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 among these people is Yihui Xie. He is the developer of the *knitr* package (among others) and also an avid writer and commenter 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 contributors to this reproducible research community include Carl Boettiger (who also developed the *knitcitations* package), Markus Gesmann (who developed *googleVis*), Jeromy Anglim, Rob Hyndman and, especially Ramnath Vaidyanathan (who developed *Slidify* and is an active *knitr*).

The vibrant 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 have already been asked.

Thank you also to Victoria Stodden, Michael Malecki and a number of anonymous reviewers for helpful suggestions.

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 my course: Introduction to Social Science Data Analysis. I tested many of the explanations and examples in this book on my students. Their feedback has been very helpful for making the book clearer and more useful. Their experience with using these tools on Windows computer was also important for improving the book's Windows documentation.

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 How to read this book . . . 14
1.6.2 How this book was written . . . 14
1.6.3 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 . . . 18
2.2 Practical tips for reproducible research . . . 21
2.2.1 Document everything! . . . 21
2.2.2 Everything is a (text) file . . . 22
2.2.3 All files should be human readable . . . 23
2.2.4 Explicitly tie your files together . . . 25
2.2.5 Have a plan to organize, store, & make your files available . . . 27

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 packages and loading commands . . . 40
3.2 Using RStudio . . . 41
3.3 Using knitr: the basics . . . 43
3.3.1 File extensions . . . 43
3.3.2 Code Chunks . . . 44
3.3.3 Global options . . . 46
3.3.4 knitr package options . . . 48
3.3.5 Hooks . . . 49
3.3.6 knitr & RStudio . . . 49
3.3.7 knitr & R . . . 52

4 Getting Started with File Management 55
4.1 File paths & naming conventions . . . 56
4.1.1 Root directories . . . 56
4.1.2 Subdirectories & parent directories . . . 56
4.1.3 Spaces in directory & file names . . . 57
4.1.4 Working directories . . . 57
4.2 Organizing your research project . . . 57
4.3 Setting directories as RStudio Projects . . . 59
4.4 R file manipulation commands . . . 60
4.5 Unix-like shell commands for file management . . . 63
4.6 File navigation in RStudio . . . 67

II Data Gathering and Storage 69

5 Storing, Collaborating, Accessing Files, Versioning 71
5.1 Saving data in reproducible formats . . . 72
5.2 Storing your files in the cloud . . . 73
5.2.1 Dropbox . . . 73
5.2.2 Storage . . . 74
5.2.2.1 Accessing Data . . . 74
5.2.3 Collaboration . . . 75
5.2.3.1 Version control . . . 75
5.2.4 GitHub . . . 76
5.2.4.1 Setting up GitHub: Basic . . . 78
5.2.4.2 Version Control with Git . . . 78
5.2.4.3 Remote Storage on GitHub . . . 84
5.2.4.4 Accessing on GitHub . . . 86
5.2.4.5 Collaboration with GitHub . . . 86
5.2.5 Summing up the GitHub workflow . . . 87
5.3 RStudio & GitHub . . . 87

5.3.1 Set Up . . . 87
5.3.2 Using Git in RStudio projects . . . 89

6 Gathering Data with R **93**
6.1 Organize your data gathering: make files . . . 93
6.1.1 R Make-like files . . . 94
6.1.2 GNU Make . . . 95
6.2 Importing locally stored data sets . . . 101
6.2.1 Importing a single locally stored file . . . 101
6.3 Importing data sets from the internet . . . 102
6.3.1 Data from non-secure (`http`) URLs . . . 102
6.3.2 Data from secure (`https`) URLs . . . 103
6.3.3 Compressed data stored online . . . 104
6.3.4 Data APIs & feeds . . . 105
6.4 Advanced Automatic Data Gathering: web scraping . . . 107

7 Preparing Data for Analysis **109**
7.1 Cleaning data for merging . . . 109
7.1.1 Get a handle on your data . . . 109
7.1.2 Reshaping Data . . . 110
7.1.3 Renaming variables . . . 113
7.1.4 Ordering data . . . 114
7.1.5 Subsetting data . . . 115
7.1.6 Recoding variables strings/ numeric variables . . . 117
7.1.7 Creating new variables from old . . . 118
7.1.8 Changing variables types . . . 120
7.2 Merging data sets . . . 121
7.2.1 Binding . . . 121
7.2.2 The merge command . . . 121
7.2.3 Duplicate values . . . 123
7.2.4 Duplicate columns . . . 124

III Analysis and Results **127**

8 Statistical Modelling and knitr **129**
8.1 Incorporating analyses into the markup . . . 130
8.1.1 Full code chunks . . . 130
8.1.2 Showing code & results inline . . . 131
8.1.2.1 LaTeX . . . 131
8.1.2.2 Markdown . . . 133
8.1.3 Dynamically including non-R code in code chunks . . . 133
8.2 Dynamically including modular analysis files . . . 134
8.2.1 Source from a local file . . . 135
8.2.2 Source from a non-secure URL (`http`) . . . 136
8.2.3 Source from a secure URL (`https`) . . . 137

9 Showing Results with Tables **139**
9.1 Table Basics . . . 140
9.1.1 Tables in LaTeX . . . 140
9.1.2 Tables in Markdown/HTML . . . 144
9.2 Creating tables from R objects . . . 148
9.2.1 *xtable* & *apsrtable* basics with supported class objects 148
9.2.1.1 *xtable* for LaTeX . . . 149
9.2.1.2 *xtable* for Markdown . . . 150
9.2.1.3 *apsrtable* for LaTeX . . . 150
9.2.2 *xtable* with non-supported class objects . . . 151
9.2.3 Basic *knitr* syntax for tables . . . 153
9.2.4 Knitting tables in LaTeX . . . 153

10 Showing Results with Figures **155**
10.1 Including graphics . . . 156
10.2 Basic knitr figure options . . . 156
10.2.1 Chunk options . . . 156
10.2.2 Global options . . . 156
10.3 Creating static figures with ggplot2 . . . 156
10.4 Motion charts and basic maps with googleVis . . . 156
10.5 Animations . . . 157

IV Presentation Documents **159**

11 Presenting with LaTeX **161**
11.1 The Basics . . . 161
11.1.1 Editors . . . 161
11.1.2 Basic syntax . . . 161
11.1.3 The preamble & the body . . . 162
11.1.4 Headings . . . 162
11.1.5 Footnotes & Bibliographies . . . 162
11.1.5.1 Footnotes . . . 162
11.1.5.2 Bibliographies . . . 162
11.2 Presentations with Beamer . . . 165
11.2.1 knitr LaTeX slideshows . . . 165

12 Large LaTeX Documents: Theses, Books, & Batch Reports **167**
12.1 Planning large documents . . . 167
12.1.1 Planning theses and books . . . 167
12.1.2 Planning batch reports . . . 168
12.2 Combining Chapters . . . 168
12.2.1 Parent documents . . . 168
12.2.2 Child documents . . . 168
12.3 Creating Batch Reports . . . 169
12.3.1 stich . . . 169

13 Presenting on the Web and Beyond with Markdown/HTML 171
13.1 The Basics 171
13.1.1 Headings 171
13.1.2 Math 171
13.1.3 Escape characters 171
13.1.4 CSS style files and Markdown 171
13.1.5 Added features with MultiMarkdown 172
13.2 Simple webpages 172
13.2.1 RPubs 172
13.2.2 Hosting webpages with Dropbox 172
13.3 Reproducible websites 172
13.4 Presentations with `Slidify` 172
13.4.1 Blogging with Tumblr 177
13.4.2 Jekyll-Bootstrap and GitHub 177
13.4.3 Jekyll and Github Pages 177
13.5 Using Markdown for non-HTML output with Pandoc 177

14 Going Beyond the Book 179
14.1 Licensing Your Reproducible Research 179

Index 187

Stylistic Conventions

I use the following conventions throughout this book:

- **Abstract Variables**

Abstract variables, i.e. variables that do not represent specific objects in an example, are in `ALL CAPS 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*. I use CamelBack for file and directory names.

- **File extensions**

Like filenames, file extensions are *italicized*.

- **Individual variable values**

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

- **Objects**

Objects are printed in *italics*. I use CamelBack for object names.

- **Object Columns**

Data frame object 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*. I use CamelBack for individual variable names.

Required R Packages

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

```
install.packages("animation",
                 "apsrtable",
                 "countrycode",
                 "devtools",
                 "formatR",
                 "gdata",
                 "ggplot2",
                 "googleVis",
                 "httr",
                 "httr",
                 "knitr",
                 "knitcitations",
                 "markdown",
                 "openair",
                 "plyr",
                 "quantmod",
                 "reshape",
                 "reshape2",
                 "RCurl",
                 "rjson",
                 "RJSONIO",
                 "texreg",
                 "tools",
                 "treebase",
                 "twitteR",
                 "WDI",
                 "XML",
                 "xtable",
                 "Zelig")
```

Once you enter this code, you may be asked to select a CRAN "mirror" to download the packages from.[1] Simply select the mirror closest to you.

Ramnath Vaidyanathan's *Slidify* package (**?**) for creating R Markdown/HTML slideshows (see Chapter 13) is not currently on CRAN. It can be downloaded directly from GitHub. To do this first load the *devtools* package (Wickham and Chang, 2012a).[2] Then download *Slidify*. Here is the complete code:

```
# Load devtools
library(devtools)

# Install Slidify and ancillary libraries
install_github("slidify", "ramnathv")
install_github("slidifyLibraries", "ramnathv")
```

For more details see the *Slidify* website: `http://ramnathv.github.com/slidify/start.html#`.

If you are Windows you will also need to install Rtools (Ripley and Murdoch, 2012). You can download Rtools from: `http://cran.r-project.org/bin/windows/Rtools/`.

Fix write_bib issue

[1] CRAN stands for the Comprehensive R Network.

[2]

List of Figures

2.1 Example Workflow & Commands to Tie it Together 20

3.1 R Startup Console 30
3.2 RStudio Startup Panel 42
3.3 RStudio Source Code Pane Top Bars 43
3.7 RStudio Notebook Example 50
3.8 Folding Code Chunks in RStudio 51

4.1 Example Research Project File Tree 58
4.2 An Example RStudio Project Menu 59
4.3 The RStudio Files Pane 63

5.1 A Basic Git Repository with Hidden *.git* Folder Revealed . . 77
5.2 Part of this Book's GitHub Repository Webpage 90
5.3 The RStudio Git Tab 91

7.1 Density Plot of Fertilizer Consumption (kilograms per hectare of arable land) 115

List of Tables

2.1 A Selection of Commands for Tying Together Your Research Files 26

3.1 A Selection of *knitr* Code Chunk Options 47

5.1 A Selection of Git Commands Used in this Chapter 81

7.1 Long Formatted Data Example 111
7.2 Long Formatted Time-series Cross-sectional Data Example . 111
7.3 Wide Formatted Data Example 112
7.4 R's Logical Operators 117
7.5 Example Factor Levels 119

8.1 Knitr `engine` Values 134

9.1 Example Simple LaTeX Table 144
9.2 Linear Regression, Dependent Variable: Exam Score 150
9.3 Coefficient Estimates Predicting Examination Scores in Swiss Cantons (1888) Found Using Bayesian Normal Linear Regression 153

13.1 A Selection of HTML5 Slideshow Frameworks 173

Part I

Getting Started

1

Introducing Reproducible Research

Research is often presented in very abridged packages: slideshows, journal articles, books, or maybe even websites. These presentation documents announce a project's findings and try to convince us that the results are correct (Mesirov, 2010). It's important to remember that these documents are not the research. Especially in the computational and statistical sciences, these documents are the "advertising". The research is the "full software environment, code, and data that produced the results" (Buckheit and Donoho, 1995; Donoho, 2010, 385). When we separate the research from its advertisement we are making it difficult for others to verify the findings by reproducing them.

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

- the R statistical language that will allow you to gather data and analyze it,
- the LaTeX and Markdown markup languages that you can use to create documents–slideshows, articles, books, and webpages–to present your findings,
- the *knitr* package and other tie commands, that dynamically tie your data gathering, analysis, and presentation documents together so that they can be easily reproduced,
- RStudio, a program that brings all of these tools together in one place.

1.1 What is reproducible research?

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 findings by following the procedures originally used to gather the data and run the computer code. Of course it is sometimes difficult to replicate the original data set because of limited resources.[1] So as a next-best standard we can aim for "really reproducible research" (Peng, 2011, 1226).[2] In computational sciences[3] 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 for dynamically "*knitting*"[4] the data and the source code together with your presentation documents. Combined with well organized source files and clearly and completely commented code, independent researchers will be able to understand how you obtained your results. This will make your computational research easily reproducible.

---

[1] In this book we will actually aim for replicable research, even if we don't always achieve it. New technologies make it possible to easily replicate some kinds of data sets, especially if the original data is available over the internet.

[2] The 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).

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

[4] Much of the reproducible computational research and literate programming literatures have traditionally used the term "weave" to describe the process of combining source code and presentation documents (see 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 (knit + R). It 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 the tools of reproducible research also have direct benefits for you as a researcher.

1.2.1 For Science

Replicability has been a key part of scientific 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 developing procedures that have already been collected or figured out. Because researchers do not have to discover on their own things that have already been done, they can more quickly apply these data and procedures to building on established findings and developing new knowledge.

[5]See 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 things such as 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 are avoiding personal effort duplication; allowing you to cumulatively build on your own work more effectively.

Better teamwork

The steps you take to make sure an independent researcher can figure out what you have done also make it easier for your collaborators to understand your work and build on it. This applies not only to current collaborators, but also future collaborators. Bringing new members of a research team up to speed on a cumulatively growing research project is faster if they can easily understand what has been done already (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 own 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 your statistical models, create new graphs, alter results tables in light of new findings, and so on. You will probably try to make these changes long after you last worked on the project and long since you remembered the details of how you did

[6] Of course, it's important to keep in mind that reproducibility is "neither necessary nor sufficient to prevent mistakes" (Stodden, 2009a).

it. Whether your changes are because of journal reviewers' and conference participants' comments or you discover that new and better data has been made available since beginning the project, designing your research to be reproducible from the start makes it much easier to change things later on.

Dynamically reproducible documents in particular can make 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 with existing data sets. If you used data imputation or matching methods you may need to rerun these models. You then have to update your main statistical analyses, and recreate the tables and graphs you used to present the results. Adding a new variable essentially forces you to reproduce large portions of your research. If when you started the project you used tools that make it easier for others to reproduce your research, you also made it easier to reproduce the work yourself. You will have taken steps to have a "better relationship with [your] future [self]" (Bowers, 2011).

Higher research impact

Reproducible research is more likely to be useful for other researchers than non-reproducible research. Useful research is cited more frequently (Donoho, 2002; Piwowar et al., 2007; Vandewalle, 2012). Research that is fully reproducible contains more information, i.e. more reasons to use and cite it, than presentation documents merely showing findings. Independent researchers may use the reproducible the data or code to look at other, often unanticipated, questions. When they use your work for a new purpose they will (should) cite your work. Because of this, Vandewalle et al. even argue that "the goal of reproducible research is to have more impact with our research" (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 more researchers that use the tools of reproducibility the better. So I include enough information in the book for people who have very limited experience with these tools, including limited experience with R, LaTeX, and Markdown. They will be able to start incorporating these tools into their workflow right away. The book will also be helpful for people who already have general experience using technologies such as the R and LaTeX, but would like to know how to tie them together for reproducible research.

1.3.1 Academic Researchers

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

1.3.2 Students

Upper-level undergraduate and graduate students conducting original computational research should make their research reproducible for the same reasons that professional researchers should. Forcing yourself to clearly document the steps you took will also encourage you to think more clearly about what you are doing and reinforce what you are learning. It will hopefully give you a greater appreciation of research accountability and integrity early in your 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 assignments they not only build students' understanding of research best practice, but are also better able to evaluate and provide meaningful feedback on students' work (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, editing it, and developing systems to make it more widely available. The journal *Biostatistics* is a good example of a publication that is encouraging (actually requiring) reproducible research. From 2009 the journal has had an editor for reproducibility that ensures replication files are available and that results can be replicated using these files (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

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

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 persistant versions of them) for redistribution". The second set of tools is a **reproducible research publisher**, which prepares dynamic documents for presenting results and is easily linked to the reproducible research environment (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 Core Team, 2012; RStudio, 2012). R/RStudio can be linked to numerous reproducible research publishers such as LaTeX and Markdown with Yihui Xie's *knitr* package (2012c). 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 you to combine your 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 for reproducible research? R has a very active development community that is constantly expanding what

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

it 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 analysess to presentation documents created with markup languages such as LaTeX and Markdown. This allows you to dynamically and reproducibly present results in articles, slideshows, and webpages.

The way you interact with R has benefits for reproducible research. In general you interact with R (or any other programming and markup language) by explicitly writing down your steps as source code. This promotes reproducibility more than your typical interactions with Graphical User Interface (GUI) programs like SPSS[9] 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 LaTeX[11] 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.[12] It gives you better control over the inclusion of graphics and can cache code chunks–save the output for later. It has the ability to understand Sweave-like syntax, so

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

[10] Donald Knuth coined the term literate programming in the 1970s to refer to a source file that could be both run by a computer and "woven" with a formatted presentation document (Knuth, 1992).

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

[12] Syntax 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.

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 integrated development environment for reproducible research? R by itself has the capabilities necessary to gather data, analyse it, and, with a little help from *knitr* and markup languages, present results in a way that is highly reproducible. RStudio allows you to do all of these things, but simplifies many of them and allows you to navigate through them more easily. It 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. For LaTeX documents it can, for example, insert frequently used commands like `\section{}` for numbered sections (see Chapter 11).[13] 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. 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 C++, CSS, JavaScript, and a few other programming languages. It is closely integrated with the version control programs Git and SVN. Both of these programs allow you to keep track of the changes you make to your documents (see Chapter 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-like operating systems. They should run well on most modern computers.

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

- **R**: `http://www.r-project.org/`,
- **RStudio**: `http://www.rstudio.com/ide/download/`.

[13] If you are more comfortable with a what-you-see-is-what-you-get (WYSIWYG) word processer 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 langages covered in this book. For more information see: `http://www.lyx.org/`. I give some brief information on using Lyx with *knitr* in Chapter 3's Appendix.

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

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

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, or any other program that is a part of this workflow. Instead it shows you how these tools can fit together to make your research more reproducible. To get the most out of these individual programs I will along the way point you to other resources that cover these programs in more detail.

To that end, I can recommend a number of resources that cover more of the nitty-gritty:

- Michael J. Crawley's (2013) encyclopaedic R book, appropriately titled, **The R Book** published by Wiley.
- Similarly, Robert I. Kabacoff's (2011) useful book **R in Action** published by Manning. He also maintains a very helpful website called Quick-R (`http://www.statmethods.net/`).
- Norman Matloff's (2011) 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.

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

- 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 (2012) book **The Linux Command Line: A Complete Introduction** also published by No Starch Press.

- The RStudio website (`http://www.rstudio.com/ide/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.

1.6.1 How to read this book

This book gives you a workflow. It has a beginning, middle, and end. So, unlike a reference book it can and should be read linearly as it takes you through an empirical research processes from an empty folder to a completed set of documents that reproducibly showcase your findings.

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

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

1.6.2 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.

1.6.3 Contents overview

The book is broken into four parts. The first part (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 work reproducible near the end of the research process when they write up the results or maybe even later when a journal requires their data and code be made available for publication. Or maybe even later when another researcher asks if they can use the data from a published article to reproduce the findings. By then there may be numerous versions of the data set and records of the analyses stored across multiple folders on the researcher's 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. Focusing on reproducibility from the beginning of the 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 overview of the reproducible research process: a workflow for reproducible research. Then it covers some of the key guidelines that can help make your research more reproducible.

2.1 The Big Picture: A workflow for reproducible research

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

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

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

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

Instead of starting to use the individual tools of reproducible research as soon as you learn them I recommend briefly stepping back and considering how the stages of reproducible research *tie* together overall. This will make your 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.

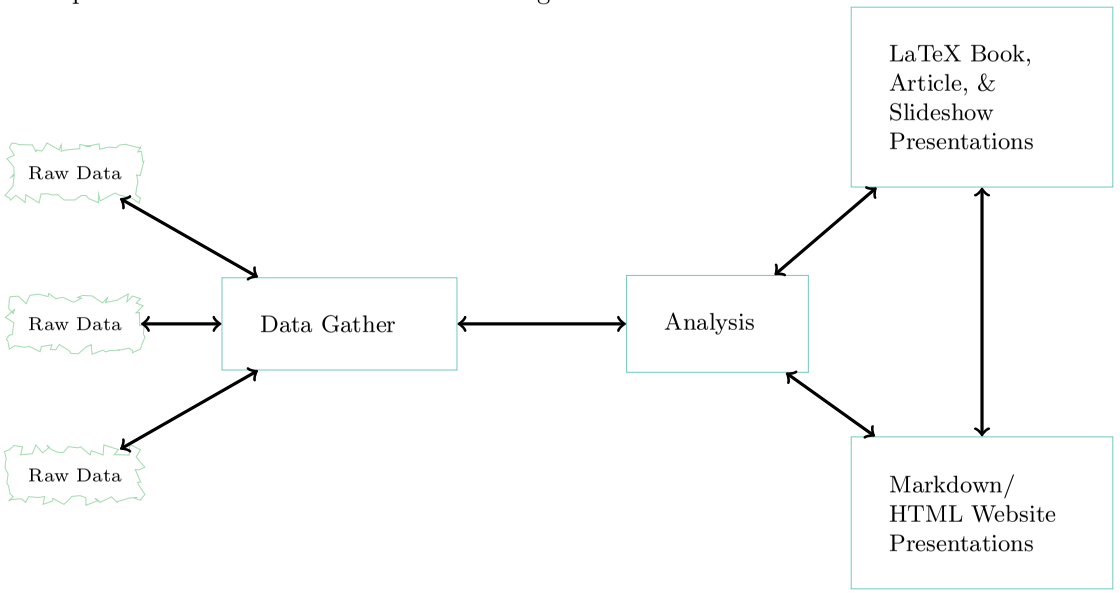
Around the edges of the figure are some of the commands you will learn to make it easier to go forwards and backwards through the process. These commands tie your research together. For example, you can use API-based R packages to gather data from the internet. You can use the `merge` command to combine data gathered from different sources into one data set. The `getURL` and `read.table` 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 tools such as 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 the main *tie commands*.

2.1.1 Reproducible Theory

An important part of the research process that I do not 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 and analysis stages. Just because I don't cover this stage of the research process doesn't mean that theory building can't and shouldn't be reproducible. It can in fact it may 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.

