

Dropbox has a simple version control system. Every time you save a document on Dropbox a new version is created. To view a previous version, navigate to the file on the Dropbox website. Then right-click on the file. In the menu that pops up select **Previous Versions**. This will take you to a webpage listing previous versions of the file, who created the version, and when it was created. A new version of a file is created every time you save a file and it is synced to the Dropbox cloud service.

5.2.4 GitHub

Dropbox adequately meets our four basic criteria for reproducible data storage and is easy to set up. GitHub meets the criteria and more, but is less straightforward at first.

GitHub is an interface and cloud hosting service built on top of the Git version control system. GitHub was not explicitly designed to host research projects or even data. It was designed to host “socially coded” computer programs—in what it Git calls “repositories”—by making it easy for a number of collaborators to work together to build a computer program. This seems very far from reproducible research.

Remember that as reproducible researchers we are building projects out of interconnected text files. In important ways this is exactly the same as building a computer program. Computer programs are also basically large collections of interconnected text files. Like computer programmers, we need ways to store, version control, access, and collaborate on our text files. Because GitHub is very actively used by people with very similar needs (who are also really good programmers), the interface offers many highly developed and robust features for reproducible researchers.

GitHub’s extensive features and heart in the computer programming community means that it takes a longer time than Dropbox to set it up and become familiar with it. So we need good reasons to want to invest the time needed to learn GitHub compared to Dropbox and a similar services. Because of this, you probably need some motivation to want to invest this time. Here is a list of GitHub’s advantages over Dropbox for reproducible research that will hopefully convince you to get started:

Storage and Access

- Dropbox simply creates folders stored in the cloud which you can share with other people. GitHub makes your projects accessible on a fully featured project website (see Figure 5.1). An example feature is that it automatically renders Markdown files called *README.md*¹⁵ in a GitHub directory on the repository’s website. This makes it easy for independent researchers to find the file and read it.
- Public repositories—those viewable by anyone—can be downloaded by anyone in a compressed format.
- GitHub can create and host a website for your research project that you could use to present the results, not just the replication files.

Collaboration

¹⁵You can use a variety of other markup languages as well. See <https://github.com/github/markup>.

Dropbox allows multiple people to share files and change them. GitHub does this and more:

- GitHub keeps meticulous records of who contributed what to a project.
- Each GitHub repository has an “Issues” area where you can note issues and discuss them with your collaborators. Basically this is an interactive to-do list for your research project.
- Each repository can also host a wiki that, for example, could explain in detail how certain aspects of a research project were done.
- Anyone can suggest changes to files in a public repository. These changes can be accepted or declined by the projects authors. The changes are recorded by the Git version control system. This could be especially useful if an independent researcher notices an error.

Version Control

- Dropbox’s version control system only lets you see the file names, the times they were created, who created them, and revert back to specific versions. Git tracks every change you make. The GitHub website and GUI programs for Mac and Windows provide nice interfaces for examining specific changes in text files.
- Dropbox creates a new version every time you save a file. This can make it difficult to actually find the version you want as the versions quickly proliferate. Git’s version control system only creates a new version when you tell it to.
- Dropbox does not merge conflicting versions of a file together. This can be annoying when you are collaborating on project and more than one author is making changes to documents. GitHub identifies conflicts and lets you reconcile them.
- Git is directly integrated into RStudio Projects.¹⁶

5.2.4.1 Setting up GitHub: Basic

There are two ways to set up and use GitHub on your computer. You can use the command line version. It’s available for Mac and Linux and Windows through Git Bash, which is automatically installed when you download it. You can also use the Graphical User Interface GitHub program currently available only for Windows and Mac. The GitHub website has excellent step-by-step instructions for how to install both versions. The first thing to do is go to the

¹⁶RStudio also supports the Subversion version control system, but I don’t cover that here.

GitHub website (<https://github.com/>) and sign up for an account. Then you should go to one of the following website for instructions on setting up Git: <https://help.github.com/articles/set-up-git>.

The remainder of this chapter is incomplete.

5.2.4.2 Storage on GitHub

Files are stored on GitHub in repositories. Repositories are kind of like folders or directories, though it's probably better to think of them as projects.

To set up a GitHub repository...

Setting up GitHub with RStudio Projects

Once you have your repository set up on GitHub you can clone it onto your computer and create an RStudio Project

5.2.4.3 Accessing on GitHub

5.2.4.4 Collaboration with GitHub

Repositories can have official collaborators. Public repositories can have unlimited collaborators. Anyone with a GitHub account can be a collaborator.

Anyone with a GitHub account can make changes to files in a public repository on the repository's website. Simply click the **Edit** button above the file and make edits. If the person making the edits is not the a repository collaborator, their edit will be sent to the repository's owner for approval.¹⁷ This is a useful way for independent researchers to catch errors and directly address them.

Branches

Syncing repository

5.2.4.5 Version Control with GitHub

GitHub's version control system is much more comprehensive than Dropbox's. However, it also has a steeper learning curve.

Reverting to an old version of a file

You can use the `git checkout` command to revert to a previous version of a document, because you accidentally deleted something important or made other changes you don't like. To 'checkout' a particular version of a file type:

¹⁷This is called a `pull` in git terminology

```
git checkout COMMITREF FILENAME
```

Now the previous version of the file is in your working directory, where you can commit it as usual.

Let's break down the code. **FILENAME** is the name of the file that you want to change¹⁸ and **COMMITREF** is the reference that git gave to the commit you want to revert back to. The reference is easy to find and copy in GitHub. On the file's GitHub page click on the **History** button. This will show you all of the commits. By clicking on **Browse Code** you can see what the file at that commit looks like. Above this button is another with a series of numbers and letters. This is the commit's SHA (Secure Hash Algorithm). For our purposes, it is the commit's reference number. Click on the **Copy SHA** button to the left of the SHA to copy it. You can then paste it as an argument to your **git checkout** command.

More Practice with Command Line GitHub

If you want more practice setting up GitHub in the command line, GitHub and the website Code School have an interactive tutorial that you might find interesting. You can find it at: <http://try.github.com/levels/1/challenges/4>.

¹⁸If it is in a repository's subdirectory you will need to include this in the file name.

FIGURE 5.1
Part of this Book's GitHub Repository Webpage

The screenshot displays the GitHub repository page for 'Reproducible Research with R/RStudio'. The repository is owned by 'christophergandrud' and has 15 issues and 0 pull requests. The latest commit, 'Some chapter 8 changes', was made 21 hours ago by 'christophergandrud' with commit hash '0f2b760cc6'.

