Christopher Gandrud

Reproducible Research with R and RStudio

Author

Christopher Gandrud
Yonsei University
Wonju, Republic of Korea

Forward

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 reader of blogs. Without him the ability to do reproducible research would be much harder and the blogging community that spreads knowledge about how to do these things would be poorer. Other great bloggers include Carl Boettiger (who also developed the *knitcitations* package), Markus Gesmann (who developed *GoogleVis*), Jeromy Anglim.

The vibrant and very helpful community at Stack Overflow `http://stackoverflow.com/` is always very helpful for finding answers to small problems that plague any coder. Importantly it makes it easy for others to find these answers.

iv

Preface

FILL IN

vi

Stylistic Conventions

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

- Abstract Variables

Abstract variables, i.e. variables that do not represent specific objects in an example, are in `ALL CAPS TYPWRITER TEXT`.

- Clickable Buttons

Clickable Buttons are in `typewriter text`.

- Code

All code is in `typewriter text`.

- Filenames and Directories

Filenames and directories more generally are printed in *italics*.
Camelback is used for file and directory names.

- Individual variable values

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

- Objects

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

- Columns

Columns are printed in *italics*

- Packages

R packages are printed in *italics*.

- Windows

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

- Variable Names

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

Required R Packages

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

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

x

List of Figures

List of Tables

Contents

xx

Part I

Getting Started

1

Introducing Reproducible Research

CONTENTS

Most people encounter research as a neat and very abridged package. This package is usually in the form of a conference presentation, journal article, book, or maybe even a website. These presentation documents announce the results of some research project and try to convince us that the results are correct.[6]. However, the article, slideshow, or book is not the research.

1.1 What is reproducible research?

FILL IN

1.2 Why should research be reproducible?

Incorporating high reproducibility into your research is important for science and it can also make your life as a researcher easier.

1.2.1   For Science

1.2.2   For You

Working to make your research reproducible from the start has a number of knock on benefits that make the research process easier *for the researcher*. A third person may or may not actually reproduce your research even if you make it easy to do so. But, it's almost certain that you will reproduce parts or even all of your research. Virtually no actual research process is completely linear. We almost never gather data, run our analyses, and present our results without also going backwards to add variables, make changes to our statistical models, create new graphs, and so on. Whether these changes are because of journal reviewers' and conference participants comments or we discover that new and better data has been made available since beginning the project, designing our research to be reproducible from the start makes it much easier to make these changes.

Changes made to one part of a research project have a way of cascading through the other parts. For example, adding a new variable to a largely completed analysis requires gathering new data and merging it into existing data sets. If we are using data imputation or matching methods this can lead to adjustments to the entire data set. We then have to update our statistical models and the tables and graphs we use to present results. Adding a new variable essentially forces us to reproduce large portions of our research. If we made it easier for others do reproduce our research we also made it easier for us to do this. Jake Bowers has referred to this as taking steps to have a "better relationship with our future selves".[1]

COMPLETE

1.3   Who should read this book?

This book is intended primarily for upper-level undergraduate and graduate students and professional researchers who want to develop a more systematic workflow that encourages reproducibility.

1.3.1   Students

1.3.2   Researchers

1.3.3   Industry practitioners

Industry practitioners 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, making your research reproducible to members of your organization can save and pass on valuable information about how analyses were done and data collected. Just as a lack of reproducibility hinders the spread of information in the scientific community, it can hinder it inside of a private organization. COMPLETE

1.4 The Tools of Reproducible Research

This book will teach you the tools you need to make your research highly reproducible. Reproducible (quantitative) research involves two broad sets of tools. The first is a **reproducible research environment** that includes the statistical tools you need to run your analyses as well as "the ability to automatically track the provenance of data, analyses, and results and to package them (or pointers to persistant versions of them) for redistribution". The second set of tools is a **reproducible research publisher** that prepares documents to present the results and is easily linked to the reproducible research environment.[6]

In this book I we will focus on learning how to use widely available and highly flexible reproducible research environment–R/RStudio. R/RStudio can be linked to numerous reproducible research publishers with Yihui Xie's *knitr* package[9]. The full list of tools tools covered in this book include:

- **R**: a programming language primarily for statistics and graphics. It can also be used for data gathering and creating presentation documents.