FIGURE 2.1
Example Workflow & Commands to Tie it Together



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 your research process and put these skills in perspective. The tips are:

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

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

2.2.1 Document everything!

In order to reproduce your research others must be able to know what you did. You have to tell them what you did by documenting as much of your research process as possible. Ideally, you should tell your readers how you gathered your data, analyzed it, and presented the results. Documenting everthing 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 you 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 how R packages are handled may vary across different operating systems, so it's important to note what system you used. Finally, you may have R set to load packages by default (see page 40 for information about packages). These packages might be necessary to run your code, but other people might not know what packages and what versions of the packages were loaded from just looking at your source code. The `sessionInfo` 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 2.15.2 (2012-10-26)
## 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] tools     grid      splines   stats     graphics  grDevices utils
##  [8] datasets  methods   base
##
## other attached packages:
##  [1] extrafont_0.13      MCMCpack_1.2-4      coda_0.16-1
##  [4] foreign_0.8-51      Zelig_3.5.5         boot_1.3-7
##  [7] xtable_1.7-0        WDI_2.2             treebase_0.0-6
## [10] ape_3.0-6           texreg_1.15         tables_0.7
## [13] RCurl_1.95-3        bitops_1.0-4.2      reshape2_1.2.2
## [16] reshape_0.8.4       quantmod_0.3-17     TTR_0.21-1
## [19] xts_0.9-1           zoo_1.7-9           Defaults_1.1-1
## [22] plyr_1.8            openair_0.7-0       memisc_0.95-38
## [25] MASS_7.3-22         lattice_0.20-10     markdown_0.5.3
## [28] knitcitations_0.1-0 bibtex_0.3-4        Hmisc_3.10-1
## [31] survival_2.37-2     httr_0.2            googleVis_0.3.3
## [34] RJSONIO_1.0-1       ggplot2_0.9.3       gdata_2.12.0
## [37] formatR_0.7         estout_1.0.1-1      devtools_0.8
## [40] countrycode_0.9     apsrtable_0.8-8     animation_2.1
## [43] knitr_0.9
##
## loaded via a namespace (and not attached):
##  [1] car_2.0-15         cluster_1.14.3     codetools_0.2-8
##  [4] colorspace_1.2-0   data.table_1.8.6   dichromat_1.2-4
##  [7] digest_0.6.0       evaluate_0.4.3     gee_4.13-18
## [10] gtable_0.1.2       gtools_2.7.0       labeling_0.1
## [13] Matrix_1.0-10      memoise_0.1        mgcv_1.7-22
## [16] munsell_0.4        nlme_3.1-106       parallel_2.15.2
## [19] pkgmaker_0.10.1    proto_0.3-10       RColorBrewer_1.0-5
## [22] rjson_0.2.11       Rttf2pt1_1.1       scales_0.2.3
## [25] stringr_0.6.2      twitteR_0.99.19    whisker_0.3-2
## [28] XML_3.95-0.1
```

Chapter 4 gives specific details about how to create files with dynamically included session information.

2.2.2 Everything is a (text) file

Your documentation is stored in files that include data, analysis code, the write up of results, and explanations of these files (e.g. data set codebooks, session info files, and so on). Ideally, you should use the simplest file format

possible to store this information. Usually the simplest file format[1] 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`) 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 files and, aided by frequent comments (see below), be able to understand how we conducted your 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 changes can be tracked 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 you want 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 character (also known as number, pound, or sharp) is ignored by R, but is readable to people who open the file. The hash character is a comment declaration character. You can use the `#`

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

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

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

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 markdown/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 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, what version of the programming language you are using or sources that may have strongly influenced the code.

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 1 March 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 its easier for you and others to understand.

There are a number of R style guides. Most of them are similar to the Google R Style Guide.[4] Hadley Wickham also has a nicely presented R style guide.[5] You may want to use the *formatR* (Xie, 2012b) 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 Explicitly tie your files together

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

[4] See: `http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`.
[5] You 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/Program	Language	Description	Chapters for Further Information
knitr	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.
`read.table`	R	Reads a table into R. You can use this to import plain-text file formated data into R.	6
`read.csv`	R	Same as `read.table` 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
`merge`	R	Merges together data frames.	7
`source`	R	Runs an R source code file.	8
`source_url`	R	From the *devtools* package. Runs an R source code file from a secure (`https`) url like those used by GitHub	8
`print(xtable())`	R	Combining the `print` & `xtable` commands creates LaTeX & HTML tables from R objects	9
`toLaTeX`	R	Converts R objects to LaTeX	2
`input`	LaTeX	Includes LaTeX files inside of other LaTeX files	12
`include`	LaTeX	Similar to `input`, but puts page breaks on either side of the `included`-ed text. Usually it is used for including chapters.	12
`includegraphics`	LaTeX	Inserts a figure into a LaTeX document.	10
`![]()`	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
Make	Shell	A shell program for automatically building many files.	6

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

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

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

Because we have so many ways to link files together there is really no need to lump many different operations into one file. So, we can make our 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 on how to use R please refer to the resources I mentioned in Chapter 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 R including:

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

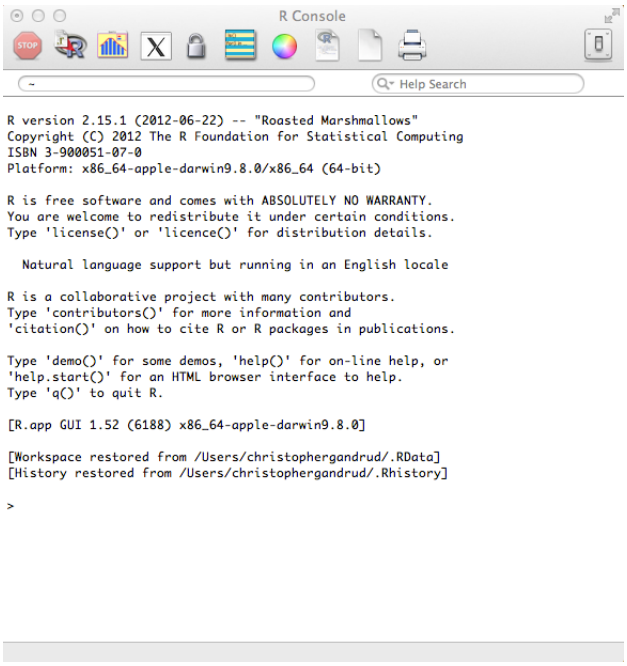
Before discussing each of these in detail let's open R and look around.[1] When you open the R GUI program you should get a window that looks something like Figure 3.1.[2] This window is the **R console**. After the startup information–information about what version of R you are using, license details,

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

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

and so 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 before, you will probably have seen it referred to as an 'object-oriented language'. What are objects? Objects are like the R language's nouns. They are things, like a vector of numbers, a data set, a word, a table of results from some analysis, and so on. Saying that R is 'object-oriented' 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.

[3] If 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 is a numeric type object. The other is a character object. We can choose almost any name we want for our objects as long as it begins with an alphabetic character and does not contain spaces.[4] Let's call our numeric object *Number*. It is a good idea to give each object a different name. Also make sure that object names are different from variable names. This will avoid many complications like accidentally overwriting an object or confusing R about what object or component you are referring to.

To put something into the object we use the assignment operator[5]: `<-`. Let's assign the number 10 to our *Number* object.

```
Number <- 10
```

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

```
Number

## [1] 10
```

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

Creating 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"
```

[4]It is common for people to use either periods (`.`) or capital letters (referred to as CamelBack) to separate words in object names instead of using spaces. For example: *new.data* or *NewData* rather than *new data*.

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

[6]The double hash is generated automatically by *knitr*. It makes easier to copy and past code into R from a presentation document by *knitr*.

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:

```
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 often want to use objects 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

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

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

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

to think of basically all R objects as composed of vectors. For example, data frames are basically multiple vectors of the same length–i.e. they have the same number of rows–attached together to form columns.

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

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

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

```
mean(StringNumObject$ChacterVect)
```

```
## 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 *CharcterVect* vector?

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

The dollar sign is called the component selector.[12] It basically extracts a part of an object. In the previous example it extracted the *CharacterVect* column from the *StringNumObject* and fed it to the `mean` command, which tried (in this case unsuccessfully) to find its mean.

We can of course use the component selector to create new objects with parts of other objects. Imagine that we have the *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, and the component name 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. I generally use the component selector. Though it creates longer code, I find code written with the component selector is easier to follow. It's always clear which object we are selecting a component from.

Add with and attach discussion.

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 lets use the *cars* data set that comes with R. Use the `head` command to get a sense of what this data set looks like.

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

```
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 and commands[13] 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 sytax 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.

[13] For 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 to. Clearly, you do not need to explicitly set an argument if you want to use it's default value.

You have to fairly precise with the syntax for your argument's values. Arguments for logical arguments must 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
```

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

[15]They 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.[16]

```
ls()
```

```
##  [1] "CharacterVect"     "DataDuplicates"    "DataNotDuplicates"
##  [4] "DataUrl"           "DispropData"       "doInstall"
##  [7] "FCLabels"          "FertConsumpData"   "FertOutliers"
## [10] "FinalCleanedData"  "FinRegulatorData"  "gm1"
## [13] "m1"                "M1"                "M1Table"
## [16] "m2"                "m3"                "MeanNumericVect"
## [19] "MergedData1"       "MergedData2"       "MoltenFert"
## [22] "MoltenFertSub"     "NBModel"           "NBModelSum"
## [25] "NBSumDataFrame"    "NBTable"           "NewNumeric"
## [28] "Number"            "NumericVect"       "outreg"
## [31] "ParentDirectory"   "Plot"              "StringNumObject"
## [34] "temp"              "toInstall"         "UDSData"
## [37] "url"               "UrlAddress"        "WideFert"
## [40] "Words"             "x1"                "x2"
## [43] "y1"                "y2"
```

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

[16]Note: your workspace will probably include different objects than this example. These are all of the objects created to knit the book up to this point.

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.[17] The only time when saving your workspace is very useful is when it includes an object that was computationally difficult and took a long time to create. In this case 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 packages and loading commands

Commands are stored in R packages. The commands we have used so far were loaded automatically by default. One of the great things about R is the many user-created packages[18] 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 packages used in this book see page xv. When you install a package, you will likely be given a list of mirrors from which you can download the package. Simply select the mirror closest to you.

Once you have installed a package you need to load it so that you can use its functions. Use the `library` command to load a package.[19] Use the following code to load the *ggplot2* package that we use in Chapter 10 to create figures.

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

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

[19] You will probably see R packages referred to as "libraries", though this is a misnomer. See this blog post by Carlisle Rainey for a discussion: `http://www.carlislerainey.com/2013/01/02/packages-v-libraries-in-r/?utm_source=`

```
library(ggplot2)
```

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

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.

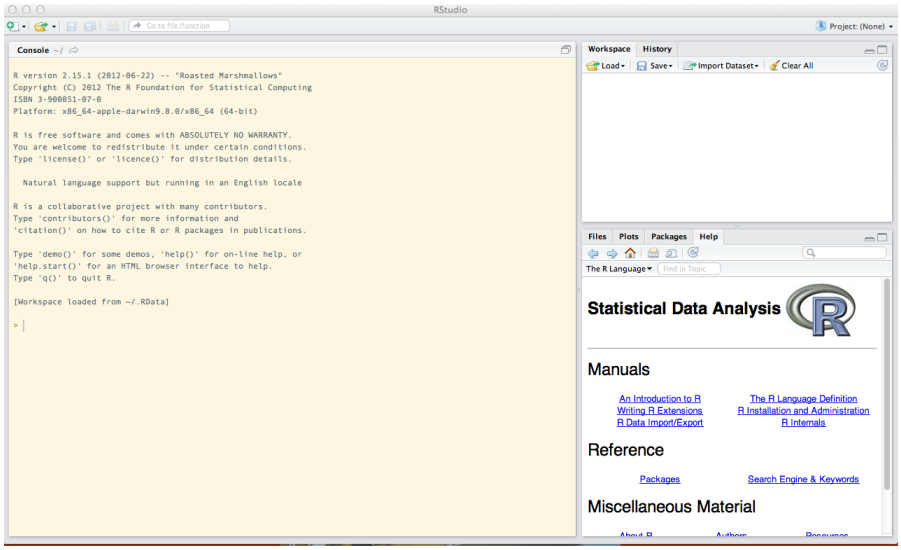
The default window

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.

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.

`rss&utm_medium=rss&utm_campaign=packages-v-libraries-in-r` (posted 2 January 2013).

FIGURE 3.2
RStudio Startup Panel



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.

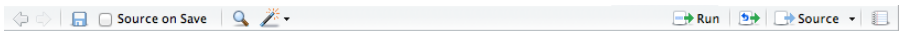
R source code files have the file extension `.R`. You can create a new source code document, which will open a new Source pane, by going to the menu bar and clicking on `File` → `New`. In this drop down menu you have the option to create a variety of different source code documents. Select the `R Source` 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 it[20] 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

[20]If 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.

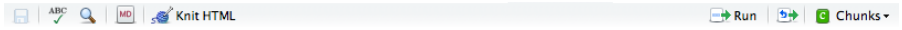
icon next to `Source` is for compiling RStudio Notebooks. We will look at RStudio Notebooks later in this chapter.

FIGURE 3.3
RStudio Source Code Pane Top Bars

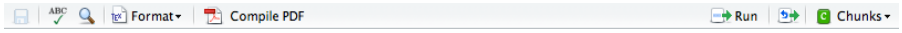
R Source Code



R Markdown Files



LaTeX Markdown Files



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 by markup language.

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 you want to include R code into your markup presentation documents, 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 an 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.[21] *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. 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

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

Sweave-style code chunk syntax is fairly baroque compared to the Rtex-style syntax. To begin a code chunk in an `Rtex` file 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)
%%
```

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-1`.

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. Also remember that 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 on a page 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 the figures.

TABLE 3.1
A Selection of *knitr* Code Chunk Options

Chunk Option Label	Type	Description
`eval`	Logical	Whether or not to run the chunk.
`echo`	Logical	Whether or not to include the code in the presentation document.
`error`	Logical	Whether or not to include errors.
`engine`	Character	Set the programming language for *knitr* to evaluate the code chunk with.
`fig.align`	Character	Aligns figures.
`fig.height`	Numeric	Sets figures' height.
`fig.width`	Numeric	Sets figures' width.
`include`	Logical	When `include=FALSE` the chunk is evaluated, but the results are not included in the presentation document.
`message`	Logical	Whether or not to include message messages.
`results`	Character	How to include results in the presentation document.
`warning`	Logical	Whether or not to include warnings.

These commands are discussed in more detail in Chapter 8.

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 document[22] 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 you can center align all of the figures in a Markdown document created *knitr* code chunks. Place the following code at the beginning of the document:

```
<<ChunkLabel>>=
# Center align all knitr figures
opts_chunk$set(fig.align='center')
@
```

3.3.4 knitr package options

Knitr package options affect how the package itself runs. For example, the `progress` option can be set as either `TRUE` or `FALSE`[23] depending on whether or not you want a progress bar to be displayed when you knit a code chunk.[24] 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).

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

```
<<ChunkLabel>>=
# Turn off knitr progress bar
opts_knit$set(progress=FALSE)
@
```

[22] In 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).

[23] It's set as `TRUE` by default.

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

3.3.5 Hooks

You can also set 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 change depending on the type of file you have open in it. RStudio uses a file's extension to determine what type of file you have open.[25] 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 click the `Compile Notebook` icon in the Source pane's top bar (see Figure 3.3).[26] 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 is a site for sharing your Notebooks over the internet. You can publish not only Notebooks, but also any *knitr* Markdown document you compile in RStudio.

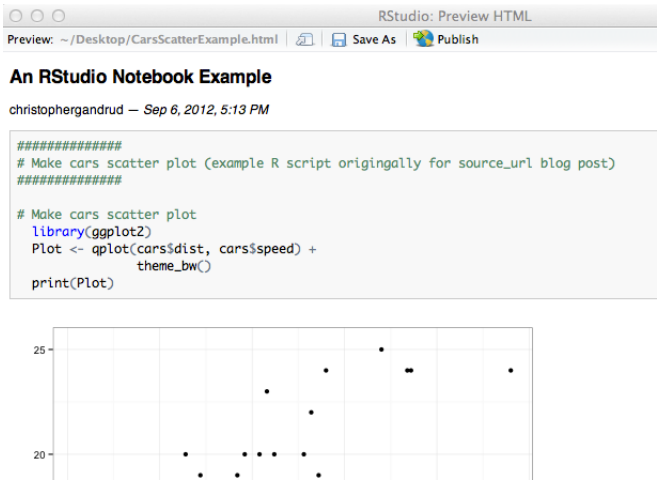
R Markdown

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

[25] You 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.

[26] Alternatively, `File` → `Compile Notebook...`

FIGURE 3.7
RStudio Notebook Example



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 on 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 buttons in the R 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 it 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).

FIGURE 3.8
Folding Code Chunks in RStudio

Not Folded

Folded

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 instead of `MD`. This actually inserts LaTeX formatting commands into your document for things such as section headings and bullet lists. These commands 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 to knit `.Rnw` files you should click on `Tools`

in the RStudio menu bar then click on `Options` window. Once the **Options** window opens, click on the `Sweave` button. Select `knitr` from the drop down menu for "Weave files using:". Finally, click `Apply`.[27]

3.3.7 knitr & R

As *knitr* is a regular R package, you can of course knit documents in R (or using the console in RStudio). All of the *knitr* syntax 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 an example Markdown file *Example.Rmd* you first set us the `setwd` command to set the working directory (for more details see Chapter 4) to the the folder where the *Example.Rmd* file is located. In this example it is located on the desktop.[28]

```
setwd("~/Documents/")
```

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

In this example, using the *knit* command only creates a Markdown file and not an HTML file, as 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.

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

This is a bit tedious. Luckily, there is a command in the *knitr* package that

[27] In the Mac version of RStudio, you can also access the `Options` window via `RStudio` →`Preferences` in the menu bar.

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

combines `markdownToHTML` 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 simplify this process by using the `knit2pdf` command to compile a PDF file from a `.Rnw` or `.Rtex` document.

Appendix: knitr and Lyx

You may be more comfortable using a what-you-see-is-what-you-get editor, similar to Microsoft Word. Lyx is a WYSIWYG LaTeX editor that can be used with *knitr*. I don't cover Lyx in detail in this book, but here is a little information to get you started.

Set Up

To set up Lyx so that it can compile `.Rnw` files click `Document` in the menu bar then `Settings`. In the left-hand panel the second option is `Modules`. Click on `Modules` and select `Rnw (knitr)`. Click `Add` then `Ok`. Now, compile your LaTeX document in the normal Lyx way.

Code Chunks

Enter code chunks into TeX Code blocks within your Lyx documents. To create a new TeX Code block select `Insert` →`TeX Code`.

4

Getting Started with File Management

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

- Explicitly tie your files together,
- Have a plan to organize, store and make your files available.

Apart from the times when 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, benefit from work you have already done and collaborate with others.

Using tools such as R, *knitr*, and markup languages like LaTeX requires fairly detailed knowledge of where files are stored in your computer. Handling files reproducibly may require you to use command line tools to access and organize your files. R and Unix-like shell programs allow you to control files–creating, deleting, relocating–in powerful and really reproducible ways. By typing these commands you are documenting every step you took. This is a major advantage over graphical user interface-type systems where you organize files by clicking and dragging them with the cursor. However, text commands require you to know your files' specific addresses–their file paths.

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 how to 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. The skills you will 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 stored in the cloud.

4.1 File paths & naming conventions

All of the operating systems covered in this book organize files in hierarchical directories, also know as 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, as 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.

On Windows computers you can have multiple root directories, one for each storage device or partition of a storage device. 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 Macs, the root directory is simply denoted by a forward slash (`/`) with nothing before it.

4.1.2 Subdirectories & parent directories

You will probably not store all of your files in 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. Directories inside of other directories are child directories or subdirectories of a 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`.[2] 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.[3] Escape characters tell R to interpret the next character or sequence of characters differently. For example, on page 73 you'll see how `\t` can be interpreted by R as a tab rather than the letter "t". Add another escape character to neutralize the escape character so that R interprets it as a backslash. In other words use an escape character to the escape character. Another option for writing Windows file names in R is to use one forward slash (`/`).

[1] To simplify things, I use the terms 'directory' and 'folder' interchangeably in this book.

[2] For 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`

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

On Unix-like systems, including Mac computers, directories are indicated with a forward slash (`/`). The file path of the *Data* file on a Unix-like system would be: `/ExampleProject/Data`

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

4.1.3 Spaces in directory & file names

It is generally good practice to avoid putting spaces in your file and directory names. For example, I called the example project parent directory "ExampleProject" rather than "Example Project". Spaces in file and directory names can sometimes create problems for compter programs trying to read the file path. It may believe that the space indicates that the path name has ended. To make multi-word names easily readable without using spaces, adopt a convention such as CamelBack. In CamelBack new words are indicated with capital letters, while all other letters are lower case. For example, "ExampleProject".

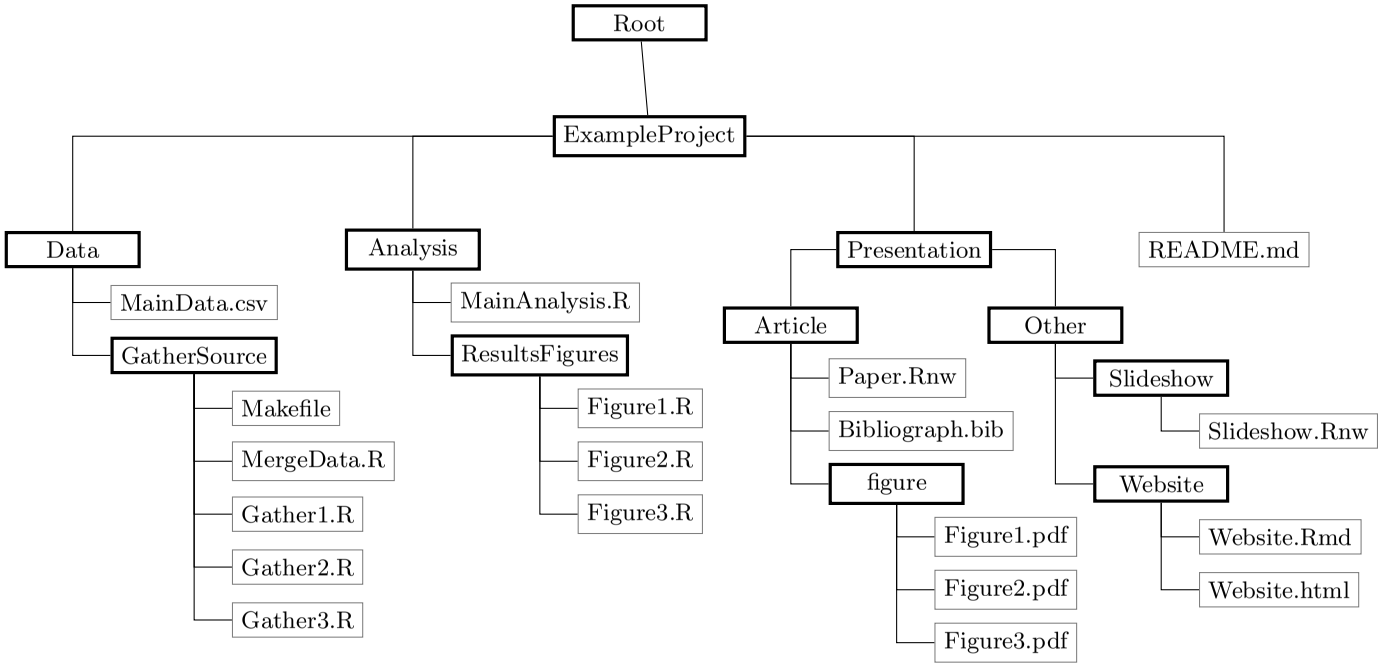
4.1.4 Working directories

When you use R 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 looks for files and other directories, unless you tell it to look elsewhere. It is also where it will save files. Later in this chapter we will cover commands for finding and changing the working directory.

4.2 Organizing your research project

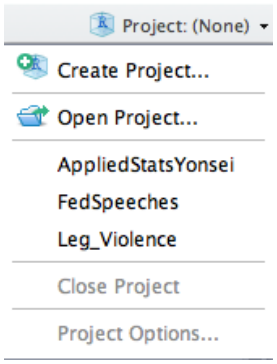
Figure 4.1 gives an example of how the files in a simple reproducible research project could be organized. The project's main parent directory is called *ExampleProject*. Inside this directory are three subdirectories: a data gathering directory, an analysis directory, and a presentation directory. Each of these directories contains 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 the main subdirectories of *ExampleProject* 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 like its title, author(s), topic, any copyright information, and so on. It should also indicate how the folders in the project are organized and give instructions 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 the `README` file is written using Markdown, it will also be properly formatted. Figure 5.2 shows an example of this.

FIGURE 4.2
An Example RStudio Project Menu



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 use the `toLatex` command in a code chunk. The code chunk should have the option `results='asis'`. The code is:

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

4.3 Setting directories as RStudio Projects

If you are using RStudio, you may want to organize your files as 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 turn 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 create a project from a directory already under version control (we'll do this at the end of 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 for a number of reasons:

- the project is listed in RStudio's Project menu where it can be opened easily (see Figure 4.2).
- when you open the .Rproj file RStudio automatically sets the working directory to the project's directory and loads the 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,
- it helps you version control your files.

4.4 R file manipulation commands

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

`getwd`

To find your current working directory use the `getwed` 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.