The repository contains the following files:

| name | age | message | history |
|---------------|--------------|---|---------|
| Old | 4 days ago | Ch6 changes [christophergandrud] | |
| Source | 21 hours ago | Some chapter 8 changes [christophergandrud] | |
| Temp | 13 days ago | 29 Sept edits [christophergandrud] | |
| Writing_Setup | 21 hours ago | Some chapter 8 changes [christophergandrud] | |
| .DS_Store | 4 months ago | Added book files create shell script 2 [christophergandrud] | |
| .Rhistory | a month ago | Day of Isaac additions [christophergandrud] | |
| BookMake.R | 12 days ago | end of September changes [christophergandrud] | |
| Outline.md | 13 days ago | 29 Sept edits [christophergandrud] | |
| README.md | 2 days ago | end of 10 Oct [christophergandrud] | |

The README.md file contains the following text:

```

Reproducible Research for R and RStudio

Christopher Gandrud

```

6

Gathering Data with R

How you gather your data has a big effect on how reproducible your research will be. Of course you should try your best to document everything. Replication will be easier if your documentation—especially, variable descriptions, sources, and source code—makes it so that you and your computer can understand what you have done. If all of your data gathering steps are embedded in source code, then independent researchers (and you) can more easily regather the data. Regathering data will be easiest if your source code can tie your analysis code all the way back to the raw data—the rawer the better. Of course this may not always be possible. You may need to conduct interviews or compile information from paper based archives, for example. The best we can sometimes do is describe our data gathering process in detail. Nonetheless, R’s automated data gathering capabilities for internet-based information are extensive. Learning how to take full advantage of them greatly increases reproducibility and can save researchers considerable time and effort.

In this chapter you’ll learn how to gather quantitative data in a reproducible and, in some cases, fully replicable way. You’ll start by learning how to use data gathering make files to organize your whole data gathering process so that it can be completely reproduced. Then we will discuss the details of actually loading data into R from various sources both locally on your computer and via the internet. In the next chapter (Chapter 7) you’ll learn the details of how to clean up raw data so that it can be merged together into data frames that you can use for statistical analyses.

6.1 Organize your data gathering: make files

Before getting into the details of using R to gather data, let’s start by creating a plan to organize the process. Organizing your data gathering process from the beginning of a research project improves the possibility of reproducibility and can save significant effort over the course of the project by making it easier to add and regather data later on.

A key part of replicable data gathering with R, like replicable research in general is segmenting the process into discrete files that can be run by a common make file. Segmentation makes it easier to navigate research text

and find errors in the source code more easily. The make file's output is the data set(s) that you'll use in the statistical analyses. There are two types of source code files that the make file runs: data gathering/clean up files and merging files. Data clean up files bring raw (the rawer the better) individual data sources into R and transform them into something that can be merged with data from the other sources. Some of the R tools for data clean up and merging will be covered in Chapter 7. In this chapter we mostly cover the ways to bring raw data into R. Merging files are executed by the make file after it runs the data gathering/clean up files.

It's a good idea to have the source code files use as very raw data as input. And have them not directly changed these raw data files. Instead changes should be output to new objects and data files. Doing this makes it easier to reconstruct the steps you took to create your data set. Also, while cleaning and merging your data you may transform it in an unintended way, for example, accidentally deleting some observations that you had intended to keep. Having the raw data makes it easy to go back and correct your mistakes.

In data gathering and make files you usually only need one or two commands `setwd` and `source`. As we talked about in Chapter 4, `setwd` simply tells R where to look for and place files. `source` tells R to run code in an R source code file.¹ Lets see what a data make file might look like for our example project (see Figure 4.1). The file paths in this example are for Unix-like systems.

```
#####  
# Example Make file  
# Christopher Gandrud  
# Updated 10 October 2012  
#####  
  
# Set working directory  
setwd("/ExampleProject/Data/")  
  
# Gather and clean up raw data files.  
source("/GatherSource/IndvDataGather/Gather1.R")  
  
source("/GatherSource/IndvDataGather/Gather2.R")  
  
source("/GatherSource/IndvDataGather/Gather3.R")  
  
# Merge cleaned data files into object CleanedData  
source("/GatherSource/MergeData.R")
```

¹We use the `source` command is used more in the Chapter 8.

```
# Save cleaned & merged Data as MainData.csv
write.csv(CleandedData, file = "/DataFiles/MainData.csv")
```

In this code, first we set the working directory. Then the make file ran three source code files to gather data from three different sources. These files gathered the data, cleaned it so that it could merged together. Next the make file ran a source code file that merged the data frames. Finally, it saved the output data frame *CleanData* as a .csv formatted file called *MainData.csv* using the `write.csv` command. In our example project, this would be the main file we use for statistical analysis.

Now that we've covered the big picture, let's learn the different tools you will need to know to gather data from different types of sources.

6.2 Importing locally stored data sets

The most straightforward place to load data from is a local file, e.g. one on stored on your computer. Though storing your data locally does not really encourage reproducibility, most research projects will involve loading data this way to some extent. The tools you will learn for importing locally stored data files will also be important for most of the other methods further on. Let's briefly look at how to load single and multiple files locally.

6.2.1 Importing a single locally stored file

As we have seen, plain-text file based data stored on your computer can be loaded into R using the `read.table` command. If you are using RStudio you can do the same thing with drop down menus. To open a plain-text data file click on **Workspace** → **Import Dataset...** → **From Text File...** In the box that pops up, specify the separator, whether or not you want the first line to be treated as variable labels, and other options. This is initially easier than using `read.table`. But it is less reproducible.

If the data is not stored in plain-text format, but is instead saved by another statistical program such as SPSS, SAS, or Stata, we can import it using commands in the *foreign* package. For example, imagine we have a data file called *Data1.dta* stored in our working directory. This file was created by the Stata statistical program. To load the data into an R data frame object called *StataData* simply type:

```
# Load library
library(foreign)

# Load Stata formatted data
Data <- read.dta(file = "Data1.dta")
```

As you can see, commands in the *foreign* library have similar syntax to `read.table`. To see the full range of commands and file formats that the *foreign* package supports use the following command

```
library(help = "foreign")
```

If you have data stored in a spreadsheet format such as Excel's `.xlsx`, it may be best to first clean up the data in the spreadsheet program by hand and then save the file in plain-text format. When you clean up the data make sure that the first row has the variable names and that observations are in the following rows. Also, remove any extraneous information—notes, colors, and so on—that will not be part of the data frame.

To aid reproducibility, locally stored data should include careful documentation of where the data came from and how, if at all, it was transformed before it was loaded it into R. Ideally the documentation would be written in a text file saved in the same directory as the raw data file.

6.2.2 Looping through multiple files

This subsection needs to be written.

6.3 Importing data sets from the internet

There are many ways to import data that is stored on the internet directly into R. We have to use different methods depending on where and how the data is stored.

6.3.1 Data from non-secure (http) URLs

Importing data into R that is located at a non-secure URL—it starts with `http`—is straightforward provided that:

- the data is stored in a simple format, e.g. plain-text,
- the file is not embedded in a larger HTML website.

We have discussed the first issue in detail. You can determine if the data file is embedded in a website by opening the URL. If you only see the raw plain-text data, you are probably good to go.²

To import the data simply include the URL as the file name in your `read.table` command. We saw earlier how to download a CSV data file from a Dropbox *Public* folder with the shortened URL `http://bit.ly/PhjaPM`:

```
Data <- read.table("http://bit.ly/PhjaPM",
                   sep = ",", header = TRUE)
```

6.3.2 Data from secure (https) URLs

We have to take a few extra steps to download data from a secure URL. You can tell if the data is stored at a secure web address if it begins with `https` rather than `http`. We need the help of the `getURL` command in the *RCurl* package (Lang, 2012) and `textConnection`. The latter command is in base R. The two rules about data being stored in plain text-formats and not being embedded in a large HTML website apply to secure web addresses as well.