- ***knitr***: an R package for literate programming, i.e. it allows us to combine our statistical analysis and the presentation of the results into one document.

- **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 research, save and document previous versions, and make this information widely available.

- **Unix-like shell programs**: These tools are useful for setting up and working with large research projects.[1] They also allow us to use command line tools like Pandoc for converting documents from one markup language to another.

---

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

1.5   Why use R/RStudio for reproducible research?

Why R?

Why use a statistical programming language like R for reproducible research? R is more than just a statistics program, like , Stata, or SPSS. It can be used to integrate all stages of the research process not just the statistical analysis stage. As we will see in this book, R can be used to gather data, run statistical analyses, and we can use the *knitr* R package to connect our analysis to our presentation documents created with markup languages such as LaTeX, Markdown, and HTML we can dynamically present our results in articles, slideshows, and webpages.[2]

The way we interact with R or any other programming and markup language promotes reproducibility more than our interactions with Graphical User Interface (GUI) programs like SPSS[3] and Microsoft Word. When we write (and save) R code and embed it in documents using markup languages we are being forced to explicitly express the steps we take to do our research. When we do research clicking through drop down menus in GUI programs, the steps we take are lost. Or at least documenting them requires considerable extra effort.

Why RStudio?

Why use the RStudio integrated development environment for reproducible research? R by itself has the capability 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 us to do all of these things, but simplifies many of these tasks and allows you to navigate them in a more visual way. It is a as happy medium between R's text-based interface and a pure GUI.

RStudio is designed for reproducible research. RStudio is very tightly integrated with technologies such as `knitr`, Markdown, and LaTeX that enable us to present reproducible results. Compiling LaTeX PDF documents or HTML webpages in RStudio requires many fewer steps than doing the same thing in plain R.

Not only does RStudio do many of the things that R can do, but more easily it is very good stand alone editor for writing documents with LaTeX, Markdown, HTML. There are many LaTeX editors available, both open source and paid, as well as other ways to compile LaTeX documents, including directly

---

[2]When we use these types of markup languages to create presentation documents what we write is text plus the instructions for how to turn this into a final document. The instructions are written using a markup language. Part IV of this book (Presentation Documents) discusses how to use the LaTeX and Markdown languages for reproducible research.

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

through the command line. RStudio is currently the best program for creating reproducible LaTeX, Markdown, and HTML documents. It has full syntax highlighting, even for documents with `knitr` code (which it can collapse when you just want to work on the text). It can spell check LaTeXdocuments. It handles `knitr` code chunks beautifully making it easy to navigate through complex documents and run individual chunks. For LaTeXdocuments it can, for example, insert common commands like `\section*{}` for unnumbered sections or set up lists.

Finally, RStudio not only has tight integration with various markup languages, it also integrates other tools such as CSS, JavaScript, and a few other programming languages as well as being 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. This is important for reproducible research since the version control program is documenting many of the the steps you took to make your project. As with many tools of reproducible research, version control has other benefits for making researchers' lives easier. Maybe most importantly, when you keep your documents under version control you can go back to older versions. This is useful if you accidentally delete an important paragraph, for example.

1.5.1 Installing the Software

Before you read this book you should install the software. All of the software covered in detail in this book is open source programs and can be easily downloaded for free. It is available for Windows, Mac, and Unix. They should run well on most modern computers.

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

- **R**: `http://www.r-project.org/`,

- **RStudio**: `http://rstudio.org/download/`.

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

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

Installing markup languages

If you are planning on creating LaTeXdocuments you need to install a LaTeXdistribution. They are also open source and available for Windows, Mac, and Unix. They can be found at: `http://www.latex-project.org/ftp.html`. Please refer to that site for more installation information.

If you want to create markdown documents you will need to install the the

in R. You can do this the same way that you install any package in R, with the `install.packages` command.[4]

The command line

Cloud storage & versioning

1.6   Book overview

The purpose of this book is to give you the tools that you will need to do reproducible research with R and RStudio.