```
list.files()

## [1] "chapter4.Rnw" "images4"
```

You can see that the *Chapter4* folder has the file *chapter4.Rnw* (the markup file used to create this chapter) and a child directory called *images4* where I stored the original versions of the figures show in this chapter.

`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 you may want to create a new directory. You can use the `dir.create` command to do this.[4] 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:

[4]Note: you will need the correct system permissions to be able to do this.

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

cat

If you want to create a new file and put text into it use the `cat` (concatenate and print). For example to create a new file in the current working directory called *ExampleEcho.md* that includes the text "Reproducible Research with R and RStudio" type:

```
cat("Reproducible Research with R and RStudio",
    file = "ExampleCat.md")
```

You can use `cat` to also print the contents of a one or more objects to a file. In this example we created a Markdown formatted file by using the `.md` file extension. We could of course change the file extension to `.R` to set it as an R source code file, `.Rnw` to create a *knitr* LaTeX file and so on.

The `cat` command will overwrite existing files with the new contents. To add the text to existing files use the `append = TRUE` argument.

```
cat("More Text", file = "ExampleCat.md",
    append = TRUE)
```

unlink

Finally, you can use the `unlink` command to delete files and directories.

```
unlink("C:\\ExampleProject\\SourceCode.R")
```

Important the `unlink` command permanently deletes files, so be very careful using this command.

file.rename

You can use the `file.rename` to obviously rename a file. It can also be used to move a file from one directory to another. For example, imagine that we

want to move the *ExampleCat.md* file from the directory *ExampleProject* to one called *MarkdownFiles* that we already created.[5]

```
file.rename(from = "C:\\ExampleProject\\ExampleCat.md",
            to = "C:\\MarkdownFiles\\ExampleCat.md")
```

`file.copy`

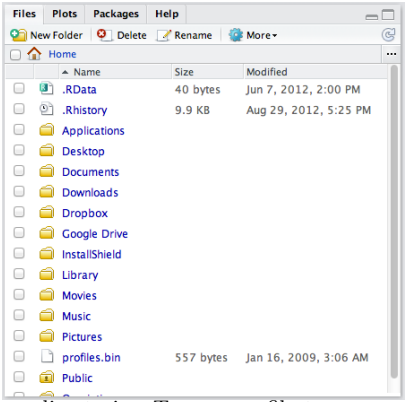
The `file.rename` fully moves a file from one directory to another. To copy the file to another directory use the `file.rename` command. It has the same syntax as `file.rename`:

```
file.copy(from = "C:\\ExampleProject\\ExampleCat.md",
            to = "C:\\MarkdownFiles\\ExampleCat.md")
```

4.5 Unix-like shell commands for file management

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 in which you to type commands to interact with your computer's operating system. We will especially return to shell commands in the next chapter when we discuss Git version control and makefiles for compiling large documents and batch reports as well as the command line program Pandoc (Chapter 12). We don't have enough space to fully introduce shell

FIGURE 4.3
The RStudio Files Pane



[5]The `file.rename` command won't create new directories. To move a file to a new directory you will need to create the directory first with `dir.create`.

programs or even all of the commands for manipulating files. We are just going to cover some of the basic and most useful commands 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 commands discussed in this chapter should work in both Unix-like shells and Windows Powershell. That being said, I have not tested all of them in Powershell.

It's important at this point to highlight a key difference between R and Unix-like shell syntax. In shell commands you don't need to put parentheses around your arguments. For example if I want to change my working directory to my Mac Desktop in a shell using the `cd` command I simply type:[6]

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

In this example `Me` is my user name.

`cd`

As we just saw, to change the working directory in the shell just use the `cd` (change directory) command. If you are in a child directory and want to change the working directory to its parent you can simply type:

```
cd -
```

If, for example, our current working directory was */User/Me/Desktop* and we typed `cd -` then the working directory would change to */User/Me*.

`pwd`

To find your current working directory use the `pwd` command (present working directory). This is essentially the same as R's `getwd` command.

[6] Many shell code examples in other sources include the shell prompt, like the `$` in Bash. It's like R's > prompt. I don't include the prompt in code examples in this book because you don't type them.

```
pwd

## /Users/Me/Desktop
```

`ls`

The `ls` (list) command works very similarly to R's `list.files` command. It shows you what is in the current working directory.

```
ls

## chapter4.Rnw images4
```

`mkdir`

Use `mkdir` to create a new directory. For example, if I wanted to create a directory in my Linux root directory called *NewDirectory* I would type:

```
mkdir /NewDirectory
```

If running this code gives you an error message like this:

```
mkdir: /NewDirectory: Permission denied
```

you simply need to use the `sudo` command to run the command with higher privileges.

```
sudo mkdir /NewDirectory
```

Running this code will prompt you to enter your administrator password.

`echo`

There are a number of ways to create new files in Unix-like shells. One of the simplest ways is with the `echo` command. This command simply prints its arguments. For example:

```
echo Reproducible Research with R and RStudio
## Reproducible Research with R and RStudio
```

If you add the greater than symbol (`>`) after the text you want to print then a file name, `echo` will create the file (if it doesn't already exist) in the current working directory then print the text into the file.

```
echo Reproducible Research with R and RStudio > ExampleEcho.md
```

Using only one greater than sign will completely erase the *ExampleEcho.txt* file's contents and replace them with `Reproducible Research with R and RStudo`. To add the text at the end of an existing file use two greater than signs (`>>`).

```
echo More text. >> ExampleEcho.md
```

There is also a `cat` shell command. It works slightly differently than the R version of the command and I don't cover it here.

`rm`

The `rm` command is similar to R's `unlink` command. It removes (deletes) files or directories. Again, be careful when using this command, because it permanently deletes the 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.

```
rm ExampleEcho.md
```

`mv`

To move a file from one directory to another with the shell use the `mv` (move) command. For example, to move the file *ExampleEcho.md* from *ExampleProjects* to *MarkdownFiles* both in the root directory:

```
mv /ExampleProject/ExampleEcho.md /MarkdownFiles
```

Note that like the /MarkdownFiles directory must already exist. You can also use the `mv` command to simply rename files, just like the R command `file.rename`.

`cp`

The `mv` command completely moves a file from one directory to another. To simply copy a version of the file to a new directory use the `cp` command. The syntax is similar to `mv`:

```
cp /ExampleProject/ExampleEcho.md /MarkdownFiles
```

`system` *(R command)*

You can run shell commands from within R using R's `system` command. For example, to run the `echo` command from within R type:

```
system("echo Text to Add > ExampleEcho.md")
```

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. This pane allows us to navigate to specific files and folder, delete and rename files. To select a folder as the working directory tick the dialog box next to the file then click the `More` button and select `Set As Working Directory`. Under the `More` button you will also find options to `Move` and `Copy` files.

The **Files** pane is a GUI, so our actions in the Files pane are not as easily reproducible as the commands we learned earlier in this chapter.

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 your findings. A useful way to make your files accessible is to store them on a cloud storage service[1] (see Howe, 2012). This chapter describes in detail two different cloud storage services–Dropbox and GitHub–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** 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 mobile devices. For example, we may use a computer at work to run computationally intensive analysis, while editing our presentation document on a tablet computer while riding the train to the office. So, we need to be able to **access** our files from multiple devices 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 earlier versions. Also, when working on a collaborative project, one of the authors may accidentally delete something in a file that another author needed. 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 want to.

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:

[1] These 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, lets 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 sets[2] a plain-text format like comma-separated values (`.csv`) or tab-separated values[3] (`.tsv`) are good ways 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 as well as spreadsheet programs like Microsoft Excel. Using text file formats helps future proof your research. Version control systems that track changes to text–like Git–are also very effective version control systems for 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 = ",")
```

[2] I 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 from RDataMining: `http://rdatamining.wordpress.com/2012/05/06/online-resources-for-handling-big-data-and-parallel-computing-in-r/` (posted 6 May 2012). One popular service for large file storage is Amazon S3 (`http://aws.amazon.com/s3/`). I haven't used this service and can't suggest ways to integreate it with R.

[3] Sometimes this format is called tab-delimited values.

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 `write.csv`. This command simply makes it so that you don't have to write `sep = ","`.[4]

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 able to save data in a wide variety of other file formats, mostly through the *foreign* package (see Chapter 6). 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.[5] 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,[6] the Dataverse Network Project,[7], figshare[8] and RunMyCode.[9] 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 Dropbox[10] and Google Drive.[11] These services not only store your data in the cloud, but also provide ways to share files. They even include basic version

[4] `write.csv` is a 'wrapper' for *write.table*.

[5] Dropbox 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.

[6] `https://cloud.sdsc.edu/hp/index.php`

[7] `http://thedata.org/`

[8] `http://figshare.com/`

[9] `http://www.runmycode.org/`

[10] `http://www.dropbox.com/`

[11] `https://drive.google.com/`

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

5.2.2 Storage

When you sign up for Dropbox and install the program[12] 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 when you sign up to the service you'll receive a limited amount of storage space for free; usually a few gigabytes. This is probably enough storage space for a number of text file based research projects.

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 me manually (point-and-click) downloaded onto your computer and then loaded into R. Files in the Dropbox *Public* folder can be downloaded directly into R. Files not in the *Public* folder have to be downloaded manually.[13] 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 data frames (see Chapter 5) or the `source` command for source code files (see Chapter 8).

To give you a preview of how to download data on financial regulator type directly into R from Dropbox, try downloading a data file from my Public folder. The data set's URL is: `http://dl.dropbox.com/u/12581470/code/Replicability_code/Fin_Trans_Replication_Journal/Data/public.fin.msm.model.csv`.[14] I've used the URL shortening service bitly[15] to make this link fit on the page.

[12] See `https://www.dropbox.com/downloading` for downloading and installation instructions.

[13] This is not completely true. It could be possible to create a web scraper that could download data from a file not in your *Public* folder. However, this is a hassle and not practical, especially given that accessing files from the *Public* folder is so easy.

[14] This data is from Gandrud (2012)

[15] See `https://bitly.com/`.

```
# Download data on Financial Regulators
# stored in a Dropbox Public folder
FinRegulatorData <- read.table("http://bit.ly/PhjaPM",
                      sep = ",", header = TRUE)

# Show variables in Data
names(FinRegulatorData)

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

Storing files in the *Public* folder clearly makes replication easier because R can access these files easily.

When 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-link icon appear on the far right. Clicking on this icon will get you the link.

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 or upload them through the website. 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 a new version is created on Dropbox. 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. You can see a list of changes made to files in your Dropbox folder by going to the website and clicking on `Events`.

Note that with a free Dropbox account, previous versions of a file are only stored for **30 days**. To be able to save previous versions for more than 30 days you will need a paid account. For more details see: `https://www.dropbox.com/help/113/en`.

5.2.4 GitHub

Dropbox adequately meets our four basic criteria for reproducible data storage. It is easy to set up and use. GitHub meets the criteria and more, especially when it comes to version control. It is, however, less straightforward at first. In this section we will learn enough of the basics to get you started using GitHub to store, access, collaborate on, and version control your research.

GitHub is an interface and cloud hosting service built on top of the Git version control system. Git does the version control. GitHub stores the data remotely as well as providing a number of other features, some of which we look at below. GitHub was not explicitly designed to host research projects or even data. It was designed to host "socially coded" computer programs–in what Git calls "repositories"–repos for short–by making it easy for a number of collaborators to work together to build computer programs. 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 for novice users to set up and become familiar with. So we need good reasons to want to invest the time needed to learn GitHub. Here is a list of GitHub's advantages over Dropbox for reproducible research that will hopefully convince you to get started using it:

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 5.2). An example feature is that it automatically

renders Markdown files called *README.md*[16] in a GitHub directory on the repository's website. This makes it easy for independent researchers to find the file and read it.

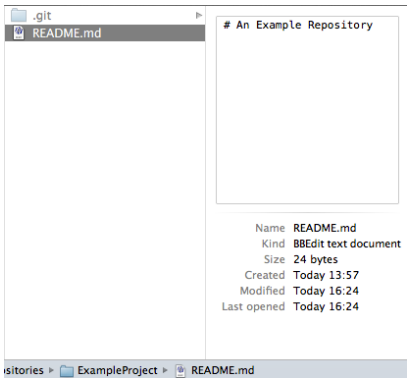
- GitHub can create and and host a website for your research project that you could use to present the results, not just the replication files.

Collaboration

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. It also stores the issues so you have a full record.
- Each repository can also host a wiki that, for example, could explain in detail how certain aspects of a research project were done.

FIGURE 5.1
A Basic Git Repository with Hidden *.git* Folder Revealed



- Anyone can suggest changes to files in a public repository. These changes can be accepted or declined by the project's 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 files' 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.

[16] You can use a variety of other markup languages as well. See `https://github.com/github/markup`.

- 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 multiply. Git's version control system only creates a new version when you tell it to.

- All files in on Dropbox are version controlled. Git allows you to ignore specific files.

- Unless you have a paid account, previous file versions in Dropbox disappear after 30 days. GitHub stores previous versions indefinitely for all account types.

- 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 at the same time. GitHub identifies conflicts and lets you reconcile them.

- Git is directly integrated into RStudio Projects.[17]

5.2.4.1 Setting up GitHub: Basic

There are at least three ways to use Git/GitHub on your computer. You can use the command line version of Git. It's available for Mac and Linux (in the Terminal) as well as Windows (through Git Bash).[18] You can also use the Graphical User Interface GitHub program. Currently it's only available for Windows and Mac. RStudio also has GUI-style Git functionality for RStudio Projects. In this section I focus on how to use the command line version, because it will help you understand what the GUI versions are doing and allow you to better explore more advanced Git features not covered in this book. In the next section I will mention how to use Git with RStudio Projects.

The first thing to do to setup GitHub is go to their website (`https://github.com/`) and sign up for an account. Second, you should go to the following website for instructions on setting up Git: `https://help.github.com/articles/set-up-git`. The instructions on that website are very comprehensive, so I'll direct you there for the full setup information. Note that installing the GUI version of GitHub also installs Git and, on Windows, Git Bash.

5.2.4.2 Version Control with Git

Git is primarily a version control system, so we will start our discussion of how to use it by looking at how to version your repositories.

[17] RStudio also supports the Subversion version control system, but I don't cover that here.

[18] The interface for Git Bash looks a lot like the Terminal or Windows PowerShell.

Setting up Git repositories locally

You can setup a Git repo on your computer with the command line.[19] I keep my repositories in a folder called *git_repositories*.[20] It has the root folder as its parent. Imagine that we want to set up a repository in this directory for a project called *ExampleProject*. Initially it will have one README file called *README.md*. To do this we would first type into the Terminal for Mac and Linux computers:

```
# Make new directory 'ExampleProject
mkdir /git_repositories/ExampleProject

# Change to directory 'ExampleProject'
cd /git_repositories/ExampleProject

# Create new file README.md
echo "# An Example Repository" > README.md
```

So far we have only made the new directory and set it as our working directory (see Chapter 4). All of the examples in this section assume your current working directory is set to the repo. Then with the `echo` Shell command we created a new file named *README.md* that includes the text `# An Example Repository`. Note that the code is basically the same in Windows PowerShell, though obviously the directory names are different. Also, you don't have to do these steps in the command line. You could just created the new folders and files the same what that you normally do with your mouse in your GUI operating system.

Now that we have a directory with a file we can tell Git that the we want to treat the directory *ExamplProject* as a repository and that we want to track changes made to the file *README.md*. Use Git's `init` (initialize) command to set directory as a repository. See Table 5.1 for the list of Git commands covered in this chapter.[21] Use Git's `add` command to add a file to the Git repository. For example,

[19] Much of the discussion of the command line in this section is inspired by Nick Farina's blog post on Git (see `http://nfarina.com/post/9868516270/git-is-simpler`, posted 7 September 2012).

[20] To follow along with this code you will first need to create a folder called *git_repositories* in your root directory. Throughout this section I use Unix file path conventions.

[21] For a comprehensive guide to Git commands see `http://git-scm.com/`.

```
# Initialize the Git repository
git init

# Add README to the repository
git add README.md
```

You probably noticed that you always need to put `git` before the command. This tells the command line what program the command is from. When you initialize a folder as a Git repository a hidden folder called *.git* is added to the directory (see Figure 5.1). This is where all of your changes are kept. If you want to add all of the files in the working directory to the Git repository type:

```
# Add all files to the repository
git add .
```

Now when we want Git to track changes made to files added to the repository we can use the `commit` command. In Git language we are "committing" the changes to the repository.

```
# Commit changes
git commit -a -m "First Commit, created README file"
```

The `-a` (all) command commits changes made to all of the files that have been added to the repository. You can include a message with the commit using the `-m` command like: `"First Commit, created README file"` Messages help you remember general details about individual commits. This is helpful when you want to revert to old versions. **Remember:** Git only tracks changes when you commit them.

Finally, you can use the `status` command for details about your repository, including uncommitted changes. Generally it's a good idea to use the `-s` (short) argument, so that the output is more readable.

```
# Display status
git status -s
```

TABLE 5.1
A Selection of Git Commands Used in this Chapter

Command	Description
`add`	Add a file to a Git repository.
`branch`	Create and delete branches.
`checkout`	Checkout a branch.
`clone`	Clone a repository (for example the GitHub version) into the current working directory.
`commit`	Commit changes to a Git repository.
`fetch`	Download objects from the remote (or another) repository.
`.gitignore`	Not a git command, but a file you can add to your repository specifying what files/file types for Git to ignore.
`init`	Initialize a Git repository.
`log`	Show commit history.
`merge`	Merges two or more commits/branches together.
`pull`	`fetch` data from a remote repository and try to `merge` it with your commits.
`push`	Add committed changes to a remote Git repository, i.e. GitHub.
`remote add`	Add a new remote repository to an existing project.
`rm`	Remove files from Git version tracking.
`status`	Show the status of a Git repository including uncommitted changes made to files.

Note: when you use these commands in the command line, you will need to precede them with `git` so the shell knows where they are from.

It is useful to step back for a second and try to understand what Git is doing when you commit your changes. In the hidden *.git* folder Git is saving all of the information in compressed form from each of your commits into a sub-folder called *Objects*. Commit objects[22] are everything from a particular commit. I mean everything. If you delete all of the files in your repository (except for the *.git* folder) you can completely recover all of the files from your most recent commit with the `checkout` command:

```
# Checkout latest commit
git checkout --
```

You can also change to a particular commit of a particular file with `checkout`. Simply replace the `---` with the commit reference. The reference is easy to find and copy in GitHub.[23] Click on the link that lists the number of repo commits on the right-hand side of the repo's webpage. 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. In the next section we'll briefly look at how to switch versions in RStudio.

Branches

Sometimes you may want to work on an alternative version of your projects and then merge changes to this alternative version back into the main one. For example the main version could be the most stable current copy of your research, while the alternative version could be a place where you test out new ideas. Git allows you to create a new *branch* (alternative version of the repo) which can be merged back into the *master* (main) branch. To see what branch you are using type:

```
# Show git branch
```

[22]Other Git objects include trees (sort of like directories), tags (bookmarks for important points in a repo's history) and blobs (individual files).

[23]You can of course search your commit history and roll back to a previous commit using only the command line. To see the commit history use the `log` command (more details at `http://git-scm.com/book/en/Git-Basics-Viewing-the-Commit-History`). When a repo has many commits, this can be a very tedious command to use, so I highly recommend the GUI version of GitHub or the repo's GitHub website.