Let's try an example. I have data in comma-separated values format stored at a GitHub repository. The URL for the “raw” (plain-text) version of the data is `https://raw.githubusercontent.com/christophergandrud/Disproportionality_Data/master/Disproportionality.csv`.³ Imagine that we put the address as a character string into an object called *UrlAddress* (not shown).⁴ To download it into R we could use this code:

```
# Load package
library(RCurl)
```

²If the data is embedded in a larger website—the way data is usually stored on the cloud version of Dropbox—you may still be able to download it into R. However, this can be difficult and varies depending on the structure of the website. So, I do not cover it in this book.

³To find the URL for the raw version of a file on the GitHub website simply click the **Raw** button on the right just above the file preview.

⁴See page 30 for how to put a character string into an object. I do not show how this was done in the book, due to space constraints.

```
# Pull data from the internet
Data <- getURL(UrlAddress)

# Convert Data into a data frame
Data <- read.table(textConnection(Data),
                   sep = ",", header = TRUE)

# Show variables in data
names(Data)

## [1] "country"          "year"
## [3] "disproportionality"
```

6.3.3 Compressed data stored online

Sometimes data files are large, making them difficult to store and download without compressing them. There are a number of compression methods such as Zip and tar.⁵ Zip files have the extension `.zip` and tar files use extensions such as `.tar` and `.gz`. In most cases⁶ we can download, decompress, and create data frame objects from these files directly in R.

To do this we need to:⁷

- create a temporary file with `tempfile` to store the zipped file, which we will later remove with the `unlink` command at the end,
- download the file with `download.file`,
- decompress the file with one of the `connections` commands in base R,⁸
- read the file with `read.table`.

The reason that we have to go through so many extra steps is that compressed files are more than just a single file, but can contain a number of files as well as metadata.

Let's download a compressed file called *uds_summary.csv* from Pemstein

⁵Tar archives are sometimes referred to as 'tar balls'.

⁶Some formats that require the *foreign* package to open are more difficult. This is because functions such as `read.dta` for opening Stata `.dta` files only accept file names or URLs as arguments, not connections, which we create for unzipped files.

⁷The description of this process is based on a Stack Overflow comment by Dirk Eddelbuettel (see <http://stackoverflow.com/questions/3053833/using-r-to-download-zipped-data-file-extract-and-import-data?answertab=votes#tab-top>, accessed 16 July 2012).

⁸To find a full list of commands type `?connections` into the R console.

et al. (2010). It is in a compressed file called *uds_summary.csv.gz*. The file's URL address is http://www.unified-democracy-scores.org/files/uds_summary.csv.gz, that I shortened⁹ to <http://bit.ly/S0vxk2> because of space constraints.

```
# For simplicity, store the URL in an object called 'url'.
url <- "http://bit.ly/S0vxk2"

# Create a temporary file called 'temp' to put the zip file into.
temp <- tempfile()

# Download the compressed file into the temporary file.
download.file(url, temp)

# Decompress the file and convert it into a dataframe
# class object called 'data'.
Data <- read.csv(gzfile(temp, "uds_summary.csv"))

# Delete the temporary file.
unlink(temp)

# Show variables in data
names(Data)

## [1] "country" "year"      "cowcode" "mean"      "sd"
## [6] "median"  "pct025"   "pct975"
```

6.3.4 Data APIs & feeds

The remainder of this chapter is incomplete.

There are growing number of packages that can gather data directly from their sources and import them into R. Needless to say, this is great for reproducible research since it not only makes the data gathering process easier (you don't have to download many of Excel files and fiddle around with them before even getting the data into R, but it also makes replicating the data gathering process much more straightforward. Some examples include:

- Need to Complete list.
- The *openair* package, which beyond providing a number of tools for

⁹Again, I used bitly (bitly.com) to shorten the URL.

analysing air quality data also has the ability to directly gather data directly from sources such as Kings College London's London Air (<http://www.londonair.org.uk/>) database with the `importKCL` command.

6.4 Basic web scraping

6.4.1 Gathering and parsing text from the web

6.4.2 Scraping tables

7

Preparing Data for Analysis

This chapter is incomplete.

Once we have gathered the raw data that we want to include in our statistical analyses we generally need to clean it and merge it into a single data file. This chapter covers some of the basics of how to clean data files and merge together into one data frame them using R.

If you are very familiar with data transformations in R you may want to skip onto the next chapter.

7.1 Cleaning data for merging

7.1.1 Renaming variables

7.1.2 Changing variables types

7.1.3 Creating ID Variables

7.1.4 Sorting & ordering data

7.2 Merging data sets

7.2.1 Binding

7.2.2 The merge command

Part III

Analysis and Results



8

Statistical Modelling and knitr

When you have your data cleaned and organized you will begin to examine it with statistical analysis. To make your analysis really reproducible you should dynamically connect the source code of your analysis to your presentation documents. You will be able to run your analysis and present the results whenever you compile your presentation documents. Doing this makes it very clear how you found the results that you are advertising. It also keeps your the presentation of your results—including tables and figures—up-to-date with any changes you make to your data and analysis.

You can dynamically tie your statistical analyses and presentation documents together with *knitr*. In Chapter 3 we learned basic *knitr* syntax. In this chapter you will begin to learn *knitr* syntax in much more detail, particularly code chunk options for including dynamic code in your presentation documents. This includes code that is run in the background, i.e. not shown in the presentation document as well as displaying the code and output in your presentation document both as a separate block and inline with the text. You will also learn how to dynamically include code from languages other than R. We will also examine how to use *knitr* when we have segmented our analysis source code files.

The goal of this and the next two chapters—which cover dynamically presenting results in tables and figures—is to show you how to tie your analyses into your presentation documents so closely that every time the documents are compiled they actually reproduce your analysis and present the results.

Please see the next part of this book, Part IV, for details on how to create the LaTeX and Markdown documents that can include *knitr* code chunks.

8.1 Incorporating analyses into the markup

For a relatively short piece of code that you don't need to run in multiple presentation documents it may be easy to type it directly into code chunks written in your markup document. In this section you will learn how set *knitr* options to handle these code chunks.

8.1.1 Full code chunks

By default *knitr* code chunks are run by R, the code and any text output (including warnings and error messages) are inserted into the text of your presentation documents in blocks. The blocks are positioned in the final presentation document text exactly where they are written in the markup version. Figures are inserted as well. Let's look at the main options for determining how R code is handled by *knitr*.

eval

If you would like to only show your code as a block in the text of your presentation document set the **eval** option to **FALSE**

cache

If you want to store a code chunk's output for use later, rather than running the code chunk every time you compile your presentation document set the option **cache=TRUE**. This will only run the code chunk if the code changes. It is very handy if you have a code chunk that is computationally intensive to run.