1.6.1   What this book is not.

This book describes a workflow for reproducible research primarily using R and RStudio. It is designed to give you the necessary tools to use this workflow for your own research. It is not designed to be a complete introduction to R, RStudio, GitHub, the command line, or any other program that is a part of this workflow. Instead it shows you how these tools can fit together to make your research more reproducible.

To get the most out of these individual programs I point you to other resources that cover these programs in more detail.

That being said, my goal in this for this book to be self-sufficient to the extent that 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 often encountered 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 frustratingly mysterious conventions and/or pointing you in the direction of a good explanation.

To that end, I can recommend a number of books for that cover more of the nitty-gritty of R and the command line.

- Michael J. Crawley's encyclopaedic R book, appropriately titled, **The R Book** published by Wiley.

- Norman Matloff's tour through the programming language aspects of R called **The Art of R Programming: A Tour of Statistical Design Software** published by No Starch Press.

- For an excellent introduction to the command line in Linux and Mac,

---

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

though with pretty clear implications for Windows users if they are running PowerShell (see Chapter 2) see William E. Shotts Jr.'s book **The Linux Command Line: A Complete Introduction** also published by No Starch Press.

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

1.6.2 How to read this book.

This book tells a story. It has a beginning, middle, and end. So, unlike a reference book it can and should be read like a novel, taking you through an empirical research processes from an empty folder maybe called *ResearchPaper* to a completed set of documents that 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 a LaTeX) document into a PDF. Please feel free to do this.

If you are experienced with R in particular you may want to skip over the first two sections of Chapter 3: Getting Started with R/RStudio. The latter part of this chapter contains important basic information on the *knitr* package.

1.6.3 How this book was written

This book practices what it preaches. It can be reproduced. It was written using the programs and methods that it describes. Full documentation and source files can be found at the Book's **GitHub** repository. Feel free to read and even copy (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.

In the same spirit, I encourage you to make your research files–not just data, but analysis code and markup–available for other researchers to learn from. Not only does reproducibility help us evaluate past work, but it also pushes forward knowledge in the scientific community.

1.6.4 Contents overview.

2

Getting Started with Reproducible Research

CONTENTS

Researchers often start thinking about making their research reproducible near the end of the research process when they get start writing up the results. Or maybe later, like when a journal has conditioned an article's acceptance on making its data available or another researcher asks if they can use the data. By this point there may be various versions of the data set and records of the analyses strewn across multiple folders on the researcher's computer. It can be difficult and time consuming to create an accurate account of how the results were reached. As a result many attempts at providing replication information are incomplete and may not give us an accurate account of how research results were found. Keeping your eye on reproducibility from the beginning of the research process and continuing to follow a few simple guidelines throughout your research can help solve these problems.

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

2.1 The Big Picture: A workflow for reproducible research

To make our research accurately reproducible we should start thinking at the beginning–the data gathering stage–about how other researchers and our-

selves will be able to reproduce our results. This book teaches the tools for an integrated workflow for reproducible research. It covers tools that we can use across the three basic stages that most researchers–especially quantitative researchers–go through to answer their research questions:

- data gathering,

- data analysis,

- results presentation.

2.1.1   Data Gathering

It is common in many disciplines (I know this is true in political science, my discipline) to

2.1.2   Data Analysis

2.1.3   Results Presentation

2.2   Practical tips for reproducible research

Before we start learning the details of reproducible research with R and RStudio it is useful to cover a few broad tips that will help us organize our research process and put these skills in perspective. The tips are:

1. Document everything!,

2. Everything is a file,

3. All files should be human readable,

4. Reproducible research projects are many files explicitly tied together,

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

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.

The other tips and Many of the other One important part of documenting everything with R is to *record your session info*. Many things in R stay the same over time, which makes it easy for future researchers to recreate what was done in the past. However, things–syntax in particular–do can change

from one version of R to another. Also, the way R functions may be handled slightly different on different operating systems. Finally, you may have R set to load packages by default. These packages might be necessary to run your code, but other people might not be able know what packages were loaded from just looking at your source code. The `sessionInfo` command prints a record of all of these things. The information from the session I used to create this book up until this chapter is:

```
sessionInfo()

## R version 2.15.1 (2012-06-22)
## Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)
##
## locale:
## [1] C/en_US.UTF-8/C/C/C/C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
## [1] knitcitations_0.1-0 bibtex_0.3-0
## [3] knitr_0.7.9
##
## loaded via a namespace (and not attached):
##  [1] RCurl_1.91-1    XML_3.9-4       codetools_0.2-8
##  [4] digest_0.5.2    evaluate_0.4.2 formatR_0.6
##  [7] pkgmaker_0.8    plyr_1.7.1     stringr_0.6.1
## [10] tools_2.15.1    xtable_1.7-0
```

It is good practice to include make output of the session info available either in the main document or in a separate text file. If you want to nicely format the information for a LaTeX document simply use the `toLatex` command.