```
git branch
## * master
```

To create a new branch use, simply enough, the `branch` command. For example, to create a new branch called *Test*:

```
# Create Test branch
git branch Test
```

You can now use `checkout` to checkout the branch.[24] Here is a short cut for creating and checking out the branch:

```
# Create and checkout Test branch
git checkout -b Test
```

To merge changes you commit in the *Test* branch to the *master* first checkout the *master* branch then:

```
# Merge master and Test branches
git merge Test
```

Not, when you merge a branch you may encounter conflicts in the files that make it impossible to smoothly merge the files together. Git will tell you what and where these are, you then need to decide what to keep and what to delete.

Having Git ignore files

There may be files in your repository that you do not want to keep under version control. Maybe this is because they are very large files, cache files from *knitr* and other files that are they byproduct of compiling an R LaTeX document. To have Git ignore a file simply create a file called *.gitignore*.[25] You can either put this file in the repository's parent directory to create a *.gitignore* file for the whole repository. You can also place one in a child

[24] To delete the *Test* branch use the `-d` argument, i.e. `git branch -d Test`.
[25] Note that like *.git*, *.gitignore* files are hidden.

directory to ignore files only in that directory. In the *.gitignore* file add ignore rules by simply including the names of the files that you want to have Git ignore.

For example, a *.gitignore* file that is useful for ignoring files that are the byproduct of compiling an R LaTeX file would look something like this:

```
# Ignore LaTeX compile byproduct files #
########################################
*.aux
*.bbl
*.blg
cache/*
figure/*
*.log
*.pdf
*.gz
*.tex
```

The asterisks (`*`) is a "wildcard" and stands for any character. In other words, it tells Git to look for files with any name that end in the specified file extensions. This is faster than writing out the full name of every *.tex* file, for example. It also makes it easy to copy the rules into new repos. You'll also notice the `cache/*` and `figure/*` rules. These tell Git to ignore all of the files in the *cache* and *figure* directories. These files are the product of caching code chunks and creating figures, respectively.

Note, Git will not ignore files that have already been commited to a repository. To ignore these files you will first need to remove them from Git with Git's `rm` (remove) command. If you wanted to remove a file called *ExampleProject.tex* from version tracking so that it could be ignored type:

```
# Remove ExampleProject.tex from Git version tracking
git rm --cached ExampleProject.tex
```

Using the `--cached` argument tells Git not to track the file, but not delete it.

For more information on *.gitignore* files see GitHub's reference page on the topic at: `https://help.github.com/articles/ignoring-files`.

5.2.4.3 Remote Storage on GitHub

So far we've been using repos stored locally. To create repositories stored on GitHub, first create a new repository through the GitHub website. On your

main GitHub account page click the `New repository` button. In the next page that appears give the repository a name, brief description, choose whether to make it public or private. Also click the check box to to create a *README.md* file in the repository. When you click `Create Repository` you will be directed to the repository's GitHub page.

To download the repository onto your local computer you need to "clone" it. The repo's GitHub page contains a button called `Clone in ...`.[26] Clicking this will open GUI GitHub (if it is installed) and prompt you to specify what directory on your computer you would like to clone the repository into. You can also use the `clone` command. Image that the URL for a repo called *Example Project* is `https://github.com/USERNAME/ExampleProject.git`. To clone it into the */git_repositories* directory type:[27]

```
# Change working directory
cd /git_repositories/

# Clone ExampleProject
git clone https://github.com/USERNAME/ExampleProject.git
```

Pushing files to a GitHub repo

So far we have learned how to add and commit changes to a local Git repo. To add your commits to the remote GitHub repository use the `push` command. For example, if your current working directory is the Git repo you want to push and you have already added/committed the changes you want to include in the remote repo type:

```
# Add changes to the GitHub remote master branch
git push origin master
```

The `origin` is simply your locally stored repository. If you have not set up password caching[28] you will now be prompted to give your GitHub user name and password.

[26] The button will indicate the operating system you are using. For example in Figure 5.2 it says `Clone in Mac`.

[27] If you are on the repo's webpage you can copy it's URL by clicking on the `copy to clipboard` icon next to the URL on the same line as the `Clone in ...` button.

[28] See `https://help.github.com/articles/set-up-git` for more details.

5.2.4.4 Accessing on GitHub

Downloading into R & viewing files

In Chapter 6 we learn how to load data from a text file stored on GitHub directly into R. In general it is similar to loading data from Dropbox Public folders, but because the files are stored on a secure server it requires a few extra steps. See page 103 for more details.

The GitHub web user interface also allows you, your collaborators (see below) or if the repo is public, anyone to look at text files with a web browser. Collaborators can actually also create, modify and commit changes in the web user interface. This can be useful for making small changes, especially from a mobile device without a full installation of Git. Anyone with a GitHub account can make changes to files in a public repository on the repo's website. Simply click the `Edit` button above the file and make edits. If the person making the edits is not a designated collaborator, their edits will be sent to the repository's owner for approval.[29]. This can be a useful way for independent researchers to catch errors and directly address them.

5.2.4.5 Collaboration with GitHub

Repositories can have official collaborators that can make changes to files in the repo. Public repositories can have unlimited collaborators. Anyone with a GitHub account can be a collaborator. To add a collaborator to a repository you created click on the `Settings` button on the repository's website (see Figure 5.2). Then click the `Collaborators` button on the left-hand side of the page. You will be given a box to `Add a friend`. You can search for other people's GitHub user names. If your collaborator doesn't have one, they will have to create a new account. Once you add someone as a collaborator they can clone the repository onto their computer as you did earlier.

Syncing repository

What do you do if you and your collaborators are both making changes to the files? To avoid too many conflicts, it is a good idea to sync your local repository with the remote repository **before** you push your commits to GitHub. Use the `pull command` to sync your local and remote repository. First add and commit your changes, then type:

```
# Sync repository
git pull
```

If you have merge conflicts you will probably want to resolve these in the

[29]This is called a `pull` in git terminology. See the next section for more details.

individual files and commit the changes. Finally, push your merged changes up to the remote repository.

5.2.5 Summing up the GitHub workflow

We've covered a lot of ground in this section. Let's sum up the basic GitHub workflow you will probably follow. Let's assume you already have a repo on GitHub that you've cloned to your computer.

1. add any changes you've made with `git add .`,
2. `commit` the changes,
3. `pull` your collaborators' changes from the GitHub repo and resolve any merge conflicts,
4. `push` your changes to GitHub.

More Practice with Command Line Git & 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`.

5.3 RStudio & GitHub

RStudio can accomplish many of the Git commands covered in the previous section with a graphical user interface.

5.3.1 Set Up

You can Git initialize new RStudio projects as repositories, Git initialize existing projects, and create RStudio Projects from cloned repos.

New Git version controlled repository

To create a new project with Git version control go to `Project` in the RStudio menu bar. Then click `Create Project ...`. Select `New Project`. Enter the Projects name and desired directory. Make sure to check the dialog box for `Create a git repository for this project`.

Git initialize existing projects

If you have an existing RStudio Project and want to add Git version control to it first go to `Project` in the RStudio menu bar. Then select `Project Options`

`...`. Select the `Git/SVN` icon. Finally select `Git` from the drop down menu for `Version Control System:`.

You can push an existing repository stored on your computer to a new remote one on GitHub, for example. To do this first create a new repo on GitHub with the same name as your RStudio project like we did earlier.[30] Then copy the remote repository's URL like we saw before when we cloned a repository. Open a new shell from withing RStudio. To do this, click the `Shell` button in Git tab's `More` drop down menu. Then use the Git's `remote` command with the `add` argument. For example, if your repository's GitHub URL is `https://github.com/USERNAME/ExampleProject.git` then type:

```
# Add a remote (GitHub) repository to an existing repo
git remote add origin https://github.com/USERNAME/ExampleProject.git
```

This will add the local `origin` repository to GitHub. Finally, push the repository to GitHub:

```
# Push local repository to GitHub for the first time
git push -u origin master
```

This is almost identical to the `push` command we saw earlier. The only difference is the addition of the `-u` (upstream tracking) argument. This argument simply adds a tracking reference for the upstream (GitHub) repository branches.

Once you have done this you will not need to use the shell to push changes to GitHub. You can use the `Push` command in the `More` drop down menu in your project's *Git* tab. Just like when pushing from the shell, you will be asked for your GitHub user name and password.

Clone repository into a new project

To create a new project from a cloned GitHub Repository again go to `Project` in the RStudio menu bar. Then click `Create Project ...`. Select the `Version Control` option and then `Git`. Finally paste the repository's URL in the field called `Repository URL:`, enter the directory you would like to locate the cloned repo in and click `Create Project`.

[30] If your project already has a README file make sure you don't have GitHub initialize the repository with another README file.

5.3.2 Using Git in RStudio projects

When you open a project with a Git repository in RStudio you will see a new tab in the same pane as *Workspace/History* (see Figure 5.3).

This tab allows you do to many of the things we covered in the previous section. To add and commit files to the repository click on the dialog boxes next to the file names. In the bottom panel of Figure 5.3 you can see that I've created a new R file called *ExampleScript.R* and clicked the dialog box next to it, along with the other files. The yellow question marks in the top panel have now become green A's. Clicking `Commit` opens a new window called **Review Changes** where you can commit the changes. Simply write a commit message in the box called *Commit Message* on the upper-right side of the **Review Changes** window and click `Commit`. If you add files names to the *.gitignore* files, they will not show up in RStudio's *Git* tab.

If you are using a GitHub repo that is associated with a remote repository on GitHub, for example, you can push and pull it with the `Pull Branches` and `Push Branches` buttons in the `More` drop down menu. You can also use the same icons in the **Review Changes** window.

The Git tab also allows you to change branches, revert to previous commits, view your commit history. You can also always use the `More` → `Shell ...` option to open a new Shell with the project set as the working directory to complete any Git task you might want.

FIGURE 5.2
Part of this Book's GitHub Repository Webpage

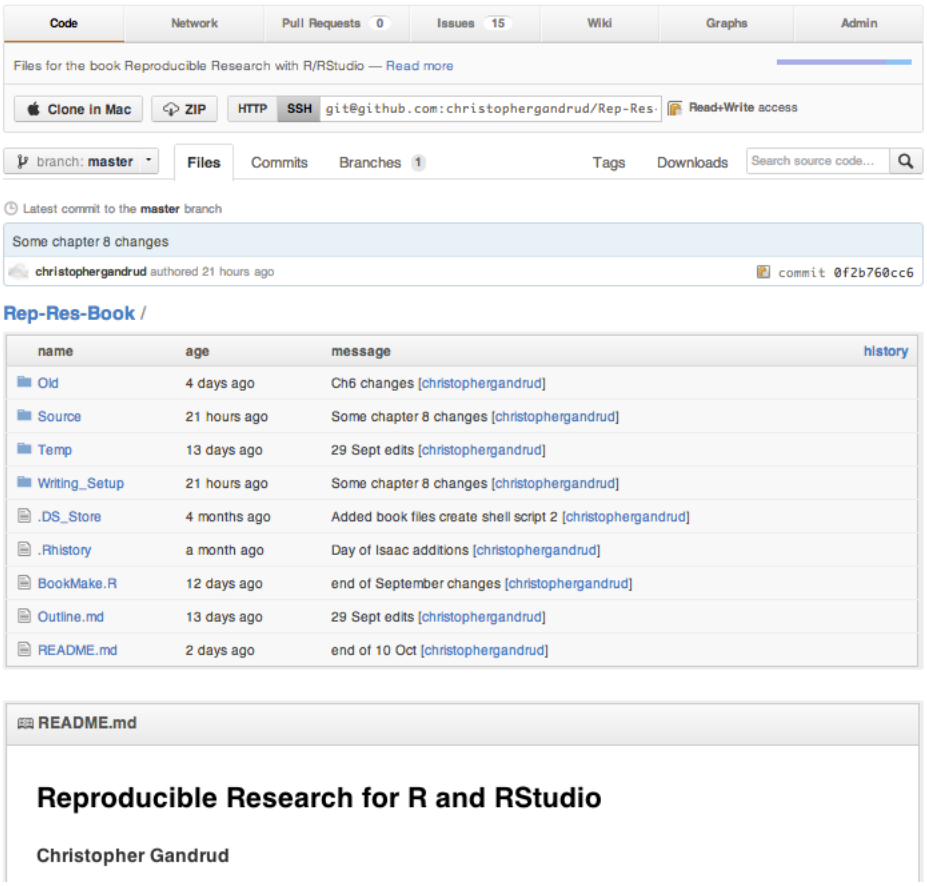
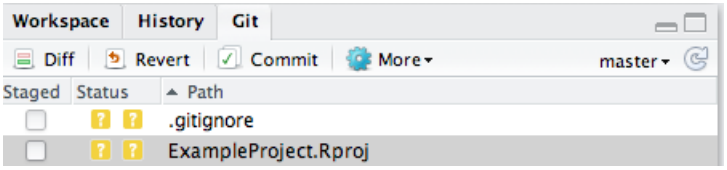
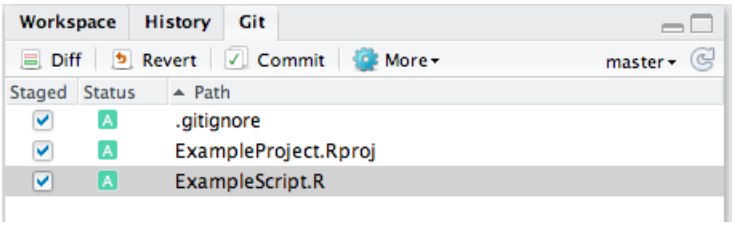


FIGURE 5.3
The RStudio Git Tab

ExampleProject Git Tab



Adding Changes to the Repository



6

Gathering Data with R

How you gather your data directly impacts how reproducible your research will be. Of course you should try your best to document every step of your data gathering process. Reproduction will be easier if your documentation–especially, variable descriptions and source code–makes it easy for you and others to understand what you have done. If all of your data gathering steps are tied together by your source code, then independent researchers (and you) can more easily regather the data. Regathering data will be easiest if running your code allows you to get 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 you can sometimes do is describe your data gathering process in detail. Nonetheless, R's automated data gathering capability for internet-based information is extensive. Learning how to take full advantage of these capabilities greatly increases reproducibility and can save you considerable time and effort over the long run.

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 makefiles to organize your whole data gathering process so that it can be completely reproduced. Then you will learn the details of how to actually load 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 you significant effort over the course of the project by making it easier to add and regather data later on.

A key part of reproducible data gathering with R, like reproducible research in general, is segmenting the process into discrete files that can all be

run by a common "make" file. In this chapter we'll learn how to create make-like files run exclusively from R as well as GNU Make,[1] which you run from a shell.[2] Learning how to create R make-like files is fairly easy. Using GNU Make does require learning yet more new syntax. However, it has one very clear advantage: it only runs a source code file that has been updated since the last time you called the makefile. This is very useful if part of your data gathering process is very computationally and time intensive.

Segmentation your data gathering and it together with some sort of make file allows you to more easily navigate research text and find errors in the source code. 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 individual data sources into R and transform them so that can be merged together 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 very raw data as input. Your source code should avoid directly changing 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 wanted to keep. Having the raw data makes it easy to go back and correct your mistakes.

The files for the examples used in this section can be downloaded from GitHub at: `http://bit.ly/YnMKBG`.

6.1.1 R Make-like files

When you create make-like files in R to organize and run your data gathing 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.[3] Lets see what an R 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 and the make-like file is called *Makefile.R*.

[1] GNU stands for "GNU's Not Unix", indicating that it is Unix-like.

[2] To standardize things, I use the terms "R make-like file" for files created and run in R and the standard "makefile" for files run by Make.

[3] We use the `source` command is used more in the Chapter 8.

```
#################
# Example R make-like file
# Christopher Gandrud
# Updated 15 January 2013
#################

# 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 frames into data frame object CleanedData
source("GatherSource/MergeData.R")
```

This code first sets the working directory. Then it runs three source code files to gather data from three different sources. These files gather the data and clean it so that it can be merged together. The cleaned data frames are available in the current workspace. Next the make file runs the *MergeData.R* file that merges the data frames and saves the output data frame as a CSV formated file.[4] The CSV file would be the main file we use for statistical analysis.

You can run the commands in this file one by one or run the make-like file through the `source` command to run it all at once.

6.1.2 GNU Make

R make-like files are a simple way to tie together a segmented data gathering process. If one or more of the source files that our example before runs is computationally intensive it is a good idea to run them only when they are updated. However, this can become tedious, especially if there are many segments. The well established GNU Make command line program[5] deals with this problem by comparing the output files' time stamps[6] to time stamps of

[4] See the full file at: `http://bit.ly/YnMKBG`.

[5] GNU Make was originally developed in 1977 by Stuart Feldman as a way to compile computer programs from a series of files, its primary use to this day. For an overview see: `http://en.wikipedia.org/wiki/Make_(software)`.

[6] A file's time stamp records the time and date when it was last changed.

the source files that created them. If a source file has a time stamp that is newer than it's output, Make will run it. If the source's time stamp is older than it's output, Make will skip it.

In Make terminology the output files are called "targets" and the files that create them are called "prerequisites". You specify a "recipe" to create the targets from the prerequisites. The general form is:

```
TARGET ... : PREREQUISITE ...
    RECIPIE
    ...
    ...
```

Note that, unlike in R, tabs are important in Make. The indicate what lines are the recipe. Make uses the recipe to ensure that targets are newer than prerequisites. If they are newer, it does nothing.

If you are using a Linux or Mac computer you already have Make installed.[7] Windows users will have Make installed if they have already installed Rtools (see page xvi).

The basic of reproducible data gathering with Make is similar to what we saw before, with a few twists and some new syntax. Let's see an example that does what we saw before: gather data from three sources, clean and merge the data and save it as in CSV format.

Example Makefile

The first thing we need to do is create a new file called *Makefile*[8] and place it in the same directory as the data gathering files. Before modifying the makefile first make sure that the `write.csv` command is moved into the *MergeData.R* file. Also, think carefully about how R packages are loaded. Imagine you have commands in a file called *Gather2.R*, for example, from libraries loaded in *Gather1.R*. If Make doesn't run *Gather1.R* because its outfile is up to date, but needs to run *Gather2.R* it won't be able to find the commands. The simplest solution is to load required packages in each source code file.

Now let's look at the actual makefile:

```
#################
# Example Makefile
```

[7] To verify this open the Terminal and type: `make --version`. This should output details about the current version of Make installed on your computer.

[8] Alternatively you can call the file *GNUmakefile* or *makefile*.

```
# Christopher Gandrud
# Updated 15 January 2013
# Influenced by Rob Hyndman (31 October 2012)
# See: http://robjhyndman.com/researchtips/makefiles/
#################

# Key variables to define
RDIR = .
MERGE_OUT = MergeData.Rout

# Create list of R source files
RSOURCE = $(wildcard $(RDIR)/*.R)

# Files to indicate when the RSOURCE file was run
OUT_FILES = $(RSOURCE:.R=.Rout)

# Default target
all: $(OUT_FILES)

# Run the RSOURCE files
$(RDIR)/%.Rout: $(RDIR)/%.R
    R CMD BATCH $<

# Remove Out Files
clean:
    rm -fv $(OUT_FILES)

# Remove MergeData.Rout
cleanMerge:
    rm -fv $(MERGE_OUT)
```

Ok, let's break down the code. The first part of the file defines variables that will be used later on. For example, in the first line of executable code (`RDIR = .`) we create a simple variable[9] called `ROUT` with a period (`.`) as its value. In Make, and Unix-like shells periods indicate the current directory. The next line allows us to specify the outfile created by running the *MergeData.R* file. This will be useful later when we create a target for removing this file to ensure that the *MergeData.R* file is always run.

The third executed line (`RSOURCE:= $(wildcard $(RDIR)/*.R)`) creates

[9]Simple string variables are often referred to as "macros" in GNU Make. A common convention in Make and Unix-like shells generally is to use all caps for variable names.

a variable containing a list of all of the names of files with the extension *.R*, i.e. our data gathering and merge source code files. This line has some new syntax, so let's work through it. The dollar signs (`$`). In make (and Unix-like shells generally) a dollar sign (`$`) followed by a variable name substitutes the variable name for the value of the variable.[10] For example, `$(RDIR)` inserts the period `.` that we defined as the value of `RDIR` previously. The parentheses are included to clearly demarcate where the variable name begins and ends.[11]

You may remember the asterisk (`*`) from the previous chapter. It is a "wildcard", a special character that allows you to select file names. The asterisk wildcard selects any file name. Using `*.R` selects any file name that ends in *.R*.

Why did we include the actual word `wildcard`? The `wildcard` function is different from the asterisk wildcard character. The function creates a list of files that match a pattern. In this case the pattern is `$(RDIR)/*.R`. The general form for writing the `wildcard` function is: `$(wildcard PATTERN)`.

The third line (`OUT_FILES = $(RSOURCE:.R=.Rout)`) creates a variable for the *.Rout* files that Make will use to tell how recently each R file was run.[12] `$(RSOURCE:.R=.Rout)` is a variable that uses the same file name as our RSOURCE files, but uses the file extension *.Rout*.

The second part of the make file tells Make what we want to create and how to create it. In the line `all: $(OUT_FILES)` we are specifying the makefile's default target. Targets are the files that you instruct Make to make. `all:` sets the default target, it is what Make tries to create when you enter the command make in the terminal with now arguments. We will see later how to instruct Make to compile different targets.

The next two executable lines (`$(RDIR)/%.Rout: $(RDIR)/%.R` and `R CMD BATCH $<`) actually runs the R source code files in the directory. The first line specifies that the *.Rout* files are the targets of the *.R* files (the prerequisites. The percent sign (`%`) is another wildcard. Unlike the asterisk, it replaces the selected file names throughout the command used to create the target.

The dollar and less than signs (`$<`) indicate the first prerequisite for the target, i.e. the *.R* files. `R CMD BATCH` is a way to call R from a Unix-like shell, run source files and output the results to other files. In Windows you will probably need to change `R` to the path for your *R.exe* file. For example: *C:\Programs Files\R\2.15.2\bin\R.exe*. The outfiles it creates have the extension *.Rout*.

The next two lines specify another target: `clean`. When you type `make clean` into your shell Make will follow the recipe: `rm -fv $(OUT_FILES)`. This removes (deletes) the *.Rout* files. The `f` argument (force) ignores files

[10] This is a kind of parameter expansion. For more information about parameter expansion see (Frazier, 2008).

[11] Braces ({}) are also sometimes used for this.

[12] The R outfile contains all of the output from the R session used while running the file. These can be a helpful place to look for errors if your make files gives you an error like `make: *** [Gather.Rout] Error 1`.

that don't exist and the `v` argument (verbose) tells you what is happening. When you delete the *.Rout* files, Make will run all of the *.R* files the next time you call it.

The last two lines help us solve a problem created by the fact that our simple makefile doesn't push changes downstream. For example, if we make a change to *Gather2.R* and run `make`, only *Gather2.R* will be rerun. The new data frame will not be added to the final merged data set. To overcome this problem the last two lines of code create a target called `cleanMerge` this removes only the *MergeData.Rout* file.

Running the MakeFile

To run the makefile for the first time simply type `make` into your shell. It will create the CSV final data file and three files with the extension *.Rout*, indicating when the segmented data gathering files were last run.[13] Remember to make sure your current working directory is the one with the files you want to run.

When you run `make` in the shell for the first time you should get the output:

```
## R CMD BATCH Gather1.R
## R CMD BATCH Gather2.R
## R CMD BATCH Gather3.R
## R CMD BATCH MergeData.R
```

If you run it a second time without changing the R source files you will get the following output:

```
## make: Nothing to be done for `all'.
```

To remove all of the *.Rout* files use set the make target to `clean`:

```
make clean

## rm -fv ./Gather1.Rout ./Gather2.Rout ./Gather3.Rout
```

[13] If you open these files you fill find the output from the R session used when the their source file was last run.