Unfortunately, the **cache** option has some limitations. For example, other code chunks can't access objects have been cached.

echo

The opposite of **eval=FALSE** is to have the code chunk evaluated—run—but do not include the code in the presentation document. You can do this by setting **echo=TRUE**. We will use this option extensively in chapters 9 and 10 when you learn how to run R source code to produce tables and figures that are included in the text.

warning, message, error

If you don't want to include in the text of your presentation documents the warnings, messages, and error messages that R outputs when it runs a code chunk just set the **warning**, **message**, and **error** options to **FALSE**.

8.1.2 Showing code & results inline

Sometimes we want to have some R code or text output show up inline with the rest of our presentation document text. For example, you may want to include a small chunk of stylized code in our text when we discuss how we did an analysis. Or you may want to dynamically report the mean of some variable in your text so that the text will change if you change the data. The *knitr* syntax for including inline code is different for the LaTeX and Markdown languages.

8.1.2.1 LaTeX

Inline static code

If you want to include a code snippet inline with your text you can simply use the LaTeX command `\tt`. This sets our text to ‘typewriter’ font, the standard font for inline code in LaTeX (I use it in this book, as you have probably noticed). It is equivalent to the `eval=FALSE` option for full code chunks.

Inline dynamic code

If you want to dynamically show the results of some R code in our LaTeX produced text we can use the `\Sexpr` command. This is a pseudo LaTeX command. Its structure is more like a LaTeX command’s structure than `knitr` in that you enclose our R code in curly brackets (`{}`) rather than the usual `<<>>= . . . @` syntax for block code chunks.

For example, imagine that you wanted to include the mean—591—in the text of your document. The `rivers` numeric vector, loaded by default in R, has the length of 141 major rivers recorded in miles. You can simply use the `mean` command to find the mean and the `round` command to round it to the nearest whole number:

```
round(mean(rivers), digits = 0)

## [1] 591
```

To have just the output show up inline with the text of your document you would type something like:

```
The mean length of 141 major rivers in North America
is \Sexpr{round(mean(rivers), digits = 0)} miles.
```

This produces the sentence:

The mean length of 141 major rivers in North America is 591 miles.

8.1.2.2 Markdown

Inline static code

To include static code inline in an R Markdown (and regular Markdown) document, enclose the code in single backticks (```). For example:

```
This is example R code: `MeanRiver <- mean(rivers)`.
```

produces:¹

```
This is example R code: MeanRiver <- mean(rivers) .
```

Inline dynamic code

Including dynamic code in the body of your R Markdown text is similar to including static code. The only difference is that you put the letter `r` after the first single backtick. For example:

```
`r mean(rivers)`
```

will include the mean value of the *rivers* vector in the text of your Markdown document.

8.1.3 Dynamically including non-R code

You are not limited to dynamically including just R code in your presentation documents. *knitr* can run code from a variety of other languages including: Python, Ruby, Bash, Haskell, and Awk. All you have to do to dynamically include code from one of these languages is set the **engine** code chunk option. For example, to dynamically include a simple line of Ruby code in an R Markdown document simply type:

```
```{r engine='ruby'}
print "Reproducible Research"
```

```
Reproducible Research
```
```

The programming language values **engine** can take are listed in Table 8.1. Please note that currently the range of functions *knitr* supports for these languages is less extensive what it supports for R. For example, there is no syntax highlighting.

¹The exact look of the text depends on the CSS style file you are using.

8.2 Dynamically including segmented analysis files

There are a number of reasons that you might want to have your R source code located in separate files from your markup documents even if you compile them together with *knitr*.

First, it can be unwieldy to edit both your markup and long R source code chunks in the same document, even with RStudio's handy *knitr* code folding and chunk management options. There are just too many things going on in one document.

Second, you may want to use the same code in multiple documents—an article and presentation for example. It is nice to not have to copy and paste the same code into multiple places, but have multiple documents link to the same source code. Plus if you make changes to the source code, these changes will automatically be made across all of your presentation documents. You don't need to make the same changes multiple times.

Third, other researchers trying to replicate your work might only be interested in specific parts of your analysis. If you have the analysis broken into separate and clearly labeled files that are clearly tied together in the markup file with *knitr* it is easy for these researchers to find the specific bits of code that they are interested.

TABLE 8.1

| Knitr engine Values | |
|------------------------|----------------------|
| Value | Programming Language |
| <code>awk</code> | Awk |
| <code>bash</code> | Bash |
| <code>gawk</code> | Gawk |
| <code>haskell</code> | Haskell |
| <code>highlight</code> | Highlight |
| <code>python</code> | Python |
| <code>R</code> | R (default) |
| <code>ruby</code> | Ruby |
| <code>sh</code> | Bash |

8.2.1 Source from a local file

Usually in the early stages of research you may want to run code stored in analysis files located on your computer. Doing this is simple. The *knitr* syntax is the same as for block code chunks. The only change is that instead of writing all of our code in the chunk we save it to its own file and use the `source` command in to access it. For example in an R Markdown file you could run the R code in a file called *MainAnalysis.R* from our ExampleProject like this:

```
```{r, echo=FALSE}
Run main analysis
source("/ExampleProject/Analysis/MainAnalysis.R")
```
```


Notice that we set the `echo=FALSE` option. This will run the analysis and produce objects created by the analysis code that can be used by other code chunks, but the results are not shown in the text of our presentation document. Errors, messages, and warnings will be shown, if we do not tell *knitr* to hide them.

8.2.2 Source from a non-secure URL (http)

Sourcing from your computer is fine if you are working alone and do not want others to access your code. Once you start collaborating and generally wanting people to be able to reproduce your analyses, you need to use another storage method.

The simplest solution to these issues is to host the replication code in your Dropbox public folder. You can find the file's public URL the same way we did in Chapter 5. Now use the `source` command the same way as before with the URL as the argument.

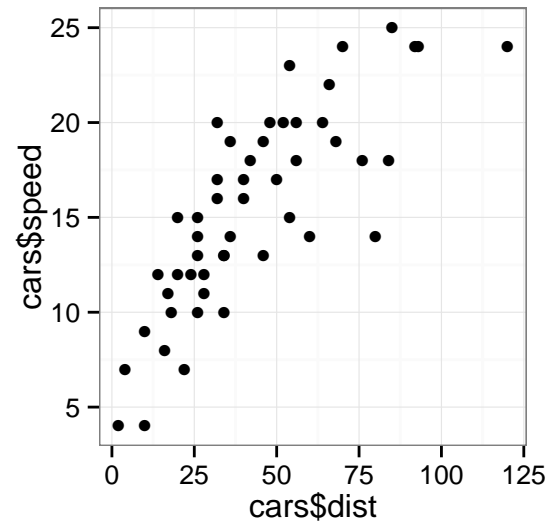
8.2.3 Source from a secure URL (https)

If you are using GitHub or another service that uses secure URLs to host your analysis source code files you need to use the `source_url` command in the *devtools* package. For GitHub based source code we find the file's URL the same way we did in Chapter 5. Remember to use the URL for the *raw* version of the file. I have a short script hosted on GitHub for creating a scatterplot from data in R's *cars* data set. The script's shortened URL is <http://bit.ly/Ny1n6b>.² To run this code and create the scatterplot using `source_url` we simply type:

```
# Load library
library(devtools)

# Run the source code to create the scatter plot
source_url("http://bit.ly/Ny1n6b")
```

²The original URL is at <https://raw.githubusercontent.com/christophergandrud/christophergandrud.github.com/master/SourceCode/CarsScatterExample.R>. This is very long, so I shortened it using bitly (see <http://bitly.com>). You may notice that the shortened URL is not secure. However, it does link to original secure https URL.