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

Otherwise, you can use the `print` command.

2.2.2 Everything is a (text) file

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

possible to store this information. Usually the simplest file format is the the humble, but versatile, text file is the simplest file format.[1]

Text files are extremely nimble. They can hold data in, for example, comma-separated values `.csv` files. They can contain our analysis code in `.R` files. And they can be the basis for our presentation documents as markup documents like `.tex` or `.md`, for LaTeXand Markdown files respectively. All of these files can be opened by any program that can read text files–i.e. files with the generic file extension `.txt`.

The main reason reproducible research is best stored in text files is that this helps **future proof** our research. Other common file formats, like Microsoft Word or Excel documents change regularly and may not be compatible with future versions of these programs. Text files, on the other hand, can be opened by a very wide range past and, more likely than not, future programs. Even if future researchers do not have R or a LaTeXdistribution, they will still be able to open our text files and, aided by frequent comments, be able to understand how we conducted our research.

Text files are also very easy to search and manipulate with a wide range of programs–such as R–that can find and replace text characters as well as merge and separate files. They have a number of clear benefits for reproducible research including enabling versioning and track changes in 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 try to read them and understand them. Computer code is a way of communicating with the computer. It is 'machine readable' in that the computer is able to use it to understand what we want done.[2] Hopefully, the researcher understands what they are communicating when they write their code. However, there is a very good chance that other people (or the researcher six months in the future) will not understand what is being communicated. So, you need to make your documentation 'human readable'. To make your documentation accessible to other people you need to **comment frequently** and **format your code using a style guide**. For especially important piece of code you should use **literate programming** so that it is very clear to others how you accomplished a piece of research.

Commenting

In R everything on a line after a `#` hash (number) character is ignored by R, but is readable to people who open the file. The hash character is a comment

---

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

declaration character. You can use the `#` to place comments telling other people what you are doing. Here are some examples:

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

## [1] 4
```

Because on the first line the `#` is placed at the very beginning, 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 fines the answer `4`, but it ignores all of the words after the hash.

Different languages have different comment declaration characters. In LaTeXeverything 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.

Style guides

In natural language writing you don't necessarily need to follow a for things such as punctuation. 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 R code. It's good to follow consistent rules for formatting your code so that:

- it's easier for others to understand,

- it's easier for you to understand.

There are a number of R style guides. Most of them are similar to the Google R Style Guide (`http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`). Hadley Wickham has a nicely presented style guide. You can find it at `https://github.com/hadley/devtools/wiki/Style`. You can also use the `formatR` 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 statistical model and an explanation of this code in an appendix following your article. This is commonly referred to as literate programming [3]. This book discusses how to use the *knitr* package, a very useful tool for literate programming.

2.2.4   Reproducible research projects are many files explicitly tied together

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

The most dynamic way to do this is to explicitly link your files together using what I'll call **tie commands**. These commands instruct the computer program you are using to gather data, run an analysis, or compile a presentation document to use information from another file. I have compiled the main tie commands used in this book in Table 2.1.

TABLE 2.1

Commands for Tying Together Your Research Files

Command/Package	Language	Description	Chapters for Further Information
knitr	R	R package with functions for tying R commands into presentation documents	Discussed throughout the book
read.table	R	Reads a table into R	6
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
input	LaTeX	Includes LaTeXfiles inside of other LaTeXfiles	12
include	LaTeX	Similar to input, but puts page breaks on either side of the included text. Usually this it is used for including chapters chapters.	12

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

3

Getting Started with R, RStudio, and knitr

CONTENTS

If you have rarely or never used R before the first two sections of this chapter give you enough information to be able to get started and understand the code I use in this book. For more detailed introductions to R please refer to the related resources I mentioned in chapter 1.6.1. Experienced R users might want to skip the first two sections of the chapter. This chapter also gives a brief overview of RStudio. It highlights the key features of 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 and how it is integrated into RStudio.

3.1 Using R: the basics

As a computer language, R has FILL in

This section covers some of the very basic syntax in R to get you started. If you have little experience with R, reading this section will make it much easier for you to follow along with the examples in the book. I cover the key components of the R language including:

19

- objects & assignment,

- component selection,

- functions, commands, and arguments,

- libraries.

Before discussing each of these components let's open up R and look around.[1] When you open up R you should get a window that looks something like what you see in Figure 3.1.[2] This window is the **R console**. Under the session information–what version of R you are using, your workspace, and so on–you should see a >. This is where you enter R code.[3] To run R code that you have typed into the console type the `Enter` or `Return` key. Now that we have a new R session open we can get started.

3.1.1   Objects

You will probably have read that 'R is an object-oriented language'. What are objects? Objects are like the R language's nouns. They are things, such as a list 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.

Numeric & string objects

Different types of objects have different data types or modes. Let's make two basic objects of the numeric and character mode. We can choose almost any name we want for our objects.[4] Let's call our numeric object *Number*. To put something into the object we use the **assignment operator**: <-. Let's assign the number 10 to our Number object.