```
## ./MergeData.Rout
## ./Gather1.Rout
## ./Gather2.Rout
## ./Gather3.Rout
## ./MergeData.Rout
```

If we run the following code:

```
# Remove MergeData.Rout and make all R source files
make cleanMerge all
```

then Make will first remove the *MergeData.Rout* file (if there is one) and then run all of the R source files as need be. *MergeData.R* will always be run. This ensures that changes to the gathered data frames are updated in the final merged data set.

Makefiles and RStudio Projects

ADD

Other information about Makefiles

Note that Make relies heavily on commands and syntax of the shell program that you are using. The above example was written and tested on a Mac. It should work on other Unix-like computers without modification.

You can use Make to build almost any project from the shell, not just run R source code files. It was an integral part of early reproducible computational research (Fomel and Claerbout, 2009; Buckheit and Donoho, 1995). Rob Hyndman more recently posted a description of the makefile he uses to create a project with R and Latex.[14] The complete source of information on GNU Make is the online manual. It is available at: `http://www.gnu.org/software/make/manual/`.

[14]See his blog at: `http://robjhyndman.com/researchtips/makefiles/`. Posted 31 October 2012. This method largely replicates what we do in this book with *knitr*. Nonetheless, it has helpful information about Make that can be used in other tasks. It was in fact helpful for writing this section of the book.

6.2 Importing locally stored data sets

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. The most straightforward place to load data from is a local file, e.g. one stored on your computer. Though storing your data locally does not really encourage reproducibility, most research projects will involve loading data this way at some point. 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 foreign package
library(foreign)

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

As you can see, commands in the *foreign* package 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 into R. Ideally the documentation would be written in a text file saved in the same directory as the raw data file.

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–ones that start 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.[15]

To import the data simply include the URL as the file name in your `read.table` command. We saw in Chapter 5 how to download a CSV data file from a Dropbox *Public* folder with the shortened URL `http://bit.ly/PhjaPM`:

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

[15]If 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.

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 (Temple Lang, 2012a) 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.[16] The URL for the "raw" (plain-text) version of the data is `https://raw.github.com/christophergandrud/Disproportionality_Data/master/Disproportionality.csv`.[17] Imagine that we put the address as a character string into an object called *UrlAddress* (not shown).[18] To download it into R we could use this code:

```
# Load package
library(RCurl)

# Download Electoral disproportionality data
DataUrl <- getURL(UrlAddress)

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

# Show variables in data
names(DispropData)

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

If running `getURL(UrlAddress)` gives you an error about an `SSL certificate problem` simply add the argument `ssl.verifypeer = FALSE`. This allows you to skip certification verification and access the data.[19]

[16] For full information about the disproportionality data set please see `http://christophergandrud.github.com/Disproportionality_Data/`.

[17] To 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.

[18] See 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.

[19] For more details see the *RCurl* help page at `http://www.omegahat.org/RCurl/FAQ.html`.

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.[20] Zip files have the extension `.zip` and tar files use extensions such as `.tar` and `.gz`. In most cases[21] you can download, decompress, and create data frame objects from these files directly in R.

To do this you need to:[22]

- create a temporary file with `tempfile` to store the zipped file, which you 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,[23]
- 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 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 shortened[24] 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'.
```

[20] Tar archives are sometimes referred to as 'tar balls'.

[21] Some 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 you create for unzipped files.

[22] The 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`, posted 10 June 2010.)

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

[24] Again, I used bitly (`bitly.com`) to shorten the URL.

```
UDSData <- read.csv(gzfile(temp, "uds_summary.csv"))

# Delete the temporary file.
unlink(temp)

# Show variables in data
names(UDSData)

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

6.3.4 Data APIs & feeds

There are a growing number of packages that can gather data directly from a variety of internet sources and import them into R. Most of these packages use the sources' application programing interface (API) that allow programs interact with the website. Needless to say, this is great for reproducible research. It not only makes the data gathering process easier as you don't have to download many 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 of these packages include:

- The *openair* package (Carslaw and Ropkins, 2012), which beyond providing a number of tools for analyzing 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.
- The *quantmod* package (Ryan, 2011) allows you to access data from Google Finance,[25] Yahoo Finance[26] and the US Federal Reserve's FRED[27] economic database.
- The *treebase* package by Boettiger and Temple Lang (2012) allows you to access phylogenetic data from TreeBASE.[28]
- The *twitteR* package (Gentry, 2012) access Twitter's[29] API. This allows you to download data from twitter including tweets and trending topics.
- The *WDI* package (Arel-Bundock, 2012b) allows you to directly download data from the World Bank's Development Indicators database.[30] This

[25] `http://www.google.com/finance`
[26] `http://finance.yahoo.com/`
[27] `http://research.stlouisfed.org/fred2/`
[28] `http://treebase.org`
[29] `https://twitter.com/`
[30] `http://data.worldbank.org/data-catalog/world-development-indicators`

database includes numerous country-level economic, health, and environment variables.

The rOpenSci[31] group has and is developing a number of packages for accessing scientific data from web-based sources with R. They have a comprehensive set of packages for accessing biological data and academic journals. For a list of their packages see: `http://ropensci.org/packages/index.html`. Another fairly comprehensive and regularly updated list of APIs available as R package on Stack Exchange's Cross Validated website.[32]

API Package Example: World Bank Development Indicators

Each of these packages has its own syntax and it is impossible to go over all of them here. Nonetheless, let's look at an example of accessing World Bank data with the *WDI* to give you a sense of how these packages work. Imagine that we want to gather data on fertilizer consumption. We can use *WDI*'s `WDIsearch` command to find fretilizer consumption data available at the World Bank:

```
# Load WDI package
library(WDI)

# Search World Bank for fertilizer consumption data
WDIsearch("fertilizer consumption")

##      indicator
## [1,] "AG.CON.FERT.MT"
## [2,] "AG.CON.FERT.PT.ZS"
## [3,] "AG.CON.FERT.ZS"
##      name
## [1,] "Fertilizer consumption (metric tons)"
## [2,] "Fertilizer consumption (% of fertilizer production)"
## [3,] "Fertilizer consumption (kilograms per hectare of arable land)"
```

This shows us a selection of indicator numbers and their name.[33] Let's gather data on countries' fertilizer consumption in kilograms per hectare of arable land. The indicator number for this variable is: AG.CON.FERT.ZS. Now we can use the command `WDI` to gather the data and put it in an object called *FertConsumpData*.

[31] `http://ropensci.org/`

[32] `http://stats.stackexchange.com/questions/12670/data-apis-feeds-available-as-packages-in-r`

[33] You can also search the website itself. The indicator numbers are at the end of each indicators' URL.

```
# Gather fertilizer consumption/hectare arable land data from WDI
FertConsumpData <- WDI(indicator = "AG.CON.FERT.ZS")

# Show head of FertConsump data frame
head(FertConsumpData)
```

```
##   iso2c                country AG.CON.FERT.ZS year
## 1    1A             Arab World          67.64 2005
## 2    1A             Arab World          63.07 2004
## 3    1A             Arab World          63.14 2003
## 4    1A             Arab World          56.21 2002
## 5    S3 Caribbean small states          57.39 2005
## 6    S3 Caribbean small states          72.54 2004
```

You can see that `WDI` has downloaded data for four variables: **iso2c**,[34] **country**, **AG.CON.FERT.ZS** and **year**.

6.4 Advanced Automatic Data Gathering: web scraping

If a package does not already exist to access data from a particular website there are other ways to automatically "scrape" data from the website with R. This section briefly discusses some of the R's web scraping tools and techniques to get you headed in the right direction to do more advanced data gathering.

The general process

Simple web scraping involves downloading a file from the internet, parsing it (i.e. reading it), and extracting the data you are interested in then putting it into a data frame object. We already saw a simple example of this when we downloaded data from the a secure HTTPS website. Using the *RCurl* package we downloaded the content of a website from a URL address into R with *RCurl*'s `getURL` command. We then parsed downloaded text as CSV formatted data, extracted it and put it into a new data frame object.

This was a relatively simple process, because the webpage was very simply formatted. It basically only contained the CSV formatted text. So, the process of parsing and extracting the data was very straightforward. You may not be so lucky with other data sources. Data may be stored in an HTML formatted table within a more complicated HTML marked up webpage. The

[34] This is a the countries' or regions' International Standards Organization's two letter codes. For more details see: `http://www.iso.org/iso/country_codes.htm`.

XML package (Temple Lang, 2012c) has a number of useful commands such as `readHTMLTable` for parsing and extracting this kind of data. The *XML* package also clearly has functions for handling XML formated data.[35]. If the data is stored in JSON[36] you can read it with the *rjson* (Couture-Beil, 2012) or *RJSONIO* (Temple Lang, 2012b) packages.

There are more websites with API's than R packages designed specifically to access each one. It an API is available to access the data you want from a website the *httr* package (Wickham, 2012a) may be useful. It is a wrapper for *RCurl* intended to make accessing APIs easier.

As of the time when I was writing this book Brian Abelson's *scraply* package was not on CRAN. I would like to bring your attention to it now though because it may turn out to be a useful addition to the R web scraping tool chest. It looks like an especially promising way of handling errors that might occur while scraping a website. Errors are a persistent issue in web scraping. More information is available at the package's GitHub site: `https://github.com/abelsonlive/scraply/`.

More tools to learn about for web scraping

Beyond learning about the various R packages that are useful for R web scraping, an aspiring web scraper should probably invest time learning a number of other skills:

- HTML: Obviously you will encounter a lot of HTML markup when web scraping. Having a good understanding of the HTML markup language will be very helpful. W3 Schools (`http://www.w3schools.com/`) is a free resource for learning HTML as well as JSON, JavaScript, XML, and other languages you will likely come across while web scraping.

- Regular Expressions: Web scraping often involves using finding character patterns. Some of this is done for you by the R packages above that parse text. There are times, however, when you are looking for particular patterns, like tag IDs that are particular to a given website and change across the site based on a given pattern. You can use regular expressions to deal with these situations. R has a comprehensive, if bar bones introduction to regular expressions. To access it type `?regex` into your R console.

- Looping: Web scraping often involves applying a function to multiple things, e.g. tables or HTML tags. To do this in an efficient way you will need to use loops and apply functions generally available in *base* R. Matloff (2011) provides a comprehensive overview. The *plyr* packages (Wickham, 2012b) is also particularly useful.

[35] XML stands for "Extensible Markup Language"
[36] JSON means "JavaScript Object Notation"

7

Preparing Data for Analysis

Once we have gathered the raw data that we want to include in our statistical analyses we generally need to clean so that it can be merged it into a single data file. In this chapter we will learn how to create the data gather and merging files we saw last chapter. The chapter also includes information on recoding and transforming variables. This is important for merging data, but will be very useful information in later chapters as well. 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

In order to successfully merge two or more data frames we need to make sure that they are in the same format. Let's look at some of the important formatting issues and how to reformat your data frames so that they can be easily merged.

7.1.1 Get a handle on your data

Before doing anything to your data it is a good idea to take a look at it and see what needs to be done. Surprisingly, just taking a little time to look at your data will help you avoid many error messages and much frustration.

To get a sense of your data you could of course just type a data frame object's name into the R console. This will print the entire data frame. For data frames with more than a few variables and observations. We have already seen a number of commands that are useful for seeing parts of your data. The `names` command shows you the variable names of a data frame object. The `head` command shows the first few observations in a data frame and `tail` shows the last few.

The `summary` command is especially helpful for seeing not only basic descriptive statistics for all of the variables in a data frame, but also the variables' types. For example, let's us the *FertConsumpData* object we created in Chapter 6:

```
# Summarize FertConsumpData data frame object
summary(FertConsumpData)

##     iso2c              country           AG.CON.FERT.ZS      year
##  Length:984         Length:984         Min.   :   0   Min.   :2002
##  Class :character   Class :character   1st Qu.:  12   1st Qu.:2003
##  Mode  :character   Mode  :character   Median :  80   Median :2004
##                                        Mean   : 180   Mean   :2004
##                                        3rd Qu.: 161   3rd Qu.:2004
##                                        Max.   :8964   Max.   :2005
##                                        NA's   :251
```

We can immediately see that the variables **iso2c** are character strings. Because *summary* is able to calculate means, medians, and so on for **AG.CON.FERT.ZS** and **year** we know they are numeric. You can of course run *summary* on a particular variable by using the component selector (`$`):

```
# Summarize the methane emissions variable from FertConsumpData
summary(FertConsumpData$AG.CON.FERT.ZS)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       0      12      80     180     161    8960     251
```

We'll come back to why knowing this type of information is important for merging and data analysis later in this Chapter.

You can view a portion of a data frame object with the `View` command. This will open a new window that lets you see a selection of the data frame. If you are using RStudio, you can click on the data frame in the *Workspace* tab and you will get something that look similar. Note that neither of these viewers are interactive in that you can't use them to manipulate the data. They are only data viewers. To be see similar windows that you can interactively edit use the `fix` command in the same way that you use `view`. This can be useful for small edits, but remember that the edits are not reproducible.

7.1.2 Reshaping Data

Obviously it is usually a good if the data sets kept in data frame type objects. See Chapter 3 (page 34) for how to convert objects into data frames with the `data.frame` command. Not only do data sets (generally) need to be stored in data frame objects they also need to follow the same layout before they can be merged. Most R statistical analysis tools assume that your data is in

"long" format (as we also did in Chapter 3). This usually means that data frame columns are variables and rows are specific observations (see Table 7.1).

TABLE 7.1
Long Formatted Data Example

Subject	Variable1
Subject1	
Subject2	
Subject3	
...	

In this chapter we will mostly use examples of time-series cross-sectional data (TSCS) that we want to have in long-format. Long formatted TSCS has is simply a data frame where rows identify observations of a particular subject at three points in time (see Table 7.2)

TABLE 7.2
Long Formatted Time-series Cross-sectional Data Example

Subject	Time	Variable1
Subject1	1	
Subject1	2	
Subject1	3	
Subject2	1	
Subject2	2	
Subject2	3	
...		

In this chapter our TSCS data is specifically going to be countries that are observed in multiple years.
If one of your data sets is not in this format then you will need to reshape it. Some data sets are in "wide" format; where one of the columns in long formatted data is widened to cover multiple columns. This can be confusing

without an example. Table 7.3 shows how Table 7.2 looks when we widen the time variable.

TABLE 7.3
Wide Formatted Data Example

Subject	Time1	Time2	Time3
Subject1			
Subject2			
...			

Reshaping data is often the cause of much confusion and frustration. Though probably never easy, there are a number of useful R functions for changing data from wide format to long and vice versa. These include the matrix transpose command (**t**)[1] and the **reshape** command, both in loaded in R by default. Another very helpful package is *reshape2* (Wickham, 2012c). This provides more general tools for reshaping data and is worth investing some time in learning well. In this section we will cover some of *reshape2*'s basic commands and use them to reshape TSCS data frame from wide to long format. We will also encounter this package in more detail in Chapter 10 when we want to transform data so that it can be graphed.

Let's imagine that the fertilizer consumption data we previously downloaded from the World Bank is in wide rather than long format and is in a data frame objected called *WideFert*. It looks like this:[2]

```
head(WideFert)
```

```
##    iso2c        country   2002   2003    2004    2005
## 8     AF    Afghanistan  3.403  3.275   4.536   4.240
## 10    AL        Albania 97.185 98.933 100.599 111.597
## 58    DZ        Algeria  9.642  6.002  25.095   7.430
## 14    AS American Samoa     NA     NA      NA      NA
## 6     AD        Andorra     NA     NA      NA      NA
## 12    AO         Angola  1.659  1.789   4.502   2.261
```

[1] See this example by Rob Kabacoff: `http://www.statmethods.net/management/reshape.html`. Note also that because the matrix transpose function is denoted with simply as `t`, you should not give any object the name *t*.

[2] Please see the Appendix (page 125) for the code I used to reshape the data.

Let's use *reshape2*'s `melt` command to reshape this data from wide to long format. The term "melt" is intended to evoke an image of the data melting down from a wide to long format.[3] In our *WideFert* data we don't want the **iso2c** and **country** variables to be melted. These variables identify the data set's subjects. we can tell `melt` that they are id variables with the `id.vars` argument. The remaining columns (i.e. **2002**, **2003**, **2004** and **2005**) will be melted into two new variables: **variable**, and **value**. The former will contain the years and the later will contain the fertilizer consumption data. Here is the full code:

```
# Melt WideFert
MoltenFert <- melt(data = WideFert,
                   id.vars = c("iso2c", "country"))

# Show MoltenFert
head(MoltenFert)
```

```
##   iso2c        country variable  value
## 1    AF    Afghanistan     2002  3.403
## 2    AL        Albania     2002 97.185
## 3    DZ        Algeria     2002  9.642
## 4    AS American Samoa     2002     NA
## 5    AD        Andorra     2002     NA
## 6    AO         Angola     2002  1.659
```

Note that objects that are created by `melt` are often referred to as "molten" data in the *reshape2* documentation. That is why I've given our new data frame the name *MoltenFert*.

7.1.3 Renaming variables

Frequently, in the data clean up process we want to change the names of our variables. This will make our data easier to understand and may even be necessary to properly combine data sets (see below). In the previous example, for instance, our *MoltenFert* data frame has two variables–**variable** and **value**–that would be easier to understand if they were renamed **year** and **FertilizerConsumption**. Rename data frame variables is straight forward with the `rename` command in the *reshape* package (Wickham, 2011).[4]

[3] The opposite `cast` command (`dcast` in the case of data frames) is supposed to evoke an image of casting out the data from long to wide format. See page 125 for am example using the `dcast` command.

[4] *reshape* package precedes *reshape2*, which was created to improve the performance of the `melt` and `cast` commands

To rename both **variable** and **value** with the `rename` command type:

```
# Load reshape package
library(reshape)

# Rename variable = year, value = FertilizerConsumption
MoltenFert <- rename(x = MoltenFert,
                     replace = c(variable = "year",
                                 value = "FertilizerConsumption"))

# Show MoltenFert
head(MoltenFert)

##   iso2c        country year FertilizerConsumption
## 1    AF    Afghanistan 2002                 3.403
## 2    AL        Albania 2002                97.185
## 3    DZ        Algeria 2002                 9.642
## 4    AS American Samoa 2002                    NA
## 5    AD        Andorra 2002                    NA
## 6    AO         Angola 2002                 1.659
```

7.1.4 Ordering data

You may have noticed that as a result of melting *WideFert* the data is now ordered by year then country name. Typically TSCS data is sorted by country then year, or more generally: subject-year. Though not required for merging in R[5] some statistical analyses assume that the data is ordered in a specific way. Well ordered data is also easier for people to understand.

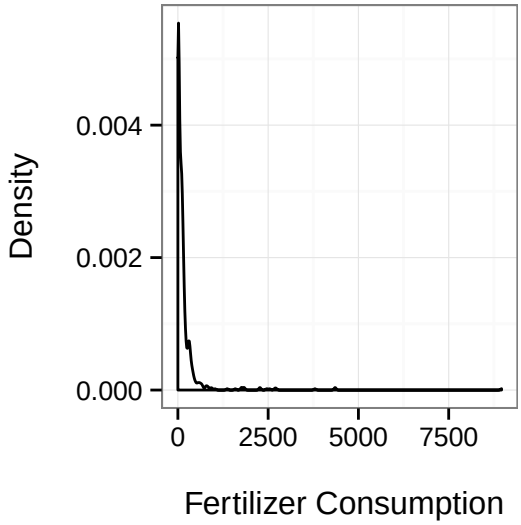
We can order observations in our data set using the `order` command. For example, to order *MoltenFert* by country-year we type:

```
# Order MoltenFert by country-year
MoltenFert <- MoltenFert[order(MoltenFert$country,
                               MoltenFert$year), ]

# Show MoltenFert
head(MoltenFert)
```

[5] Unlike in other statistical programs.

FIGURE 7.1
Density Plot of Fertilizer Consumption (kilograms per hectare of arable land)



Source: World Bank (2012)

```
##     iso2c     country year FertilizerConsumption
## 1      AF Afghanistan 2002                 3.403
## 247    AF Afghanistan 2003                 3.275
## 493    AF Afghanistan 2004                 4.536
## 739    AF Afghanistan 2005                 4.240
## 2      AL     Albania 2002                97.185
## 248    AL     Albania 2003                98.933
```

7.1.5 Subsetting data

Sometimes you may want to use only a subset of a data frame. For example, the density plot in Figure 7.1 shows us that the MoltenFert data has a few very extreme values. We can use the `subset` command to examine the outliers, for example countries that have fertilizer consumption greater than 1000 kilograms per hectare.

```
# Create outlier data frame
FertOutliers <- subset(x = MoltenFert,
                       FertilizerConsumption > 1000)

# Show FertOutliers
FertOutliers
```

```
##     iso2c     country year FertilizerConsumption
## 16     BH     Bahrain 2002                  8964
## 754    BH     Bahrain 2005                  4360
## 786    CR  Costa Rica 2005                  1030
## 98     IS     Iceland 2002                  2686
## 344    IS     Iceland 2003                  2265
## 590    IS     Iceland 2004                  2542
## 836    IS     Iceland 2005                  2461
## 109    JO      Jordan 2002                  1590
## 116    KW      Kuwait 2002                  1763
## 854    KW      Kuwait 2005                  4349
## 160    NZ New Zealand 2002                  1836
## 406    NZ New Zealand 2003                  2279
## 652    NZ New Zealand 2004                  1761
## 898    NZ New Zealand 2005                  2719
## 907    OM        Oman 2005                  1366
## 674    QA       Qatar 2004                  3796
## 194    SG   Singapore 2002                  1830
```

If we want to drop these outliers from our data set we can use `subset` again.

```
MoltenFertSub <- subset(x = MoltenFert,
                        FertilizerConsumption <= 1000)
```

In this data example, non-country units like "Arab World" are included. We might want to drop these units with the subset command as well. For example:

```
# Drop Arab World type from MoltenFertSub
MoltenFertSub <- subset(x = MoltenFertSub,
                        country != "Arab World")
```

We can also use `subset` to remove observations with missing values (`NA`) for **FertilizerConsumption**.

```
# Remove observations of FertilizerConsumption
# with missing values
MoltenFertSub <- subset(x = MoltenFertSub,
                        !is.na(FertilizerConsumption))

# Summarize FertilizerConsumption
summary(MoltenFertSub$FertilizerConsumption)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0    11.6    78.3   118.0   151.0   939.0
```

Let's step back one second. I've introduced a number of new logical operators and a new command in the four subsetting examples. The first example included a very simple one, the greater than sign (`>`). The second example included the less than or equal to operator: `<=`. The third example included the not equal operator: `!=`. In R exclamation points (`!`) generally denote 'not'. We used this again in the final example in combination with the `is.na` command. This command indicates if an element is missing, so `!is.na` means "not missing". For the full list of R's logical operators see Table 7.4. You can use these operators and command when subsetting data and throughout R.

TABLE 7.4
R's Logical Operators

Operator	Meaning
`<`	less than
`>`	greater than
`==`	equal to
`<=`	less than or equal to
`>=`	greater than or equal to
`!=`	not equal to
`a | b`	a or b
`a & b`	a & b
`isTRUE(a)`	determine if a TRUE
`is.na`	missing
`!is.na`	not missing
`duplicated`	duplicated observation
`!duplicated`	not a duplicate observation

7.1.6 Recoding variables strings/ numeric variables

You may want to recode your variables. In particular you when you merge data sets together you need to have **identical** identification variables that R can use to match your data on. If in one data set observations for the Republic of Korea are referred to as "Korea, Rep." and in another it is labeled "South Korea" they will not be merged. We need to recode values in the variables that we want to match our data sets on. For example, in *MoltenFertSub* Korea is labeled "Korea, Rep.". To recode it to "South Korea" we type:

```
# Recode country == "Korea, Rep."" to "South Korea"
MoltenFertSub$country[MoltenFertSub$country ==
                        "Korea, Rep."] <- "South Korea"
```

This code assigns "South Korea" to all values of the country variable that equal "Korea, Rep.".[6] You can a similar technique to recode numeric variables as well. The only difference is that you need to omit the quotation marks.