9

Showing Results with Tables

Graphs and other visual methods, discussed in the next chapter, can often be a more effective way to present results than tables.¹ Nonetheless, tables of results, descriptive statistics, and so on can sometimes be an important part of communicating research findings.

Creating tables by hand can be tedious no matter what program you are using to type up your results. Even more tedious is making changes to hand-created tables when you make changes to your data and models. Creating these tables can actually introduce new errors—post-analysis!—if you incorrectly copy what is in your R output. This is a very real possibility. The mind can go numb doing that sort of work. Also, creating tables by hand is not very reproducible.

Fortunately, we don't actually need to create tables by hand. There are many ways to have R do the work for us. The goal of this chapter is to learn how to how to **automate table creation** for documents produced with both LaTeX and Markdown/HTML. There are a number of ways to turn R objects into tables written in LaTeX or HTML markup. In this chapter I mostly focus on the `xtable` and `texreg` packages. `xtable` can created tables for both of these markup languages. `texreg` only produces output for LaTeX. `knitr` allows us to incorporate these tables directly into our documents.

Warning: Automating table creation removes the possibility of adding errors to our analyses by incorrectly copying R output, which is a big potential problem in hand-created tables. Be warned, it is not an error free process. We could easily create inaccurate tables through coding errors. For example, we may incorrectly merge together columns in so that our id variables no longer match the data they are supposed to.

So, as always, it is important to 'eyeball' the output. Does it make sense? If we picked a couple values in the R output do they match what is in our final table? If not, we need to go back to the code and see where things have gone wrong. With that caveat, let's start making tables.

¹This is especially true of the small-print, high-density coefficient estimate tables that are sometimes descriptively called 'train schedule' tables.

9.1 Table Basics

Before getting into the details of how to create tables from R objects we need to first learn how generic tables are created in LaTeX and Markdown/HTML.

9.1.1 Tables in LaTeX

9.1.2 Tables in Markdown/HTML

We can use *xtable* and the `print` command to also create tables for Markdown and HTML documents. Instead of setting the `type` argument to `'latex'` we simply put it to `'html'`.

9.2 Creating tables from R objects

9.2.1 `xtable` & `texreg` basics with supported class objects

9.2.1.1 `xtable` for LaTeX

9.2.1.2 `xtable` for Markdown

9.2.2 `xtable` with non-supported class objects

`xtable` and other commands in similar packages are very convenient for making tables from objects in supported classes.² With supported class objects `xtable` knows where to look for the vectors containing the things-coefficient names, standard errors, and so on—that it needs to create the table. With unsupported classes, however, it doesn't know where to look for these things. We need to help it out.

`xtable` does have a way of dealing with `matrix` and `data frame` class objects. The rows of these objects become the rows of the table and the columns become the table columns. So, to create tables with non-supported class objects we need to

1. find and extract the information from the unsupported class object that we want in the table,
2. convert this information into a matrix or data frame where the rows and columns of the object correspond to the rows and columns of the table that we want,
3. use `xtable` with this object to create the table.

²To see a full list of classes that `xtable` supports type `methods(xtable)` into the R console.

Imagine that we want to create a results table showing the covariate names, coefficient means, and quantiles for marginal posterior distributions from a Bayesian normal linear regression using the `zelig` command (Goodrich and Lu, 2007) and data from the *swiss* data frame that comes with R

Note, I am having trouble with this code using Zelig version 4 and am currently working with the packaged developers to sort the issue out. The code does work with Zelig version 3.5.5.

```
# Load required library
library(Zelig)

NBModel <- zelig(Examination ~ Education, model = "normal.bayes",
                 data = swiss, cite = FALSE)

# Find NBModel's class
class(NBModel)

## [1] "MCMCZelig"
```

Using the `class` command we found that the model output object is an `MCMCZelig` class object. This class is not supported by `xtable`. If we try to create a summary table called *NBTable* of the results we will get the following error:

```
## Error: no applicable method for 'xtable' applied to an object of
class "MCMCZelig"
```

With unsupported class objects we have to create the summary ourselves and extract the things that we want from it manually. This is where a good knowledge of vectors comes in handy.

First, let's create a summary of our output object *NBModel*:

```
NBModelSum <- summary(NBModel)
```

We created a new object of the class `summary.MCMCZelig`. We're still not there yet as this object contains not just the covariate names and so on but also information we don't want to include in our results table like the formula that

we used. The second step is to extract a matrix from inside *NBModelSum* called *summary* with the component selector (`$`). This matrix is where the things we want in our table are located. I find it easier to work with data frames, so we'll also convert the matrix into a data frame.

```
NBSumDataFrame <- data.frame(NBModelSum$summary)
```

Here is what our model results data frame looks like:

```
##           Mean      SD   X2.5.   X50.  X97.5.
## (Intercept) 10.1397 1.31673  7.5579 10.1566 12.7058
## Education    0.5786 0.09118  0.3963  0.5781  0.7609
## sigma2       34.9703 7.81260 22.9567 33.8782 53.2172
```

Now we have a data frame object that `xtable` can handle. After a little cleaning up (see the chapter's source code for more details) we can use *NBSumdata* *frame* with `xtable` as before to create the following table:

| | Mean | 2.5% | 50% | 97.5% |
|-------------|-------|-------|-------|-------|
| (Intercept) | 10.14 | 7.56 | 10.16 | 12.71 |
| Education | 0.58 | 0.40 | 0.58 | 0.76 |
| sigma2 | 34.97 | 22.96 | 33.88 | 53.22 |

TABLE 9.1

Coefficient Estimates Predicting Examination Scores in Swiss Cantons (1888)
Found Using Bayesian Normal Linear Regression

It may take a bit of hunting to find what you want, but a similar process can be used to create tables from objects of virtually any class.³ Hunting for what you want is generally easier by clicking on the object in RStudio's workspace pane.

9.2.3 Basic knitr syntax for tables

So far we have only looked at how to create LaTeX and HTML tables from R objects. How can we knit these tables into our presentation documents?

The most important `knitr` chunk option for showing the markup created

³This process can also be used to create graphics.

by these packages as tables is **results**. The **results** option can have one of three values:

- 'markup',
- 'asis',
- 'hide'.

hide clearly hides the results of whatever we have in our code chunk; no results show up. To include tables we create from R objects in our LaTeX or Markdown output we should set **results='asis'** or **results='markup'**. **asis** simply includes the raw output in the presentation document. **markup** uses an output hook to mark up the results. NEED TO EXPLAIN

9.3 Tables with `apsrtable`

10

Showing Results with Figures

This chapter is incomplete.