```
Number <- 10
```

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

---

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

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

[3]If you are using a Unix-like system such as Ubuntu or Mac OS 10, you can 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 it 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.

[4]Objects must begin with an alphabetic character and cannot have spaces.

FIGURE 3.1
R Startup Console
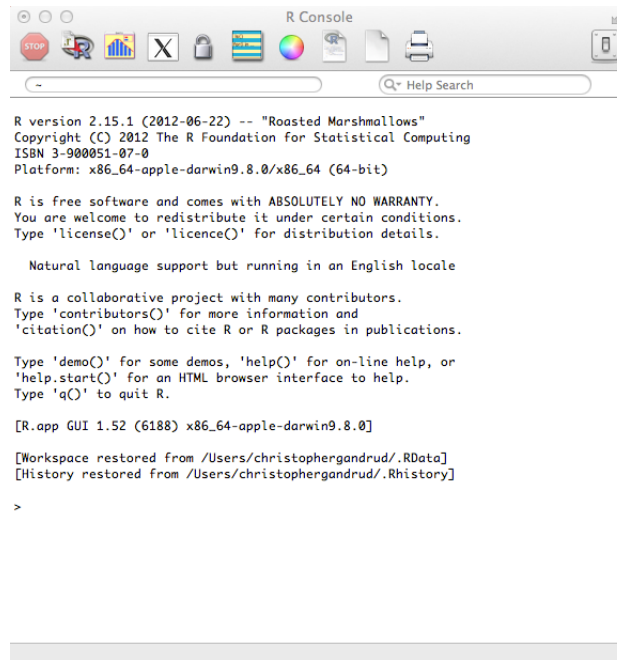

```
Number
```

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

Lets's briefly breakdown this output. 10 is clearly the contents of *Number*. The double hash (##) is included in the output by *knitr* to tell you that this is output rather than R code.[5] 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 we enclose the character string (letters in a word for example) inside of quotation marks (""). To create an object called *Words* that contains a character string "Hello World".

```
Words <- "Hello World"
```

An object's class is important to keep in mind as it determines what things

---

[5]This makes it easier to copy and past code included in a presentation document by *knitr*.

we can do to it. For example you cannot take the mean of a character mode
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` to mean
missing. To find out what class an object has use the `class` command. For
example:

```
class(Words)

## [1] "character"
```

Vector & data frame Objects

Sp far we have only looked at objects with a single number or string.[6] Clearly
we want to be using objects–data sets–that have many strings and numbers.
In R these are usually **data frame** type objects and are roughly equivalent to
the type of data structure you would be familiar with from using a program
such as Microsoft Excel. We will be using data frames extensively throughout
the book. It is also useful to cover other, simpler types of objects, primarily
vectors. Vectors are R's "workhorse".[5] Knowing how to use vectors will be
especially helpful when we clean up raw data in Chapter 7 and make tables
in Chapter **??**.[7]

Vectors:

Vectors are the "fundamental data type"[5] in R. They are simply an ordered
group of numbers, character strings, and so on.[8] It may be useful to think of
basically all other data types as complicated forms of vectors. For example,
data frames are basically multiple vectors of the same length–i.e. they have
the same number of rows–and possibly of different types attached together.

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

---

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

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

[8]In a vector every member of the group must be of the same type. Lists are similar to
vectors, but allow you to have different types.