We will look at how to code factor variables in the next subsection.

7.1.7 Creating new variables from old

As part of your data clean up process (or later during statistical analysis) you may want to create new variables based on existing variables. For example, we could create a new variable that is the natural logarithm of **FertilizerConsumption**. To do this we run the variable through the `log` command and assign a new variable that we'll call **logFertConsumption**.

```
MoltenFertSub$logFertConsumption <- log(
                                    MoltenFertSub$FertilizerConsumption
                                    )
```

We can use a similar procedure to create new variables from R's many other mathematical commands and arithmetic operations.[7]

Factor variables from numeric

Creating factor variables

We can also create factor variables from numeric or string variables. For example, we may want to turn the continuous numeric **FertilizerConsumption** variable into an ordered categorical (i.e. factor) variable. Imagine that we want to create a factor variable called **FertConsGroup** with four levels called 'low', 'medium low', 'medium high', 'high'. To do this lets first create a new numeric variable based on the values listed in Table 7.5. To do a procedure that is similar to the variable recoding we did earlier:[8]

[6] The *countrycode* package (Arel-Bundock, 2012a) is very helpful for creating standardized country identification variables.

[7] E.g `+, -, *, /, ^` for addition, subtraction, multiplication, division and exponentiation, respectively.

[8] In this code I attached the data frame *MoltenFertSub* so that it is easier to read.

TABLE 7.5
Example Factor Levels

Number	Meaning	Value of **FertilizerConsumption**
1	low	< 15
2	medium low	≥ 15 & < 80
3	medium high	≥ 80 & < 150
4	high	≥ 150

```
#### Create numeric factor levels variable ####
# Attach MoltenFertSub data frame
attach(MoltenFertSub)

# Created new FertConsGroup variable based on
# FertilizerConsumption
MoltenFertSub$FertConsGroup[FertilizerConsumption
                            < 15] <- 1
MoltenFertSub$FertConsGroup[FertilizerConsumption
                            >= 15 &
                            FertilizerConsumption < 80] <- 2
MoltenFertSub$FertConsGroup[FertilizerConsumption
                            >= 80 &
                            FertilizerConsumption < 150] <- 3
MoltenFertSub$FertConsGroup[FertilizerConsumption
                            >= 150] <- 4

# Summarize FertConsGroup
summary(MoltenFertSub$FertConsGroup)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    1.00    2.00    2.47    4.00    4.00


# Detach data frame
detach(MoltenFertSub)
```

You'll notice that we don't have a factor variable yet, our new variable is nu-

meric. We can use the `factor` to convert *FertConsGroup* into a factor variable with the labels we want.

```
# Create vector of factor level labels
FCLabels <- c("low", "medium low", "medium high", "high")

# Convert FertConsGroup to a factor
MoltenFertSub$FertConsGroup <- factor(MoltenFertSub$FertConsGroup,
                                      labels = FCLabels)

# summarize FertConsGroup
summary(MoltenFertSub$FertConsGroup)

##         low  medium low medium high        high
##         195         168         167         182
```

We first created a character vector with the factor level labels and then applied using `factor`'s `labels` argument. Using `summary` with a factor variable shows gives us its level labels as well as the number of observations per level.

7.1.8 Changing variables types

Sometimes a variable will have the wrong type. For example, a numeric variable may be incorrectly made a character string when a data set is imported from Excel. You can change variables' types with a number of commands. We already saw how to convert a numeric variable to a factor variable with the `factor` command. Unsurprisingly, to convert a variable to a character use `character` and `numeric` to convert it to numeric. We can place `as.` before these commands (e.g. `as.factor`) to as an abbreviated way of coercing a change in type.

Though the commands have straightforward names, a word of caution is necessary. Always to to understand why a variable is not of the type you would expect often times variables have unexpected types because they are coded (or miscoded) in a way that you didn't anticipate. Changing the variables' types, especially when using `as. ...`, can introduce new errors. Make sure that the conversion made the changes you expected.

7.2 Merging data sets

In the previous section we learned crucial skills for cleaning up data sets. When your data sets are (a) in the same format and (b) have identically matching ID variables you can merge your data sets together. In this section we'll look at two different ways to merge data sets: binding and the `merge` command. We'll also look at ways to address a common issue when merging data: duplicated observations.

7.2.1 Binding

As we saw in Chapter 3, if your data sets are in the same order–rows in all of the data sets represent the same observation of the same subject–then you can simply use the `cbind` to bind columns from the data sets together. This situation is often very rare when merging real-world data. If your data sets are not in exactly the same order you will create a data set with nonsensical rows that combine data from multiple observations. Therefore, you should avoid using `cbind` for merging most real-world data.

If you have data sets with the exact same columns and variable types and you just want to attach one under the other you can use the `rbind` command. It binds the rows in one object to the rows in another.[9] It has the same syntax as `cbind` (see page 33). Again, you should be cautious when using this command, though it is more difficult to accidentally create a nonsensical data set. R will give you an error if it cannot match your objects' columns.

7.2.2 The merge command

Generally, the safest and most effective way to merge two data sets together is with the `merge` command. Imagine that we want to merge our MoltenFertSub data frame with two data frames we created in Chapter 6: *FinRegulatorData* and *DispropData*. The simplest way to do this is to use the merge command twice, i.e.:

```
# Merge FinRegulatorData and DispropData
MergedData1 <- merge(x = FinRegulatorData,
                     y = DispropData,
                     by = "iso2c",
                     all = TRUE)
```

[9] Some statistical programs refer to this type of action as "appending" one data set to another.

```
# Merge combined data set with and MoltenFertSub
MergedData1 <- merge(x = MergedData1,
                     y = MoltenFertSub,
                     by = "iso2c",
                     all = TRUE)

# Show MergedData1 variables
names(MergedData1)

##  [1] "iso2c"                 "idn"
##  [3] "country.x"             "year.x"
##  [5] "reg_4state"            "country.y"
##  [7] "year.y"                "disproportionality"
##  [9] "country"               "year"
## [11] "FertilizerConsumption" "logFertConsumption"
## [13] "FertConsGroup"
```

Let's go through this code. The `x` and `y` arguments simply specify which data frames we want to merge. The `by` argument specifies what variable in the two data sets identifies the observations so that we can match them. In this example we are merging by countries' ISO country two letter codes.[10] We set the argument `all = TRUE` so that we keep all of the observations from both of the data frames. If the argument is set to `FALSE` then only observations that are common to both data frames. The others will not be included.

You might have noticed that this isn't actually the merge that we want to accomplish with these data frames. Remember that observations are not simply identified in this time-series cross-section data by one country or country code variable. Instead they are identified by both country and year variables. To merge data frames based on the overlap of two variables (e.g. match Afghanistan-2004 in one data frame with Afghanistan-2004 in the other) we need to add the `union` command to `merge`'s `by` argument. Here is a full example:[11]

```
# Merge FinRegulatorData and DispropData
MergedData2 <- merge(
```

[10] Please see this Chapter's Appendix (page 126) for details on how I created ISO country two letter code variables in the *DispropData* and *FinRegulatorData* data frames.

[11] You can download a modified version of this example as part of the Make file exercise from Chapter 6: `http://bit.ly/YnMKBG`.

```
                    FinRegulatorData, DispropData,
                    union("iso2c", "year"),
                    all = TRUE)

# Merge combined data frame with MoltenFert
MergedData2 <- merge(
                    MergedData2, MoltenFertSub,
                    union("iso2c", "year"),
                    all = TRUE)

# Show MergedData2 variable names
names(MergedData2)

##  [1] "iso2c"                "year"
##  [3] "idn"                  "country.x"
##  [5] "reg_4state"           "country.y"
##  [7] "disproportionality"   "country"
##  [9] "FertilizerConsumption" "logFertConsumption"
## [11] "FertConsGroup"
```

After merging data frames it is always a good idea to look at the result and make sure that the result of the merge is what you expected. Some post merging clean up may be required to get the data frame ready for statistical analysis.

7.2.3 Duplicate values

One thing to look out for after (and before) merging are duplicate observations. You can use the `duplicated` command to check for duplicates. Use the command in conjunction with subscripts to remove duplicate observations. For example, let's create a new object called *DataDuplicates* from the iso2c-years that are duplicated in *MergedData2*. Note remember that **iso2c** and **year** are in the first and second columns of the data frame.

```
# Created a data frame of unique country-years
DataDuplicates <- MergedData2[duplicated(
                                MergedData2[, 1:2]), ]

# Show the number of rows in DataDuplicates
nrow(DataDuplicates)
```

```
## [1] 7
```

In this data frame there are 7 duplicated iso2c-year observations. This is indicated by the fact that using the `nrow` command[12] we find that the *MergedData2* data frame has 7 rows, i.e. 7 observations.

To create a data set of without duplicated observations (if there are duplicates) we just add an exclamation point (`!`) before `duplicated`–i.e. not duplicated–in the above code.[13]

```
# Created a data frame of duplicated country-years
DataNotDuplicates <- MergedData2[!duplicated(
                                  MergedData2[, 1:2]), ]
```

Note that if you do have duplicated values in your data set and you run a similar procedure on it, it will drop duplicated values that have a lower order in the data frame. To keep the lowest ordered value and drop duplicates higher in the data set use `duplicated`'s `fromLast` argument. Set it to `TRUE`.

A word of caution is in order. Lock over your data set and the source code that created the data set to try to understand why duplicates occurred. There may be a more fundamental problem in the way you are handling your data that resulted in the duplicated observations.

7.2.4 Duplicate columns

Another common post-merge clean up issue are duplicate variables. These are variables from the two data frames with the same name that were not included in `merge`'s `by` argument. For example, in our previous merged data examples there are three country name variables: **country.x**, **country.y** and **country** to signify which data frame they are from.[14]

You should of course decide what to do with these variables on a case-by-case basis. But if you decide to drop one of the variables and rename the other, you can use subscripts (as we saw in Chapter 3). The *gdata* package (Warnes et al., 2012) has a useful function called `remove.vars` that can also remove variables from data frames. For example, imagine that we want to

[12] `nrow` returns the number of rows an object has.

[13] `!duplicated` is equivalent to the `unique` command.

[14] The former two were created in the first merge between *FinRegulatorData* and *DispropData*. When the second merge was completed there were no variables named **country** in the MergeData2 data frame, so **country** did not need to be renamed in the new merged data set.

keep **country.x** and drop the other variables[15] You can use these procedures to remove other unwanted variables as well, let's also remove the extraneous **idn** variable:

```
# Load gdata and reshape packages
library(gdata)
library(reshape)

# Remove country.y, country and idn
FinalCleanedData <- remove.vars(data = DataNotDuplicates,
                                names = c("country.y",
                                          "country",
                                          "idn"))

## Removing variable 'country.y'
## Removing variable 'country'
## Removing variable 'idn'


# Rename country.x = country
FinalCleanedData <- rename(x = FinalCleanedData,
                           replace = c(country.x =
                                       "country"))
```

```
# Show FinalCleanedData variables
names(FinalCleanedData)

## [1] "iso2c"                "year"                 "country"
## [4] "reg_4state"           "disproportionality"   "FertilizerConsumption"
## [7] "logFertConsumption"   "FertConsGroup"
```

Note if you are merging many data sets it can sometimes be good to clean up duplicate columns between each `merge` call.

Appendix

R code for turning *FertConsumData* into year-wide format:

[15] This version of the country variable is the most complete.

```
# Load WDI and reshape 2 package
library(WDI)
library(reshape2)

# Gather agricultural methane emissions data from WDI
FertConsumpData <- WDI(indicator = "AG.CON.FERT.ZS")

# Melt data
## Note: data must be melted before it can be cast.
MoltenFert <- melt(data = FertConsumpData,
                   id.vars = c("iso2c", "country", "year"),
                   measure.vars= "AG.CON.FERT.ZS")

# Cast MoltenFert to year wide format
WideFert <- dcast(data = MoltenFert,
                  formula = iso2c + country ~ year,
                  value_var = "AG.CON.FERT.ZS")

# Order WideFert by country
WideFert <- WideFert[order(WideFert$country), ]
```

R code for creating iso2c country codes with the *countrycode* package:

```
# Load countrycode package
library(countrycode)

# FinRegulatorData
FinRegulatorData$iso2c <- countrycode(FinRegulatorData$country,
                                      origin = "country.name",
                                      destination = "iso2c")

# DispropData
DispropData$iso2c <- countrycode(DispropData$country,
                                 origin = "country.name",
                                 destination = "iso2c")
```

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 analyses. In this book we don't look at how to do statistical analysis in R (a subject that would and does take many books). Instead we focus on how to make your analyses really reproducible. To do this you dynamically connect your data gathering and analysis source code to your presentation documents. When you dynamically connect your data gathering makefiles and analysis source code file to your markup document you will be able to completely rerun your data gathering and analysis and present the results whenever you compile the presentation documents. Doing this makes it very clear how you found the results that you are advertising. It also automatically keeps the presentation of your results–including tables and figures–up-to-date with any changes you make to your data and analyses source code files.

You can dynamically tie your data gathering, statistical analyses and presentation documents together with *knitr*. In Chapter 3 you learned basic *knitr* syntax. In this chapter we will begin to learn *knitr* syntax in 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 separate blocks and inline with the text. We will also learn how to dynamically include code from languages other than R. We will finally examine how to use *knitr* when with segmented 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 data gathering and 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.

sessioninfohow **Reminder:** Before discussing the details of how to incorporate you analysis into your source code, it's important to reiterate something we discussed in Chapter 2. The syntax and capabilities of R packages and R itself can change with new versions. Also, as we have seen for file path names, syntax can change depending on what operating system you are using. So it is important to make your R session info available (see page 21 for details)

to make your research reproducible and future proof. If someone reproducing your research has this information they can likely download and use the exact version of the software that you used. For example, CRAN maintains an archive of previous R package versions.[1] Previous versions of R itself can also be downloaded through CRAN.[2]

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 simplest to type the code directly into chunks written in your *knitr* markup document. In this section we will learn how set *knitr* options to handle these code chunks. For a list including the chunk options covered here see Table 3.1.

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 at the point where they are written in the markup version. Figures are inserted as well. Let's look at the main options for determining how code chunks are handled by *knitr*.

`eval`

The `eval` option determines whether or not the code in a chunk will be run. Set the `eval` option to `FALSE` if you would like to include code chunks without actually running them. By default it is set to `TRUE`, i.e. the code is run.

`echo`

If you would like hide a chunks code you from the presentation document you can set `echo=FALSE`. Note that if you also have `eval=TRUE` then the chunk will still be evaluated and the output will be included in your presentation document. Clearly if `echo=TRUE` (which it is by default) then source code will be included in the presentation document.

[1] See: `http://cran.r-project.org/src/contrib/Archive/`.
[2] See: `http://cran.r-project.org/src/base/`.

`warning`, `message`, `error`

If you don't want to include the warnings, messages, and error messages that R outputs in the text of your presentation documents just set the `warning`, `message`, and `error` options to `FALSE`. They are set to `TRUE` by default.

`include`

Use `include=FALSE` if you don't want to include anything in the text of your presentation document, but you still want to evaluate a code chunk. It is `TRUE` by default.

`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`. When you do this the code chunk is run only if the code changes. This is very handy if you have a code chunk that is computationally intensive to run. The `cache` option is set to `FALSE` by default.

Unfortunately, the `cache` option has some limitations. For example, other code chunks can't access objects that have been cached. Packages that are loaded in cached chunks cannot be accessed by other chunks.

8.1.2 Showing code & results inline

Sometimes you may want to have R code or output show up inline with the rest of your presentation document's text. For example, you may want to include a small chunk of stylized code in your text when you discuss how you 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. We'll cover both in turn.

8.1.2.1 LaTeX

Inline static code

There are a number of ways to include a code snippet inline with your text in LaTeX. You can simply use the LaTeX command `\texttt` to have text show up in the `typewriter` font commonly used LaTeX to indicate that some text is code (I use typewriter font for this purpose in this book, as you have probably noticed). For example, using `\texttt{2 + 2}` will give you `2 + 2` in your text. Note that in LaTeX curly brackets (`{}`) work exactly like parentheses in R, i.e. the enclose a command's arguments.

However, the `\texttt` command isn't always ideal, because your LaTeX compiler will still try to run the code inside of the command as if it was LaTeX markup. This can be problematic if you include characters like the

backslash `\` or curly brackets `{}` which have special meanings for LaTeX. The hard way to solve this problem is to use escape characters (see Chapter 4). The backslash is an escape character in LaTeX. Probably the better option is to use the `\verb` command. It is equivalent to the `eval=FALSE` option for full *knitr* code chunks.

To use the `\verb` command pick some character you will not use in the inline code. For example, you could use the vertical bar (`|`). This will be the `\verb` delimiter. Imagine that we want to actually included '`\texttt`' in the text. We would type:

```
\verb|\texttt|
```

The LaTeX compiler will ignore almost anything from the first vertical bar up until the second bar following `\verb`. All of the text in between the delimiter characters is put in typewriter font.[3]

Inline dynamic code

If you want to dynamically show the results of some R code in your *knitr* LaTeX produced text you can use the `\Sexpr`. This is a pseudo LaTeX command; it looks like LaTeX, but is actually *knitr*.[4] Its structure is more like a LaTeX command's structure than *knitr*'s in that you enclose your R code in curly brackets ({}) rather than the `<<>>= . . . @` syntax you use for block code chunks.

For example, imagine that you wanted to include the mean of a vector of river lengths–591–in the text of your document. The *rivers* numeric vector, loaded by default in R, has the lengths of 141 major rivers recorded in miles. You can simply use the `mean` command to find the mean and the `round` command to round the result 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:

[3]For more details see the LaTeX Wikibooks page: `http://en.wikibooks.org/wiki/LaTeX/Paragraph_Formatting#Verbatim_Text` (accessed 24 November 2012).

[4]The command directly descends from Sweave.

```
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:[5]

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 in code chunks

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 use the `engine` code chunk option

[5]The exact look of the text depends on the CSS style file you are using. The example here was created with RStudio's default style file.

to tell *knitr* which language you are using. For example, to dynamically include a simple line of Ruby code in an R Markdown document type:

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

In the final HTML file you will get:[6]

```
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 than what it supports for R. For example, there is no colored syntax highlighting.

8.2 Dynamically including modular 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 slide show presentation for example. It is nice to not have to copy and paste the same code into multiple places. Instead it is easier to have multiple documents link to the same source code file. When you make changes to this source code file, the changes will automatically be made across all of your presen-

TABLE 8.1
Knitr `engine` Values

Value	Programming Language
`awk`	Awk
`bash`	Bash
`gawk`	Gawk
`haskell`	Haskell
`highlight`	Highlight
`python`	Python
`R`	R (default)
`ruby`	Ruby
`sh`	Bash

[6] Again, this was created using RStudio's default CSS style file.

tation 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 modular files that are explicitly tied together in the markup file with *knitr* it is easy for them to find the specific bits of code that they are interested in.

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 your code in the chunk, you save it to its own file and use the `source` command to access it.[7] For example, in an R Markdown file we could run the R code in a file called *MainAnalysis.R* from our *ExampleProject* like this:

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

Notice that we set the `include=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 output will not show up in the presentation document's text.

Sourcing a makefile in a code chunk

In Chapter 6 we created a GNU Makefile to organize our data gathering. You can run makefiles every time you compile your presentation document. This can keep your data, analyses, figures and tables up-to-date. One way to do this is to run the GNU makefile in an R code chunk with the `system` command (see page 67). Perhaps a better way to run makefiles from *knitr* presentation documents is to include the commands in a code chunk using the Bash engine. For example, a Sweave-style code chunk for running the makefiles in our example project would look like this:

```
<<engine='sh', include=FALSE>>=
# Change the working directory to /Data/GatherSource
```

[7] We used the `source` command in Chapter 6 in our make-like data gathering file.

```
cd /ExampleProject/Data/GatherSource/

# Run makefile
make cleanMerge all

# Change to working directory to /ExampleProject/Analysis/
cd /ExampleProject/Analysis/
@
```

Please see page 100 for details on the `make` command arguments used here.

You can of course also use R's `source` command to run an R make-like data gathering file. Unlike GNU Make, this will rerun all of the data gathering files, even if they have not been updated. This may become very time consuming depending on the size of your data sets and who they are manipulated.

One final note on including makefiles in your *knitr* presentation document source code: it is important to place the code chunk with the makefile before code chunks containing statistical analyses that depend on the data file it creates. Placing the makefile first will keep the others up-to-date as well.

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 for this issues is to host the replication code in your Dropbox public folder. You can find the file's public URL in the same way that you did in Chapter 5. Then use the `source` command the same way as we did before with the `read.table` command.[8]

Let's look at a quick example of sourcing an R function that has been made available on at a non-secure URL. Paul Johnson created a function called *outreg* for creating LaTeX tables from R objects. He has made the fucntion available his website at: `http://pj.freefaculty.org/R/WorkingExamples/outreg-worked.R`. You can directly load this function into your workspace with `source`.

[8]You can also make the replication code accessible for download and either instruct others to change the working directory to the replication file or have them change the directory information as necessary. You will need to do this with GNU makefiles like those included included with this book.

```
# Load Paul Johnson' outreg function
source("http://pj.freefaculty.org/R/WorkingExamples/outreg-worked.R")
```

Now you can use `outreg` like any other function you have loaded.