10.1 Including graphics

10.2 Basic knitr figure options

10.3 Creating figures with plot and ggplot2

10.4 Animations

10.5 Motion charts and basic maps with GoogleVis



Part IV

Presentation Documents



11

Presenting with LaTeX

LaTeX is a very powerful markup language, especially for creating PDF documents such as articles, books, and slideshows.

Much of this chapter is incomplete.

11.1 The Basics

All commands in LaTeX start with the backslash (\) escape character. For example, create a section heading we use the `\section` command. The arguments for LaTeX commands are then written inside of curly braces ({}) written like this:

```
\section{My Section Name}
```

to write a backslash and have it show up as a (\) in LaTeX

11.1.1 Editors

As I mentioned earlier, RStudio is an fully functional LaTeX editor as well as an integrated development environment for R. Of course it is oriented towards combining R and LaTeX. If you want to create a new LaTeX document you can click **File** → **New** → **R Sweave**.

Remember from Chapter 3 that R Sweave files are basically LaTeX files that can include *knitr* code chunks. You can compile R Sweave files like regular LaTeX files in RStudio even if they do not have code chunks. If you use another program to compile them you might need to change the file extension from `.Rnw` to `.tex`.

11.1.2 The header & the body

All LaTeX documents require a header. The header goes before the body of the document and specifies what type of presentation document you are creating—an article, a book, a slideshow, and so on. LaTeX refers to these as classes. You also can specify what style it should be formatted in and load any extra packages you may want to use to help you format your document.¹

The header is followed by the body of your document. You tell LaTeX where the body of your document starts by typing `\begin{document}`. The very last line of your document is usually `\end{document}`, indicating that your document has ended. When you open a new R Sweave file in RStudio it creates an article class document with a very simple header and body like this:

```
\documentclass{article}

\begin{document}

\end{document}
```

11.1.3 Headings

11.1.4 Footnotes & Bibliographies

11.1.4.1 Footnotes

Plain, non-bibliographic footnotes are easy to create in LaTeX. Simply place `\footnote{` where you would like the footnote number to appear in the text. Then type in the footnote's text and of course remember to close it with a `}`. LaTeX does the rest, including formatting and numbering.

11.1.4.2 Bibliographies

Citing R Packages with BibTeX

Researchers are pretty good about consistently citing others' articles and data. However, citing the R packages used in an analysis is very inconsistent. This is unfortunate not only because correct attribution is not being given but also because it makes reproducibility harder because it obscures important steps that were taken in the research process. Fortunately, R actually includes the

¹The command to load a package in LaTeX is `\usepackage`. For example, if you include `\usepackage{url}` in the header of your document you will be able to specify URL links in the body with the command `\url{SOMEURL}`.

tools to quickly generate citations, including the version of the package you are using. It can also add them directly to an existing bibliography file.

You can automatically create citations for R packages using the `citation` command in *base* R. For example if we want the citation information for the `Zelig` package we would simply type:

```
citation("xtable")

##
## To cite package 'xtable' in publications use:
##
##   David B. Dahl (2012). xtable: Export tables to
##   LaTeX or HTML. R package version 1.7-0.
##   http://CRAN.R-project.org/package=xtable
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {xtable: Export tables to LaTeX or HTML},
##     author = {David B. Dahl},
##     year = {2012},
##     note = {R package version 1.7-0},
##     url = {http://CRAN.R-project.org/package=xtable},
##   }
##
## ATTENTION: This citation information has been
## auto-generated from the package DESCRIPTION file and
## may need manual editing, see 'help("citation")' .
```

This gives us both the plain citation as well as the BibTeX version for use in LaTeX and MultiMarkdown documents. If you only want the BibTeX version of the citation we can use the `toBibtex` command in the *utils* package.

```
toBibtex(citation("xtable"))

## @Manual{,
##   title = {xtable: Export tables to LaTeX or HTML},
##   author = {David B. Dahl},
##   year = {2012},
##   note = {R package version 1.7-0},
##   url = {http://CRAN.R-project.org/package=xtable},
```



```
## }
```

You can append the citation to your existing BibTeX file using the `sink` command in *base* R. This command diverts our output and/or the messages to a file. For example, imagine that our existing BibTeX file is called `biblio.bib`. To add the *Zelig* package citation:

```
# Divert output to biblio.bib
sink(file = "biblio.bib",
      append = TRUE, type = c("output")
)
# Extract BibTeX citation
toBibtex(citation("xtable"))
sink()
```

This places the citation at the end of our `biblio.bib` file. It is very important to include the argument `append = TRUE`. If you don't you will erase the existing file. The argument `type = c("output")` tells R to include only the output, not the messages.

An even faster way to add citations to a bibliography is with `write.bibtex` command in the *knitcitations* package. To add the *Zelig* citation to our `biblio.bib` file we only need to enter:

```
# Load package
library(knitcitations)

# Write Zelig citation and
# to biblio.bib
write.bibtex(entry = c("xtable"),
             file = "bibliography.bib", append = TRUE)
```

Note, you will likely only want to append the citations once. Otherwise your bibliography document will grow with redundant information every time you run this command.

The *knitr* package can also create BibTeX bibliographies for R packages using the `write.bib` command. To use this command you list the packages whose citation details you want to include in a specified file. The command currently does not have the ability to append the citations to an existing file, but instead writes them to a new file.

11.2 Presentations with Beamer

You can make slideshow presentations with LaTeX. *FILL IN WITH INTRO*
knitr largely works the same way in LaTeX slideshows as it does in article or book class documents. There are a few differences to look out for.

Slide frames

A quick way to create each Beamer slide is to use the `frame` command:

```
\frame{  
}
```

If you want to include highlighted *knitr* code chunks on your slides you should add the `fragile` option to the `frame` command.² Here is an example:

```
\begin{frame}[fragile]  
  An example fragile frame.  
\end{frame}
```

Results

By default *knitr* hides the results of a code chunk. If you want to show the results in your slideshow simply set the `results` option to `'asis'`.

²For a detailed discussion of why you need to use the `fragile` option with the `verbatim` environment that *knitr* uses to display highlighted text in LaTeX documents see this blog post by Pieter Belmans: <http://pbelmans.wordpress.com/2011/02/20/why-latex-beamer-needs-fragile-when-using-verbatim/>.

12

Large LaTeX Documents: Theses, Books, & Batch Reports

In the previous chapter we learned the basics of how to create LaTeX documents to present our research findings. So far we have only covered basic short documents, like articles. For longer and more complex documents like books we can take advantage of LaTeX and *knitr* options that allow us to separate our files into manageable pieces. The pieces are usually called child files, which are combined using a parent document.

These methods can also be used in the creation of batch reports: documents that present results for a selected part of a data set. For example, a researcher may want create individual reports of answers to survey questions from interviewees with a specific age. In this chapter we will rely on *knitr* and shell scripts to create batch reports.