```
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 *Character-Vect* into a new object with the `cbind` function. This function binds the two vectors together as if they were different columns.[9]

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

# Display contents of StringNumObject
StringNumObject

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

By binding these two objects together we've created a new matrix object.[10] You can see that the numbers in the *NumericVect* column are between quotation marks. Matrices, like vectors can only have one data type or mode.
Data frames: If we want to have an object with rows and columns and allow the columns to contain data with different modes, 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)
```

---

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

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

```
# 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. 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 completely 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 no longer are in quotation marks. This does not mean, however that they are no longer character string type data. To prove this try taking the mean of *CharacterVect* like this:

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

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

3.1.3   Subscripts

3.1.4   Functions, Commands, and Arguments

3.1.5   Loading new commands from libraries

You can install new add-on packages using the `install.packages` command. By default this command downloads and installs the packages from the Comprehensive R Archive Network (CRAN). For the code you will need to install all of the package libraries used in this book see .

Commands are stored in R libraries. R automatically loads a number of basic libraries by default. One of the great things about R is the many user-created libraries[11] that greatly expand the number of commands we can use. To load a library so that you can use its functions use the `library` command. Use the following code to load the *ggplot2* library that we use in Chapter 10 to create figures.

```
library(ggplot2)

## Use suppressPackageStartupMessages to eliminate package startup
messages.
```

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

3.2   Using RStudio

As I mentioned in Chapter 1 When you first open RStudio you should get a something that looks like Figure 3.2. In this figure you see three panels. The large one on the left is the *Console*. This pane functions exactly the same as the console in regular R.

Compile Notebook

RStudio integrates a number of different methods for literate programming. The most basic is

---

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

FIGURE 3.2
RStudio Startup Panel
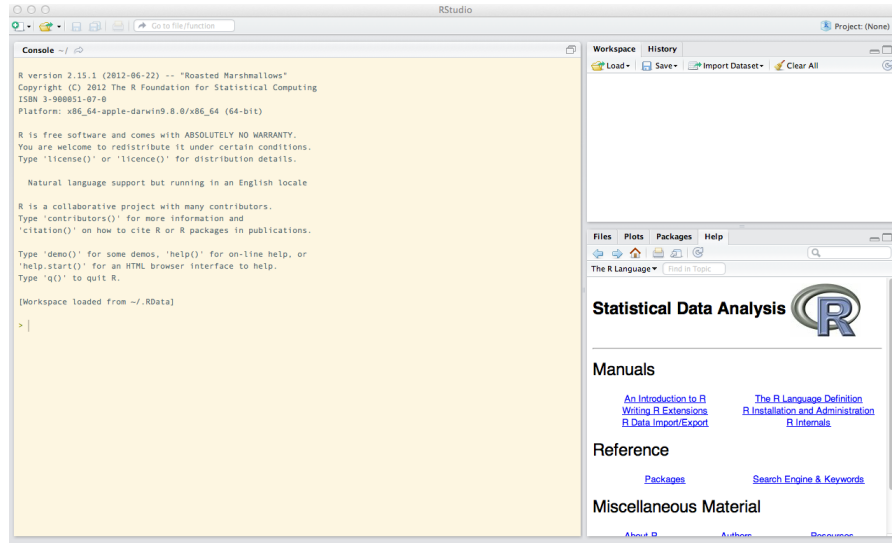
3.3 Using knitr: the basics

For more advanced literate programming in R we can use the *knitr* package.[12] This package is highly integrated into RStudio. Throughout much of the book we cover many details about how to use *knitr*. This section gives you a basic introduction to get started.