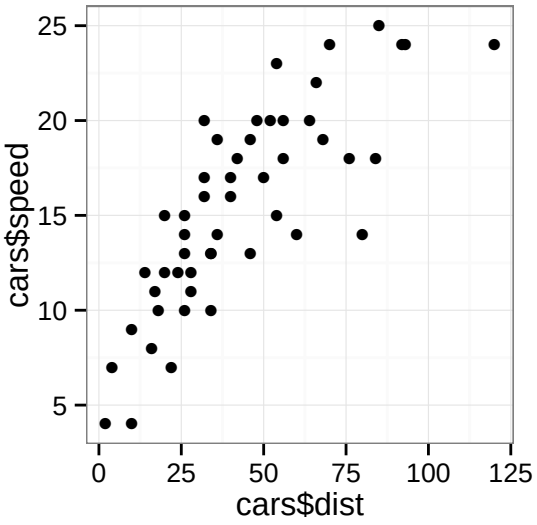
8.2.3 Source from a secure URL (`https`)

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 (Wickham and Chang, 2012a). For GitHub based source code we find the file's URL the same way we did in Chapter 6 (page 103). 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`.[9] To run this code and create the scatterplot using `source_url` you simply type:

```
# Load devtools package
library(devtools)

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

[9]The original URL is at `https://raw.github.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.



You can also use the *devtools* command `source_gitst` in a similar way to source GitHub Gists. Gists are a handy way to share code over the internet. For more details see: `https://gist.github.com/`.

9

Showing Results with Tables

Graphs and other visual methods, discussed in the next chapter, can often be a more effective way to present descriptive and inferential statistics than tables.[1] Nonetheless, tables of parameter estimates, descriptive statistics and so on can sometimes be an important tool for describing your data and presenting research findings. Learning how to dynamically connect statistical results with tables in your presentation documents aids reproducibility and can ultimately save you a lot of time.

Manually typing results into tables by hand is tedious, not very reproducible, and can introduce errors. It's especially tedious to retype tables to reflect changes you made to your data and models. Fortunately, you don't actually need to create tables by hand. There are many ways to have R do the work for you.

The goal of this chapter is to learn how to dynamically create tables for you presentation documents written in LaTeX and Markdown. There are a number of ways to turn R objects into tables that can be dynamically included in LaTeX or Markdown/HTML markup. In this chapter we mostly focus on the *xtable* (Dahl, 2012) and *apsrtable* packages (Malecki, 2012). *xtable* can create tables for both of LaTeX and Markdown/HTML. *apsrtable* usually produces publication quality tables more easily than *xtable*. Unfortunately it only works with LaTeX and is less flexible with objects of classes it does not support.[2] As we will learn at the end of the chapter, `knitr` allows us to dynamically incorporate tables from both packages into our documents.

Warning: Automating table creation removes the possibility of adding errors to your analyses by incorrectly copying output, a big potential problem in hand-created tables. However, it is not error free. You could easily create inaccurate tables through coding errors. So, as always, it is important to 'eyeball' the output. Does it make sense? If you select a couple values in the R output do they match what is in the presentation document's table? If not, you need to go back to the code and see where things have gone wrong. With that caveat, let's start making tables.

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

[2] These are not the only means available in R for creating presentation document tables from R objects. Others include the *tables* (Murdoch, 2012), *memisc* (Elff, 2012) and *estout* (?) packages as well as Paul Johnson's `outreg` function (see: `http://pj.freefaculty.org/R/WorkingExamples/outreg-worked.R`).

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. If you are not familiar with basic LaTeX or Markdown syntax you might want to skip ahead to chapters 11 and 13, respectively, before coming back to learn about making tables in these languages.

9.1.1 Tables in LaTeX

Tables in LaTeX are usually embedded in two environments: the `table` and `tabular` environments. Before discussing these particular environments, what is a LaTeX environment in general?

A LaTeX environment is a part of the markup where special commands are executed. A simple environment is the `center` environment.[3] Every thing typed in a center environment is, unsurprisingly, centered. If we typed:

```
\begin{center}
    This is a center environment.
\end{center}
```

We would create the following text in the PDF output:

This is a center environment.

LaTeX environments all follow the same general syntax:

```
\begin{ENVIRONMENT_NAME}
    . . .
    . . .
\end{ENVIRONMENT_NAME}
```

You do not have to indent the contents of an environment. Indentations neither affect how the document is compiled nor show up in the final PDF.[4] It is

[3] For a comprehensive list of LaTeX environments see: `http://latex.wikia.com/wiki/List_of_LaTeX_environments`.

[4] An aside: the `tabbing`[5] environment is a useful way to create tabbed text in LaTeX. We don't cover this here.

conventional to indent them, however, because it makes the markup easier to read.

In this chapter we will learn about two types of environments you need for tables in LaTeX. The `tabular` environment allows you to format the content of a table. The `table` environment allows you to format a table's location in the text and its caption.

The `tabular` environment

The `tabular` environment allows you to create tables in LaTeX. Let's work through the basic syntax for a simple table.[6]

To begin a simple tabular environment type `\begin{tabular}{TABLE_SPEC}`. The `TABLE_SPEC` argument allows you to specify the number of columns in a table and the alignment of text in each column. For example, to create a table with three columns, the first of which is left-justified and the latter two are center-justified we would type:

```
\begin{tabular}{l c c}
```

The `l` argument creates a left-justified column, `c` creates a centered one. If we wanted a right-justified column we would use `r`.[7] Finally we can add a horizontal line between columns by adding a vertical bar `|` between the columns arguments.[8] For example, to place a vertical line between the first and second column in our example table we would type:

```
\begin{tabular}{l | c c}
```

Now let's enter content into our table. We saw earlier how CSV files delimit individual columns with commas. In LaTeX's `tabular` environment columns are delimited with ampersands (`&`).[9] In CSV tables new lines are delimited by starting a new line. In LaTeX tables you use two backslashes (`\\`).[10] Here is a simple example of the first two lines of a table:

---

[6] For a comprehensive overview see the LaTeX Wiki page on tables: `http://en.wikibooks.org/wiki/LaTeX/Tables`.

[7] You can also specify a column's width by using `m{WIDTH}` instead. This will create a column of a specified width that is vertically justified in the middle. For example, `p{3cm}` would create a column with a width of 3 centimeters. Text in the column would automatically be wrapped onto multiple lines if need be. You can replace the `m` with either `p` or `b`. `p` vertically aligns the text at the top, `b` aligns it at the bottom.

[8] If you add two vertical bars (`||`) you will get two lines.

[9] If you want to include an ampersand in the text of your LaTeX document you need to escape it like this: `\&`.

[10] You can use two backslashes outside of the `tabular` environment as well to force a new

```
\begin{tabular}{l | c c}
    Observation & Variable1 & Variable2 \\
    Subject1 & a & b \\
```

It is common to demarcate the row with a table's column names–the first row–with horizontal lines. A horizontal line also often demarcates a table's end. You can add horizontal in the `tabular` environment with the `\hline` command.[11]

```
\begin{tabular}{l | c c}
    \hline
    Observation & Variable1 & Variable2 \\
    \hline \hline
    Subject1 & a & b \\
    \hline
```

Finally, we close the `tabular` environment with `\end{tabular}`. The full code (with a few extra rows added) is:

```
\begin{tabular}{l | c c}
    \hline
    Observation & Variable1 & Variable2 \\
    \hline \hline
    Subject1 & a & b \\
    Subject2 & c & d \\
    Subject3 & e & f \\
    Subject4 & g & h \\
    \hline
\end{tabular}
```

This produces the following table:

line. Also, to increase the space between the line you can add a vertical width argument to the double back slashes. For example, `\\[0.3cm]` will give you a three centimeter gap between the current line and the next one.

[11] The `\hline` command works outside of the `tabular` environment as well.

Observation	Variable1	Variable2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	h

Table float environment

You might notice that the table we created so far lacks a title and is bunched very closely to the surrounding text. In LaTeX we can create a `table` float environment to solve this problem. Float environments allow us to separate a table form the text, specify its location, and give it a caption.[12] To begin a `table`float environment use `\begin{table}[POSITION_SPEC]`. The `POSITION_SPEC` argument allows us to determine the location of the table. I can be set to `h` for here, i.e. where the table is written in the text. It can also be `t` to place it on the top of a page or `b` for the bottom of the page. To set a title for the table use the `\caption` command. LaTeX automatically determines the table's number, so you only need to enter the text. You can also declare a cross-reference key for the table with the `\label` command.[13] A `table` environment is of course closed with `\end{table}`. Let's see a full example.

```
\begin{table}[t]
    \caption{Example Simple LaTeX Table}
    \label{ExLaTeXTable}
    \begin{center}
        \begin{tabular}{l | c c}
            \hline
            Observation & Variable1 & Variable2 \\
            \hline \hline
            Subject1 & a & b \\
            Subject2 & c & d \\
            Subject3 & e & f \\
            Subject4 & g & h \\
            \hline
        \end{tabular}
    \end{center}
\end{table}
```

[12] We will see in the next chapter how to use `figure` floats as well.

[13] This command works throughout LaTeX. To reference the table type in the text of your document `\refKEY`, where `KEY` is what you set with the `\label` command. Use `\pageref` to reference the page number.

TABLE 9.1
Example Simple LaTeX Table

Observation	Variable1	Variable2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	h

Notice that the `tabular` environment is further nested in the `center` environment. This centers the table while leaving the table's title left-justified. The final result is Table 9.1.

You can see how typing out a table in LaTeX gets very tedious very fast. For all but the simplest tables it is best to try to have R do the table making work for you.

9.1.2 Tables in Markdown/HTML

Now we will briefly look at the syntax for creating simple Markdown and HTML tables before turning to learning how to have R create these tables for us.

Markdown tables

Markdown table syntax, as with all Markdown syntax is generally much simpler than LaTeX's tabular syntax. The markup is much more human readable. Nontheless, larger tables can still be tedious to create.

You do not need to declare any new environments to start creating a Markdown table. Just start typing. Columns are delimited in Markdown tables with a vertical bar (`|`). Rows are started with a new line. To indicate the head of the table–usually the row(s) containing the column names–separate it from the body of the table a row of dashes (e.g. `----`). Here is an example based on the table we created in the previous section:

```
Observation | Variable1  | Variable2
----------- | ---------- | ---------
Subject1    | a          | b
```

Note that it is not necessary to line up the vertical bars. You just need to have the same number of them on each row.

You can specify each columns text justification using colons on the dashed

row. For example this code will create the center-right-right justified format we did previously (with a few extra rows thrown in):

```
Observation | Variable1 | Variable2
:---------- | :-------: | :-------:
Subject1    | a         | b
Subject2    | c         | d
Subject3    | e         | f
Subject4    | g         | c
```

To create a right-right justified column simply place a colon on the right side of the dashes.

The ultimate look of a Markdown table is highly dependent on the CSS style file you are using (see Chapter 13 for how to change your CSS style file). The default RStudio CSS style file formats our table like this:

Observation	Variable1	Variable2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	c

Using a different CSS style file[14] we can get something like this:

OBSERVATION	VARIABLE1	VARIABLE2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	c

In basic Markdown you can add a caption with the heading syntax (see page 171). For example:

```
### Example Simple Markdown Table
Observation | Variable1 | Variable2
:---------- | :-------: | :-------:
Subject1    | a         | b
```

[14]The table was created using the Upstanding Citizen style from the program Marked.

will produce something like this:

Example Simple Markdown Table

OBSERVATION	VARIABLE1	VARIABLE2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	c

If you use MultiMarkdown you can use a more sophisticated caption by including the heading and cross reference key inside of square brackets (`[]`) either directly before or directly after the table. Please see Chapter 13 (page 172).

HTML tables

The *xtable* package we will learn in the next section doesn't create tables formatted by Markdown syntax. It can creates tables with HTML syntax. This is useful for us because virtually any HTML markup can be incorporated into a Markdown document. In fact, Markdown table syntax is only a stepping stone for more easily producing tables with HTML syntax. So it is useful to understand the basic syntax for HTML tables.

HTML uses element "tags" to begin and end tables. The main element we use to create tables is, well the `tables` element. This is very similar to LaTeX's `tabular` environment. An HTML element generally begins with a start tag and ends with an end tag. Clearly this is very similar to LaTeX's `\begin{}` and `\end{}` commands. Begin tags are encapsulated in a greater than and less than sign and include the element tag name (`<TAG>`). End tags are similar, but include a forward slash like this `</TAG>`. The content of the element goes between the start and end tags. For example:

```
<table>
    . . .
    . . .
</table>
```

As in LaTeX you are not required to tab the content of a table element, however, it does make the markup document easier to read and, as the number of tags proliferates, easier to write.

You can add element attributes inside of start tags.[15] For example, to add a border to the table use: `<table border="1">`.

Table rows are put inside of `tr` (table rows) element tags. Individual cells are delimited with `td` (standard cell) tags. Here is what the first row of our example table looks like in basic HTML:

```
<table>
    <tr>
        <td>Observation</td> <td>Variable1</td> <td><Variable2/td>
    </tr>
```

We can further delimit a table's header row(s) from its body with the `thead` and `tbody` tags. Finally, before making a full table its useful to mention that table captions can be include with `caption` tags. Let's put this all together:

```
<table>
    <thead>
        <tr>
            <td>Observation</td> <td>Variable1</td> <td>Variable2</td>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Subject1</td> <td>a</td> <td>b</td>
        </tr>
        <tr>
            <td>Subject2</td> <td>c</td> <td>d</td>
        </tr>
        <tr>
            <td>Subject3</td> <td>e</td> <td>e</td>
        </tr>
        <tr>
            <td>Subject4</td> <td>f</td> <td>f</td>
        </tr>
    </tbody>
</table>
```

As with Markdown tables, the ultimate appearance of the table is highly dependent on the style files you use.

[15] These work like arguments in R in that they change how the element is evaluated.

9.2 Creating tables from R objects

Just as the `read.csv` command turns an R data frame into a CSV formatted text file, there are a number of methods in R to take an object–e.g. a matrix, data frame, the output from a statistical analysis and so on–and turn them into LaTeX and HTML tables. In this section we will learn how to do this with the *xtable* and *apsrtable* packages.

9.2.1 *xtable* & *apsrtable* basics with supported class objects

The *xtable* and *apsrtable* packages are fairly easy to use if you want to convert an object of a class that they support into a table. Different R statistical model estimation commands can produce model summaries of different classes. For example, the `lm` (linear model) command creates model summaries of the lm class. For example, let's create a simple linear regression using the *swiss* data frame and `lm` command. This data frame is included with R by default. The simple linear regression model we are going to make has the *swiss* variable **Examination** as the dependent variable and **Education** as the only independent variable.[16]

```
# Fit simple linear regression model
M1 <- lm(Examination ~ Education, data = swiss)

# Show M1 class
class(M1)

## [1] "lm"
```

Improve footnote with Crawley

We can see by using the `class` that *M1* is of the `lm` class. *M1* contains items estimated by the linear regression model[17]such as the coefficient estimates and their standard errors. To get a summary of a model object's contents use the `summary` like this:

[16]For a description of these variables type `?swiss` into the console

[17]If you are unfamiliar with the syntax of R statistical estimation models the previous code might be confusing. In general 'dependent' (Y) variables are written first and are separated by the 'independent' (X) variables by a tilde `~`. The independent variables are generally separated by plus signs (`+`), typically to indicate the additive nature of the models.

```
# Show summary of M1 model object
summary(M1)

##
## Call:
## lm(formula = Examination ~ Education, data = swiss)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.932  -4.763  -0.184   3.891  12.498
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  10.1275     1.2859    7.88  5.2e-10 ***
## Education     0.5795     0.0885    6.55  4.8e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.77 on 45 degrees of freedom
## Multiple R-squared: 0.488,Adjusted R-squared: 0.476
## F-statistic: 42.9 on 1 and 45 DF,  p-value: 4.81e-08
```

To find a full list of object classes that *xtable* supports type `methods(xtable)` into the R console after you have loaded the package. To see *apsrtable*'s supported classes type `showMethods("modelInfo")` into your console.

9.2.1.1 *xtable* for LaTeX

Let's first look at how to create LaTeX tables with *xtable* by creating a table summarizing the estimates from the *M1* model object.

```
# Load xtable
library(xtable)

# Create LaTeX table from M1 and show the output markup
xtable(M1, caption = "Linear Regression, Dependent Variable:
                      Exam Score",
           label = "BasicXtableSummary",
           digits = 2)
```

```
## % latex table generated in R 2.15.2 by xtable 1.7-0 package
## % Fri Jan  4 12:47:48 2013
## \begin{table}[ht]
## \begin{center}
## \begin{tabular}{rrrrr}
##   \hline
##  & Estimate & Std. Error & t value & Pr($>$$|$t$|$) \\
##   \hline
## (Intercept) & 10.13 & 1.29 & 7.88 & 0.00 \\
##   Education & 0.58 & 0.09 & 6.55 & 0.00 \\
##    \hline
## \end{tabular}
## \caption{Linear Regression, Dependent Variable:
##                        Exam Score}
## \label{BasicXtableSummary}
## \end{center}
## \end{table}
```

This code creates a fully formatted LaTeX table of the *M1* model object's summary. The table looks like Table 9.2 when it is included in a LaTeX document. The *xtable* command not only produces complete both complete `tabular` and `table` environments, but also through the `caption` and `label` arguments also automatically adds in the table's title and cross-reference label, respectively. Finally, notice that I added the `digits = 2` argument. This specifies that I want numbers in the table to be rounded to two decimal digits.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	10.13	1.29	7.88	0.00
Education	0.58	0.09	6.55	0.00

TABLE 9.2
Linear Regression, Dependent Variable: Exam Score

9.2.1.2 *xtable* for 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 set it to `'html'`.

9.2.1.3 *apsrtable* for LaTeX

You need to include the `dcolumn` package in your LaTeX preamble, i.e. `\usepackage{dcolumn}` (see page 162 for a discussion of LaTeX preambles).

9.2.2 *xtable* with non-supported class objects

xtable is very convenient for making tables from objects in supported classes. 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. You need to help it find them.

xtable can handle 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 you need to

1. find and extract the information from the unsupported class object that you 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 you want to create,
3. use *xtable* with this object to create the table.

Imagine that you 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; Imai et al., 2012) and data from the *swiss* data frame that comes with R. First run the model:

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 Zelig package
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 a `MCMCZelig` class object. This class is not supported by *xtable*. If you try to create a summary table called *NBTable* of the results you will get the following error:

```
# Load xtable
library(xtable)

# Attempt to create a table with NBModel
NBTable <- xtable(NBModel)

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

With unsupported class objects you have to create the summary yourself and extract the elements that you want from it manually. A good knowledge of vectors, matrices, and component selection is very handy for this (see Chapter 3).

First, create a summary of your output object *NBModel*:

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

You created a new object of the class `summary.MCMCZelig`. You're still not there yet as this object contains not just the covariate names and so on but also information you don't want to include in your results table, like the formula that you used. The second step is to extract a matrix from inside *NBModelSum* called *summary* with the component selector (`$`). Remember that to see the components of an object you can use the `names` command. The *summary* matrix is where the things you want in your table are located. I find it easier to work with data frames, so let's also convert the matrix into a data frame.

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

Here is what your 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 you have a data frame object that *xtable* can handle. After a little cleaning up (see the chapter's source code for more details) you 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 9.3
Coefficient Estimates Predicting Examination Scores in Swiss Cantons (1888) Found Using Bayesian Normal Linear Regression

It may take some hunting to find what you want, but a similar process can be used to create tables from objects of virtually any class.[18] Hunting for what you want is generally easier if you look inside of it 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 tables is `results`. The `results` option can have one of three values:

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

The value `hide` clearly hides the results of you code chunk from your presentation document. To include tables created from R objects in your LaTeX or Markdown output you should set `results='asis'` or `results='markup'`. `asis` simply the writes the raw output in the presentation document where it

[18] This process can also be used to create graphics.

is then compiled with the rest of the markup. `markup` uses an output hook to mark up the results in a predefined way.

9.2.4 Knitting tables in LaTeX

In general, LaTeX tables created with *xtable* or *apsrtable* will take care of creating the content and formatting of the `tabular`. You will usually still want to embed this inside of a `table` environment to add a caption and cross-reference label.

LaTeX landscape tables

If your your LaTex table is very long, e.g. because it shows results from many estimation models, you can LaTeX's `lscape` package to create `landscape` formatting environments. Rather than orienting the text of a page so that it is in profile, a `landscape` environment turns it 90 degrees so that it oriented in landscape.

To use the *lscape* package first place `\usepackage{lscape}` in your LaTeX document's preamble (see page 162 for more information). Then begin a `landscape` environment with `\begin{landscape}` where you would like it located in the text. Then place the `table` environment information and *knitr* code for creating the table. Finally close the `landscape` environment with `\end{landscape}`.

10

Showing Results with Figures

One of the main reasons that many people use R is to take advantage of its very comprehensive and powerful set of tools for data visualization. Figures are often a much more effective way to present descriptive statistics and analysis results than the tables we covered in the last chapter. Dynamically incorporating figures with *knitr* has many of the same benefits as dynamically including tables, especially the ability to have data set or analysis changes automatically cascade into your presentation documents. The basic process for including Figures in knitted presentation documents is also very similar, though there are some important extra considerations we need to make to properly size the figures and include interactive visualizations in our presentation documents.

In this chapter we will learn some of the basics of R's powerful graphics capabilities, including base R graphics, *ggplot2* (Wickham and Chang, 2012b), *googleVis* (Gesmann and de Castillo, 2012), and *animation* (Xie, 2012a). In each case we will focus on how to include the figures in knitted presentation documents.

10.1 Including graphics

10.2 Basic knitr figure options

10.2.1 Chunk options

10.2.2 Global options

10.3 Creating static figures with ggplot2

10.4 Motion charts and basic maps with googleVis

Markus Gesmann and Diego de Castillo's *googleVis* packages allows us to easily use Google's Visualization API[1] to create interactive figures and maps. Because the visualizations are written in JavaScript they can be included in HTML presentation documents created by R Markdown. Unfortunately, they cannot be included in LaTeX produced PDFs. In the next section we will learn about *animation* package which does have some limited features for including interactive visualizations in PDFs.

Complete

Including googleVis in knitted documents

Using the `print(VISOBJECT, "chart")` prints the entire JavaScript code needed to create the visualization. The default *knitr* setting is to simply print the code, rather than run the JavaScript. This will give you a long code block. Not really what you are aiming for. To have the visualization show up in your HTML output, rather than the code block, simply set the code chunk option to `results='asis'`.

Important Note for Motion Charts

You may notice that Google motion charts do not show up in the RStudio Preview window or even in your web browser when you open the knitted HTML version of the file. You just see a big blank space where you had hoped the chart would be It will show up, however, if you use the `plot` command on a `gvis` motion chart object in the console. Motion charts can only be displayed

[1] For full details see: `https://developers.google.com/chart/interactive/docs/reference`.

when they are hosted on a web server or located in a directory 'trusted' by Flash Player.[2]

The `plot` command opens a local server, but simply opening the HTML file and the RStudio Preview window do not. An easy way to solve this problem is to simply save the HTML file in your Dropbox *Public* folder and access it through the associated public URL link (see Chapter 5). Publishing a motion chart on GitHub Pages also works well (see Chapter 13). For information on how to set a directory as 'trusted' by Flash Player see: `http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html`.

10.5 Animations

[2]Motion charts and annotated time line charts rely on Flash, unlike the other Google visualizations. For more information see Markus Gesmann's blog post at: `http://lamages.blogspot.com/2012/05/interactive-reports-in-r-with-knitr-and.html`.

Part IV

Presentation Documents

11

Presenting with LaTeX

This chapter gives you a quick introduction to basic LaTeX document structures and commands. In the next chapter (Chapter 12) we will build on these skills by learning how to use *knitr* to create more complex multi-part LaTeX documents.

Much of this chapter is incomplete.