12.1 Planning large documents

Before discussing the specifics of each of these methods, it's worth taking some time to carefully plan the structure of our child and parent documents.

12.1.1 Planning theses and books

Books and theses have a natural parent-child structure, i.e. they are single documents comprise of multiple chapters. They often include other child-like features such as title pages, bibliographies, figures, and appendices. We could include most of these features directly into one markup file. Clearly this file would become very large and unwieldy. It would be difficult to find one part or section to edit. If of your presentation markup are are difficult to find, they are difficult to reproduce.

12.1.2 Planning batch reports

COMPLETE

12.2 Combining Chapters

We will cover three methods for including child documents into our parent documents. The first is very simple and uses the LaTeX command `\input`. The second using *knitr* is slightly more complex, but gives us much more flexibility. The final method is a special case of `\input` that uses the command line program Pandoc to convert and include child documents written in non-LaTeX markup languages.

12.2.1 Parent documents

knitr global options

knitr global chunk options and package options should be set at the beginning of the parent document if you want them to apply to the entire presentation document.

12.2.2 Child documents

Child documents in the same markup language

Child documents in a different markup language

Because *knitr* is able to run not only R code but also bash programs, you can use the command line program to convert child documents that are in a different markup language into the primary markup language you are using for your document. If you have Pandoc installed on your computer,¹ you can call it directly from your parent document including your Pandoc commands in a code chunk with the `engine` option set to either `'bash'` or `'sh'`.²

For example, the part of this book is written in Markdown. The file is called *StylisticConventions.md*. It was simply faster to write the list of conventions using the simpler Markdown syntax than LaTeX, which as we saw has a more complicated way of creating lists. However, I want to include this list in my LaTeX produced book. Pandoc can convert the Markdown document into a LaTeX file. This file can then be input into my main document with the LaTeX command `\input`.

Imagine that my parent and *StylisticConventions.md* documents are in the same directory. In the parent document I add a code chunk with the options `echo=FALSE` and `results='hide'`. In this code chunk I add the following command to convert the Markdown syntax in *StylisticConventions.md* to LaTeX and save it in a file called *StyleTemp.tex*.

¹Pandoc installation instructions can be found at: <http://johnmacfarlane.net/pandoc/installing.html>.

²Alternatively you can run Pandoc in R using the `system` command.

```
pandoc StylisticConventions.md -f markdown \  
-t latex -o StyleTemp.tex
```

The options `-f markdown` and `-t latex` tell Pandoc to convert *StylisticConventions.md* from Markdown to LaTeX syntax. `-o StyleTemp.tex` instructs Pandoc to save the resulting LaTeX markup to a new file called *StyleTemp.tex*.

I only need to include a backslash (`\`) at the end of the first line because I wanted to split the code over two lines. The code wouldn't fit on this page otherwise. The backslash tells the shell not to treat the following line as a different line. Unlike in R, the bash shell only reads recognizes a command's arguments if they are on the same line.

After this code chunk we need to tell our parent document to include the converted text. To do this we follow the code chunk with the `\input` command like this:

```
\input{StyleTemp.tex}
```

Note that using this method to include a child document that needs to be knit will require extra steps not covered in this book.

12.3 Creating Batch Reports

12.3.1 stich

13

Presenting on the Web and Beyond with Markdown/HTML

13.1 The Basics

13.1.1 Headings

Headings in Markdown are extremely simple. To create a line in the style of the topmost heading—maybe a title—just place one hash mark (#) at the beginning of the line. The second tier heading just gets two hashes (##) and so on. You can also put the hash mark(s) at the end of the heading, but this is not necessary.

13.1.2 Footnotes and bibliographies with MultiMarkdown

13.1.3 Math

13.1.4 Drawing figures with CSS

13.2 Simple webpages

13.2.1 RPubS

13.2.2 Hosting webpages with Dropbox

13.3 Presentations with Slidify

13.4 Reproducible websites

13.4.1 Blogging with Tumblr

13.4.2 Jekyll-Bootstrap and GitHub

see <http://jfisher-usgs.github.com/r/2012/07/03/knitr-jekyll/>

13.4.3 Jekyll and Github Pages

13.5 Using Markdown for non-HTML output with Pandoc

Markdown syntax is very simple. So simple, you may be tempted to write many or all of your presentation documents in Markdown. This presents the obvious problem of how to convert your markdown documents to other markup languages if, for example, you would want to create a LaTeX formatted PDF.

Pandoc can help solve this problem. Pandoc is a command line program that can convert files written in Markdown, HTML, LaTeX, and a number of other markup languages¹ to any of the other formats.

To use Pandoc first install it by following the instructions at <http://johnmacfarlane.net/pandoc/installing.html>. Luckily you do not need to open a shell window in addition to R to run Pandoc. Instead you can run all Pandoc commands in R with the `system` command.

For example,

¹See the Pandoc website for more details: <http://johnmacfarlane.net/pandoc/>

14

Going Beyond the Book

COMPLETE

14.1 Licensing Your Reproducible Research

In the United States and many other countries research, including computer code made available via the internet is automatically given copyright protection. However, copyright protection works against the scientific goals of reproducible research, because work derived from the research falls under the original copyright protections (Stodden, 2009b, 36).

To solve this problem, some authors have suggest placing code under an open source software license like the GNU General Public License (GPL) (Vandewalle et al., 2007). Being designed to make software more freely available, they are not really adequate for making available the data, code, and other material needed to reproduce research findings in a way that enables scientific validation and knowledge growth (see Stodden, 2009b).

Bibliography

- Bacon, F. R. (1267/1859). *Opera qudam hactenus inedita. Vol. I. containing I.–Opus tertium. II.–Opus minus. III.–Compendium philosophi.* Retrieved from <http://books.google.com/books?id=wMUKAAAAYAAJ>.
- Ball, R. and Medeiros, N. (2011). Teaching Integrity in Empirical Research: A Protocol for Documenting Data Management and Analysis. *The Journal of Economic Education*, 43(2):182–189.
- Barr, C. D. (2012). Establishing a Culture of Reproducibility and Openness in Medical Research with an Emphasis on the Training Years. *Chance*, 25(3):8–10.
- Bowers, J. (2011). Six steps to a better relationship with your future self. *Newsletter of the Political Methodology Section, APSA*, 18(2):2–8.
- Braude, S. (1979). *ESP and Psychokinesis. A philosophical examination.* Temple University Press, Philadelphia, PA.
- Buckheit, J. B. and Donoho, D. L. (1995). *Wavelab and Reproducible Research*, pages 55–81. Springer, New York.
- Donoho, D. L. (2010). An Invitation to Reproducible Computational Research. *Biostatistics*, 11(3):385–388.
- Donoho, D. L., Maleki, A., Shahram, M., Rahman, I. U., and Stodden, V. (2009). Reproducible Research in Computational Harmonic Analysis. *Computing in Science & Engineering*, 11(1):8–18.
- Fomel, S. and Claerbout, J. F. (2009). Reproducible Reserarch. *Computing in Science & Engineering*, 11(1):5–7.
- Goodrich, B. and Lu, Y. (2007). *normal.bayes: Bayesian Normal Linear Regression.*
- Howe, B. (2012). Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science & Engineering*, 14(4):36–41.
- Kelly, C. D. (2006). Replicating Empirical Research in Behavioral Ecology: How and Why it Should be Done But Rarely Ever Is. *The Quarterly Review of Biology*, 81(3):221–236.

- King, G. (1995). Replication, Replication. *PS: Political Science and Politics*, 28(3):444–452.
- King, G., Keohane, R., and Verba, S. (1994). *Designing Social Inquiry*. Princeton University Press, Princeton.
- Knuth, D. E. (1992). *Literate Programming*. CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA.
- Lang, D. T. (2012). *RCurl: General network (HTTP/FTP/...) client interface for R*. R package version 1.95-0.1.
- Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In Härdle, W. and Rönz, B., editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg. ISBN 3-7908-1517-9.
- Matloff, N. (2011). *The Art of Programming in R: A Tour of Statistical Programming Design*. No Starch Press, San Francisco.
- Mesirov, J. P. (2010). Accessible Reproducible Research. *Science*, 327(5964):415–416.
- Nagler, J. (1995). Coding Style and Good Computing Practices. *PS: Political Science and Politics*, 28(3):488–492.
- Nosek, B. A., Spies, J. R., and Motyl, M. (2012). Scientific Utopia: II. Restructuring Incentives and Practices to Promote Truth Over Publishability. *Perspectives on Psychological Science*.
- Pemstein, D., Meserve, S. A., and Melton, J. (2010). Democratic compromise: A latent variable analysis of ten measures of regime type. *Political Analysis*, 18(4):426–449.
- Peng, R. D. (2009). Reproducible research and Biostatistics. *Biostatistics*, 10(3):405–408.
- Peng, R. D. (2011). Reproducible Research in Computational Science. *Science*, 334:1226–1227.
- R Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.
- Ramsey, N. Noweb—a simple, extensible tool for literate programming. <http://www.cs.tufts.edu/~nr/noweb/>.
- RStudio (2012). *RStudio: Integrated development environment for R*. Boston, MA. Version 0.97.142.

- Shotts Jr, W. E. (2012). *The Linux Command Line: A Complete Introduction*. No Starch Press, San Francisco.
- Stodden, V. (2009a). The reproducible research standard: Reducing legal barriers to scientific knowledge and innovation. Communia: Global Science Economics of Knowledge-Sharing Institutions Torino, Italy June 30. <http://www.stanford.edu/~vcs/talks/VictoriaStoddenCommuniaJune2009-2.pdf>.
- Stodden, V. (2009b). The Legal Framework for Reproducible Scientific Research. *Computing in Science & Engineering*, 11(1):35–40.
- Vandewalle, P. (2012). Code Sharing is Associated with Research Impact in Image Processing. *Computing in Science & Engineering*, 14(4):42–47.
- Vandewalle, P., Barrenetxea, G., Jovanovic, I., Ridolfi, A., and Vetterli, M. (2007). Experiences with Reproducible Research in Various Facets of Signal Processing Research. *Acoustics, Speech and Signal Processing*, 4:1253–1256.
- Xie, Y. (2012a). *formatR: Format R Code Automatically*. R package version 0.6.
- Xie, Y. (2012b). *knitr: A general-purpose package for dynamic report generation in R*. R package version 0.8.

Index

formatR, 24

API, 18
argument, 38
assignment operator, 31
attach, 35
Awk, 91

Bash, 91
batch reports, 60, 111

cache, 45, 88
cache code chunks, 11
cbind, 33
cd, 60
child files, 111
chunk hooks, 48
cloud storage, 53
code chunk, 12, 43
code chunk options, 45
code highlighting, 11
comma-separated values, 22, 66
command line, 13
comment declaration, 23
component selection, 34
concatenate, 33
CRAN, 40
CSS, 90

data file formats, 66
data frame, 32, 34
dir.create, 59
directories, 54
Donald Knuth, 11
drive letter assignment, 54
Dropbox, 27
Dropbox Public folder, 68

echo, 88
engine, 90
error, 88
escape character, 54, 105
eval, 88

file compression, 80
file extension, 48
file path, 53
file path naming conventions, 53
file.create, 59
foreign, 77

Gawk, 91
getURL, 18, 79
getwd, 58
git, 12, 70
Git Bash, 71
git pull, 72
GitHub, 14, 27, 57, 79, 92
Github repository, 72
global chunk options, 46, 112
Google R Style Guide, 24
Graphical User Interface, 11
GUI, 11

Haskell, 91
help file, 38
Highlight, knitr engine option, 91
hook, 99
hooks, 48

includegraphics, 18
input, 112
install.packages, 40
integrated developer environment,
10

- Jon Claerbout, 4
- knit, 4
- knitr, 4, 11, 42
- LaTeX begin document, 106
- LaTeX class, 106
- LaTeX distribution, 13
- LaTeX header, 106
- Linux, 10
- list, 32
- list.files, 58
- lists, 32
- literate programming, 10, 11, 25
- local chunk options, 46
- locally stored, 53
- ls, 39
- Lyx, 12
- Mac, 12, 30
- make file, 60, 75
- markdown package, 13
- markup language, 10
- MatLab, 4
- matrix, 32, 33
- mean, 34
- message, 88
- Microsoft Excel, 22
- Microsoft Word, 11, 22
- mirrors, CRAN, xv, 40
- NA, 32
- notebook, 48
- object-oriented, 30
- operating systems, 54
- output hooks, 48
- package options, 47, 112
- packages, 21, 40
- Pandoc, 112
- parent directory, 54
- parent document, 111
- progress bar, 47
- Python, 90, 91
- R console, 29
- R LaTeX, 48
- R libraries, 40
- R Markdown, 48
- R session, 30
- R Sweave, 43, 105
- read.table, 18, 77
- README file, 57
- repository, 70
- reproducible research environment, 9
- reproducible research publisher, 9
- reStructuredText, 11
- results, knitr option, 99
- rm, 40, 60
- root directory, 54
- RStudio, 11, 41
- RStudio Notebook, 42
- RStudio Options window, 50
- RStudio pane, 41
- RStudio Projects, 53, 57, 71, 72
- Ruby, 91
- save.image, 39
- session info, 21
- setwd, 59, 76
- Sexpr, 89
- SHA, 73
- source, 76
- source code, 11
- source command, 42
- Source pane, 48
- Stata, 77
- style guide, 24
- subdirectory, 54
- subscripts, 35
- SVN, 12
- Sweave, 11, 50
- tab-delimited values, 66
- tab-separated values, 66
- Terminal, 30
- tie commands, 18, 25
- Ubuntu, 30
- Unix, 12, 30

Unix-like shell program, 10, 53,
 60
unlink, 59

vector, 32
version control, 58

warning, 88
weave, 4
wiki, 71
Windows, 12
Windows PowerShell., 10
with, 35
working directory, 55, 59
workspace, 39
write.csv, 67, 77
write.table, 66
WYSIWYG, 12