11.1 The Basics

11.1.1 Editors

As I mentioned earlier, RStudio is a fully functional LaTeX editor as well as an integrated development environment for R. If you want to create a new LaTeX document you can click `File` in the menu bar then `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 Basic syntax

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

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

11.1.3 The preamble & the body

All LaTeX documents require a preamble. The preamble 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 can also 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 preamble 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 you 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 preamble and body like this:

```
\documentclass{article}

\begin{document}


\end{document}
```

11.1.4 Headings

11.1.5 Footnotes & Bibliographies

11.1.5.1 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. Of course remember to close the footnote with a }. LaTeX does the rest, including formatting and numbering.

11.1.5.2 Bibliographies

Citing R Packages with BibTeX

Researchers are pretty good about consistently citing others' articles and data. However, citations of R packages used in analyses is very inconsistent. This is unfortunate not only because correct attribution is not being given to those who worked to create the package, but also because it makes reproducibility harder. It obscures important steps that were taken in the research process,

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

primarily which package versions were used. Fortunately, there are R tools for quickly and dynamically generating citations, including the versions of the packages 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 inside of a code chunk. For example if you want the citation information for the `xtable` package you 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 you 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 you 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 output and/or the messages to a file. For example, imagine that your existing BibTeX file is called `biblio.bib`. To add the *xtable* 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 your `biblio.bib` file. It is **very important** to include the argument `append = TRUE`. If you don't you will erase the existing file and replace it with only the new citation. The argument `type = c("output")` tells R to include only the output, not the messages.

A more concise way to add citations to a bibliography is with `write.bibtex` command in the *knitcitations* package (Boettiger, 2012). To add the *xtable* citation to our `biblio.bib` file we only need to enter:

```
# Load package
library(knitcitations)

# Write xtable 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.

11.2.1 knitr LaTeX slideshows

Knitr largely works the same way in 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]
    \frametitle{An example fragile frame.}

\end{frame}
```

Results

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

[2]For 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

This chapter is largely incomplete

In the previous chapter you learned the basics of how to create LaTeX documents to present your research findings. So far you have only learned how to create short documents, like articles. For longer and more complex documents, like books, you 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 when creating batch reports: documents that present results for a selected part of a data set. For example, a researcher may want to 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 your 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 comprised of multiple chapters. They often include other child-like features such as title pages, bibliographies, figures, and appendices. You 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 your presentation markup files are difficult to navigate, they are difficult to reproduce.

12.1.2 Planning batch reports

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 uses *knitr* and is slightly more complex, but is more flexibile. 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

Include child documents with input

Include child documents with knitr

Child documents in a different markup language

Because *knitr* is able to run not only R code but also Bash command line programs, you can use the Pandoc 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 by including your Pandoc commands in a code chunk with the `engine` option set to either `'bash'` or `'sh'`.[2]

For example, the Stylistic Conventions part of this book is written in Markdown. The source file is called *StylisticConventions.md* It was simply faster to write the list of conventions using the simpler Markdown syntax than LaTeX, which 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

[1] Pandoc installation instructions can be found at: `http://johnmacfarlane.net/pandoc/installing.html`.

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

`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*.

```
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, Bash only recognizes a command's arguments if they are on the same line as the command. 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

This chapter is incomplete.

13.1 The Basics

13.1.1 Headings

Headings in Markdown are extremely simple. To create a line in the topmost heading style–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 Math

13.1.3 Escape characters

Markdown, like LaTeX and R uses backslashes (\) as an escape character. For example if you want to have an asterisk in the text of your document, rather than start an italicize your text type: `\*`. Two characters ampersand (`&`) and the less than sign (`<`) have special meanings in HTML[1] So, have them printed literally in your text you have to use HTML code for the characters. Ampersands are created with `&amp`. Less than signs are created with `&lt`.

13.1.4 CSS style files and Markdown

CSS style files in RStudio

[1] Ampersands declare the begining of a speacial HTML character. Less than signs begin HTML tags.

Drawing figures with CSS

13.1.5 Added features with MultiMarkdown

13.2 Simple webpages

13.2.1 RPubs

13.2.2 Hosting webpages with Dropbox

13.3 Reproducible websites

13.4 Presentations with `Slidify`

It is possible to create reproducible *knitr* HTML5 slideshows with R using Ramnath Vaidyanathan's *Slidify* package (**?**).[2] This package converts R Markdown files into HTML slideshows. There are a number of advantages to creating HTML presentations:

- You can use the relatively simple Markdown syntax.
- HTML presentations are a nice native way to show content on the web.
- Slidify presentations can incorporate virtually any content that can be included in a webpage. This includes interactive content, like motion charts created by *googleVis* (see Chapter 10).

There are a number of steps to create an HTML5 slideshow with *Slidify*:

- initialize a slideshow with the `author` command,
- edit the slideshows main R Markdown file, called *index.Rmd* by default. This includes both the files header and body.
- Use the `slidify` command to run *knitr* and compile the slideshow
- publish the slideshow online with the `publish` command.

We will cover each step in turn.

[2]For more information about Slidify please visit its excellent website at `http://ramnathv.github.com/slidify/`. For example, this site includes information on how to customize slideshow layouts.

TABLE 13.1
A Selection of HTML5 Slideshow Frameworks

Framework	Website for more information
io2012	`http://code.google.com/p/io-2012-slides/`
html5slides	`http://code.google.com/p/html5slides/`
deck.js	`http://imakewebthings.com/deck.js/`

HTML5 frameworks

Before getting into the details of how to use *Slidify*, let's briefly understand what an HTML5 slideshow is and the frameworks that make it possible. HTML5 slideshows rely on a number of web technologies in addition to HTML5, CSS,[3] JavaScript, to essentially create a website that behaves like a Latex beamer or Powerpoint presentation. They run in your web browser and you may need to be connected to the internet for them to work properly as key components may be located remotely. Most browsers have a `Full Screen` mode you can use for presentations.

There are a number of different HTML5 slideshow frameworks that let you create and style you slideshows. Table 13.1 lists some of the major frameworks supported by *Slidify*. In all of the framworks you advance through slides with the forward arrow button on your keyboard. Of course you can go back with the back arrow. Despite this similarity, the frameworks have different looks and capabilities. Check out their respective websites listed in Table 13.1 for more information. The URL's listed in Table 5 13.1 link to example slideshows.

fill in table

Installing Slidify

To get started with Slidify load the *devtools* packages and install its libraries from GitHub.[4]

```
# Load devtools
library(devtools)

# Install Slidify and ancillary libraries
install_github("slidify", "ramnathv")
```

[3] Cascading Style Sheets
[4] As of when I wrote this (December 2012) Slidify was not yet available on CRAN.

```
install_github("slidifyLibraries", "ramnathv")
```

Initializing a new slideshow

Use the `author` command to create a new slideshow. Imagine we want to create a new slideshow in *Presentation* folder of our *ExampleProject* called *MySlideShow*. To do this type:

```
# Set working directory
setwd("/ExampleProject/Presentation")

# Load Slidify
library(slidify)

# Create slide show
author("MySlideShow")
```

This will create a new folder with an R Markdown file called *index.Rmd*. It will also initialize a Git repository and create a folder called *assets*. The *assets* folder is where CSS, JavaScript, and other files needed to create the full slideshow are stored. Luckily, *Slidify* takes care of all these things for us. Though if you want to you can certainly customize these files.[5] You will primarily edit the *index.Rmd* file. You are can change the name of this file if you like.

The Slidify *header*

When you `author` a slideshow, *Slidify* automatically opens the *index.Rmd*.[6] First thing you will see in this file is the *Slidify* header:

```
---
title       :
subtitle    :
```

[5] See `http://ramnathv.github.com/slidify/customize.html` for more details on the best way to modify these files.

[6] If you are using RStudio the file will open in a new source tab. In the R application, it will open the file in your default text editor. Finally in command line R on Mac or Unix-like computers it will open in VIM.

```
author      :
job         :
framework   : io2012        # {io2012, html5slides, shower, dzslides, ...}
highlighter : highlight.js  # {highlight.js, prettify, highlight}
hitheme     : tomorrow      #
widgets     : []            # {mathjax, quiz, bootstrap}
mode        : selfcontained # {standalone, draft}
---
```

The first four lines relate to what will appear on the slideshow's title slide, i.e. the title, subtitle, author, and job.[7] The next five lines affect the slideshow's formating. The framework line allows you to change the slideshow's overall type. It is currently set by default to Google's *io2012* framework. You can see a number of other supported formats on the right side of the line. These include *html5slides*, Opera's *shower* format and *dzslides*. You can use one of these other formats by deleting `io2012` after the colon and replacing it with the name of your desired framework.

The following two lines (`highlighter` and `hitheme`) relate to which syntax highlighting theme you would like code chunks to be formatted with. The default highlighter is *highlighter.js*[8] with the *tomorrow* theme.[9]

The next line allows you to automatically include a number of different widgets. As we saw earlier in this chapter, the *Mathjax* widget lets us view well formatted math in Markdown produced documents. The *bootstrap* widget lets you take advantage of, among other things, the wide range of JavaScript plug-ins available from Twitter Bootstrap.[10] To add widgets, type their name in the square brackets (`[]`) separated by a comma.

Finally there is the `mode` option. In general you will want to use the default `selfcontained` mode.

Get more info on mode.

Slide frames and slide titles

Slidify R Markdown documents use very similar syntax to ordinary R Markdown documents. *knitr* code chunks are written in the same way. An important difference is that three dashes (`---`) delimit individual slide frames. Importantly, you need to have an empty line before and after the three dashes or else a new slide will not be created. Two hash marks (`##`) are used to indicate a slide's title.[11]

[7] This is intended as a place to put your job title and affiliation.

[8] See: `http://softwaremaniacs.org/soft/highlight/en/`.

[9] See: `https://github.com/chriskempson/tomorrow-theme`.

[10] See: `http://twitter.github.com/bootstrap/javascript.html`. For an example of how you can combine Twitter Bootstrap's *Carousel* plug-in with *googleVis* to create interactive timeline maps in slide shows see: `http://ramnathv.github.com/carouselDemo/#1`.

[11] One hash mark does creates a slide title formatted in the same way as the text.

Compiling a Slideshow

Use the `slidify` command to compile an R Markdown file into a slideshow. This run *knitr* and parse your R Markdown file into a slideshow:

```
# Change to slideshow's working directory
setwd("/ExampleProject/Presentation/MySlideShow")

# Compile the slideshow
slidify("MySlideShow.Rmd")
```

In RStudio you can click the `Knit HTML` button and it will 'slidify' the R Markdown file. The slideshow will likely not work in the RStudio preview window, but opening the *index.html* file in your web browser works fine.

Publishing Slidify slideshows

You can of course show slideshows on your own computer by opening the *index.html* file in a web browser. If you want to make your slideshow available to anyone with an internet connection use Slidify's `publish` command. This will allow you to publish your presentation via GitHub, Dropbox, or RPubs. Because we've already learned about how to use GitHub and Dropbox, I'll focus on using these two services to publish your slideshow.

To publish our example *MySlideShow* on GitHub first create a new GitHub repository called 'MySlideShow' (see Chapter 5 for instructions on how to create a new repository). Make sure the repository is empty, i.e. has no files in it. Then type in R:

```
publish(user = "USER", repo = "MySlideShow")
```

`USER` is your GitHub user name. This will create a new GitHub Pages website where your slideshow will be accessible to anyone on the internet. We look at GitHub Pages in more detail later in this chapter.

To use the webpage hosting abilities of Dropbox public folders type:

```
publish("MySlideShow", host = "dropbox")
```

This will create a new directory in your Dropbox *Public* folder. To get the

URL address for the slideshow navigate to the folder and copy the public link for the *index.html* file (see Chapter 5 for more details).

Note: Before you use the `publish` command you will need to have set up accounts for the respective services before publishing a slideshow to them. In the GitHub and Dropbox cases you also need to have set up the services on your computer. Please refer back to Chapter 5 for more details on how to set up these services.

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 want to create a LaTeX formatted PDF. As we saw in the previous chapter, 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 languages[12] to any of the other formats.

[12]See the Pandoc website for more details: `http://johnmacfarlane.net/pandoc/`

14

Going Beyond the Book

This chapter is incomplete.

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

Arel-Bundock, V. (2012a). *countrycode: Convert country names and country codes*. R package version 0.9.

Arel-Bundock, V. (2012b). *WDI: World Development Indicators (World Bank)*. R package version 2.2.

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 Protcol 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.

Boettiger, C. (2012). *knitcitations: Citations for knitr markdown files*. R package version 0.1-0.

Boettiger, C. and Temple Lang, D. (2012). Treebase: an R package for discovery, access and manipulation of online phylogenies. *Methods in Ecology and Evolution*, 3(6):1060–1066.

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.

Carslaw, D. and Ropkins, K. (2012). *openair: Tools for the analysis of air pollution data*. R package version 0.7-0.

Couture-Beil, A. (2012). *rjson: JSON for R*. R package version 0.2.11.

Crawley, M. J. (2013). *The R Book*. John Wiley and Sons Ltd., Chichester, 2nd edition.

Dahl, D. B. (2012). *xtable: Export tables to LaTeX or HTML*. R package version 1.7-0.

Donoho, D. L. (2002). How to be a highly cited author in mathematical sciences. *in-cites*. `http://www.in-cites.com/scientists/DrDavidDonoho.html`.

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.

Elff, M. (2012). *memisc: Tools for Management of Survey Data, Graphics, Programming, Statistics, and Simulation*. R package version 0.95-38.

Fomel, S. and Claerbout, J. F. (2009). Reproducible Reserarch. *Computing in Science & Engineering*, 11(1):5–7.

Frazier, M. (2008). Bash parameter expansion. *The Linux Journal*.

Gandrud, C. (2012). The diffusion of financial supervisory governance ideas. *Review of International Political Economy*, pages 1–36.

Gentry, J. (2012). *twitteR: R based Twitter client*. R package version 0.99.19.

Gesmann, M. and de Castillo, D. (2012). *googleVis: Interface between R and the Google Chart Tools*. R package version 0.3.3.

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.

Imai, K., King, G., and Lau, O. (2012). *Zelig: Everyone's Statistical Software*. R package version 3.5.5.

Kabacoff, R. I. (2011). *R in Action: Data Analysis and Graphics with R*. Manning Publications Co., Shelter Island, NY.

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.

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.

Malecki, M. (2012). *apsrtable: apsrtable model-output formatter for social science*. R package version 0.8-8.

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.

Murdoch, D. (2012). *tables: Formula-driven table generation*. R package version 0.7.

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. Restructring 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.

Piwowar, H. A., Day, R. S., and Fridsma, D. B. (2007). Sharing Detailed Research Data Is Associated with Increased Citation Rate. *PLoS ONE*, 2(3):1–5.

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/`.

Ripley, B. and Murdoch, D. (2012). *Rtools: Building R for Windows*.

RStudio (2012). *RStudio: Integrated development environment for R*. Boston, MA. Version 0.97.142.

Ryan, J. A. (2011). *quantmod: Quantitative Financial Modelling Framework*. R package version 0.3-17.

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.

Temple Lang, D. (2012a). *RCurl: General network (HTTP/FTP/...) client interface for R*. R package version 1.95-3.

Temple Lang, D. (2012b). *RJSONIO: Serialize R objects to JSON, JavaScript Object Notation*. R package version 1.0-1.

Temple Lang, D. (2012c). *XML: Tools for parsing and generating XML within R and S-Plus*. R package version 3.95-0.1.

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.

Warnes, G. R., with contributions from Ben Bolker, Gorjanc, G., Grothendieck, G., Korosec, A., Lumley, T., MacQueen, D., Magnusson, A., Rogers, J., and others (2012). *gdata: Various R programming tools for data manipulation*. R package version 2.12.0.

Wickham, H. (2011). *reshape: Flexibly reshape data*. R package version 0.8.4.

Wickham, H. (2012a). *httr: Tools for working with URLs and HTTP*. R package version 0.2.

Wickham, H. (2012b). *plyr: Tools for splitting, applying and combining data*. R package version 1.8.

Wickham, H. (2012c). *reshape2: Flexibly reshape data: a reboot of the reshape package*. R package version 1.2.2.

Wickham, H. and Chang, W. (2012a). *devtools: Tools to make developing R code easier*. R package version 0.8.

Wickham, H. and Chang, W. (2012b). *ggplot2: An implementation of the Grammar of Graphics*. R package version 0.9.3.

Xie, Y. (2012a). *animation: A gallery of animations in statistics and utilities to create animations*. R package version 2.1.

Xie, Y. (2012b). *formatR: Format R Code Automatically*. R package version 0.7.

Xie, Y. (2012c). *knitr: A general-purpose package for dynamic report generation in R*. R package version 0.9.

Index

.gitignore, 83
formatR, 25

Amazon S3, 72
ampersand in LaTeX, 141
animation, 155
API, 18, 105, 156
apsrtable, 139
argument, 38
assignment operator, 31
attach, 35, 118
Awk, 134

Bash, 134, 168
batch reports, 63, 167
beamer, 173
bitly, 74

cache, 45, 131
cache code chunks, 11
CamelBack, 57
caption, LaTeX command, 143
cast, 113
cat, R command, 62
cat, shell command, 66
cbind, 33, 121
cd, 64
child directories, 56
child files, 167
chunk hooks, 49
class, R command, 148
cloud storage, 55
code chunk, 12, 44
code chunk options, 45
comma-separated values, 23, 72
command line, 14
comment declaration, 23
component selection, 35
concatenate, 33
cp, 67
CRAN, 40, 108, 173
CRAN archive, 130
cross-sectional time-series data, 111
CSS, 133, 173, 174
CSV, 95, 96

data file formats, 72
data frame, 32, 34
dcast, 113
dir.create, 61
directories, 56
Donald Knuth, 11
drive letter assignment, 56
Dropbox, 27, 157, 176
Dropbox Public folder, 74
duplicated, 123

echo, 66, 130
element tag, HTML, 146
engine, 133
error, 131
escape character, 56, 132, 161
escape character, Markdown, 171
eval, 130

factor variable, 118
factor, command, 120
file compression, 104
file extension, 49
file path naming conventions, 55
file.copy, 63
file.create, 61
file.rename, 62
Flash Player, 157
float, LaTeX, 143

foreign, 101

Gawk, 134
getURL, 18, 103
getwd, 60
ggplot2, 155
Gist, GitHub, 138
Git, 12, 63, 76, 174
Git Bash, 78
git branch, 82
git clone, 85
git commit, 80
Git commit object, 82
Git ignore files, 83
git merge, 83
git pull, 86
git push, 85
Git upstream tracking, 88
GitHub, xvi, 14, 27, 59, 103, 137, 173, 176
GitHub Pages, 157, 176
GitHub repository, 176
global chunk options, 46, 168
GNU Make, 94, 95
GNU make, 135
Google R Style Guide, 25
googleVis, 155, 156, 172
Graphical User Interface, 11
GUI, 11

Haskell, 134
head, 109
help file, 38
Highlight, knitr engine option, 184
hook, 153
hooks, 49
HTML, 108
HTML tables, 146
HTML5, 172
httr, 108

include, 131
includegraphics, 18
input, 168
install.packages, 40
integrated developer environment, 10
International Standards Organization, 107
is.na, 117

JavaScript, 156, 173, 174
Jon Claerbout, 4
JSON, 108

knit, 4
knitr, 4, 11, 43

label, LaTeX command, 143
landscape tables, LaTeX, 150
LaTeX begin document, 162
LaTeX class, 162
LaTeX distribution, 13
LaTeX preamble, 162
LaTeX table environment, 140
LaTeX tables, 140
Linux, 10
list, 32
list.files, 61
lists, 32
literate programming, 10, 11, 25
local chunk options, 46
locally stored, 55
logical operators, 117
long formatted data, 111
loop, 108
ls, 39, 65
Lyx, 12, 53

Mac, 12, 30
makefile, 63, 94, 135
markdown package, 13
Markdown tables, 144
Markdown/HTML tables, 144
Marked, 145
markup language, 11
Markus Gesmann, 156
MatLab, 4
matrix, 32, 34
matrix transpose, 112
mean, 34

mean, R command, 132
melt, 113
memisc, R package, 139
merge, 121
message, 131
Microsoft Excel, 23
Microsoft Word, 11, 23
mirrors, CRAN, xvi, 40
mkdir, 65
MultiMarkdown, 163
mv, 67

NA, 32
names, 109
notebook, 49
nrow, 124

object-oriented, 30
operating systems, 56
order, 114
outliers, 115
output hooks, 49
outreg, 136

package options, 48, 168
packages, 21
pageref, LaTeX command, 143
Pandoc, 168
parameter expansion, 98
parent directory, 56
parent document, 167
Powerpoint, 173
prerequisites, Make, 96
progress bar, 48
pwd, 64
Python, 133, 134

R CMD BATCH, 98
R console, 29
R function, 136
R LaTeX, 49
R Markdown, 49
R packages, 40
R session, 30
R Sweave, 43, 161
rbind, 33, 121
read.table, 18, 101, 136
README file, 59
recipe, Make, 96
recode, 117
regular expressions, 108
remote, 88
repository, 76, 79
reproducible research environment, 10
reproducible research publisher, 10
Republic of Korea, 117
reshape data, 111
reshape, command, 112
reshape2, 112
reStructuredText, 11
results, knitr option, 153
rjson, 108
RJSONIO, 108
rm, 40, 66, 98
rm, Git command, 84
root directory, 56
rOpenSci, 106
round, R command, 132
RPubs, 176
RStudio, 12, 41, 176
RStudio Notebook, 43
RStudio Options window, 52
RStudio pane, 41
RStudio Projects, 55, 59, 78, 87
Rtools, xvi, 96
Ruby, 134

save.image, 39
scraply, 108
session info, 21
setwd, 61, 94
Sexpr, 132
SHA, 82
slideshow, 172
Slidify, xvi, 172
sort, 114
source, 136
source code, 11
source command, 42, 94, 135

Source pane, 49
source_gist, 138
Stata, 101
style guide, 25
subdirectory, 56
subscripts, 35
subset, 115
sudo, 65
summary, R command, 109, 148
SVN, 12
Sweave, 11, 51
syntax highlighting, 11, 12, 175
system, R command, 67, 135

tab-delimited values, 72
tab-separated values, 72
table, LaTeX environment, 143
table, R package, 139
tabular, LaTeX environment, 141
tail, 109
targets, Make, 96, 98
td, HTML element, 147
Terminal, 30, 78, 96
tie commands, 18, 25
time stamp, 95
tr, HTML element, 147
TSCS, 111
Twitter Bootstrap, 175

Ubuntu, 30
unique, 124
Unix, 12, 30
Unix-like shell program, 10, 55, 63
unlink, 62

vector, 32
verb, LaTeX command, 132
version control, 60
View R command, 110
VIM, 174

warning, 131
WDI, 106
weave, 4
web scraper, 74
web scraping, 107
wide formatted data, 111
wiki, 77
wildcard, 84, 98
wildcard, Make function, 98
Windows, 12
Windows PowerShell., 10
with, 35
working directory, 57, 61
workspace, 39
wrapper, 73
write.csv, 73
write.table, 72
WYSIWYG, 12, 53

XML, 108
xtable, 139