3.3.1 Code Chunks

When we want to include R code into our presentation documents we place them in a code chunk.

FILL IN

All labels and code chunk options must be on the same line.

Code chunk labels

Code chunk options

For the full list of *knitr* options see this page maintained by *knitr*'s creator Yihui Xie: `http://yihui.name/knitr/options#package_options`.

---

[12]The RStudio notebook actually also uses *knitr*.

3.3.2 Code Chunk Options Markup Language Specific Syntax

The code chunk options discussed so far are specifically for presentation documents written in Markdown syntax. Though the syntax for *knitr* options is the same regardless of the markup language we use

Code Chunks in Markdown

Code Chunks in LaTeX

LaTeXdocuments 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*.[4] More recent versions of `knitr` support files with the `.Rtex` extension, though the code chunk syntax is different. I will cover both types of syntax for code chunks in LaTeXdocuments, though in throughout the book I use the older and more established *Sweave* style syntax.

.Rnw-style

.Rtex-style

3.3.3 Global Options

So far we have only looked at how to set local options in *knitr* code chunks. If we want an option to apply to all of the chunks in our document we can set **global chunk options**. They 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. 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[13] You will probably want to set the option `echo=FALSE` so that *knitr* doesn't echo the code. Inside the code chunk use `opts_chunk$set`. You can set any chunk option as an argument to `opts_chunk$set`. The option will be applied across your document, unless you set a different local option.

Here is an example of how we could have all of the figures in a Markdown

---

[13]In Markdown, you can put global chunk options at the very top of the document. In LaTeXthey should be after the `\begin{document}` command (see Chapter **??** for more information on how LaTeXdocuments are structured).

document created by *knitr* code chunks center aligned. We place the following code at the beginning of the document:

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

3.3.4   knitr package options

Chunk options determine how we want to treat code chunks. We can also set package options that affect how the *knitr* package itself runs. For example, the `progress` option can be set as either `TRUE` or `FALSE`[14] depending on whether or not we want a progress bar to be displayed when we knit a code chunk.[15] 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).

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

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

3.3.5   knitr & RStudio

RStudio is highly integrated with *knitr*

Insert code chunks

Run code chunks

Change default .Rnw knitter

To use *knitr* instead of *Sweave* when knitting `.Rnw` files in RStudio you should go to the **Options** window. Click on the `Sweave` button. Select `knitr` from the drop down menu for "Weave files using:". Finally, click `Apply`. The **Options** window is in different places depending on the version of RStudio you are using:

---

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

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

- **Mac**: `RStudio → Preferences`,

- **Windows & Linux**: `Tools → Options`.

3.3.6 knitr & R

As an R package, you can of course knit documents in regular R (or using the console in RStudio). All of the syntax in the document you are knitting is the same as before. But instead of clicking a `Compile PDF` or `knit HTML` button we use is `knit` command. To knit our example Markdown file *Example.Rmd* we first set the working directory (see Chapter 4) to the the folder where your *Example.Rmd* file is located with the `setwd` command. In this example I have it on my desktop.[16]

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

Then I knit my file:

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

Note that if you do not specify the output file *knitr* will determine what the file should be. In this example it would come up with the same name and location.

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

```
mardownToHTML(file = "Example.md", 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 LaTeXand to create a PDF with a bibliography (see Chapter **??** for more details on using for bibliographies). Here is a `texi2pdf` example:

```
texi2pdf(file = "Example.tex")
```

---

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

3.4 R/RStudio Tips

Finally, here are a few other tips that make using R and RStudio a little easier.

Getting Help

You can quickly access the help file for any command. In R/the Rstudio console simply type `?` before any command. For example, to get the help file for the `mean` command just type:

```
?mean
```

In RStudio you can click on the **Help** pane (by default at the lower right, see Figure 3.2) and enter the command you want help with into the search field.

Autocomplete

In R and RStudio you do not have to type out every command, argument, object, or even directory name. After you start typing a command/argument/object/directory name you can hit the "tab" key to automatically complete the word you started. The function is particularly good in RStudio. Not only does it give you a list of words to choose from, but it also shows you an abbreviated version of the help file for commands and arguments.

4

Getting Started with File Management

CONTENTS

Careful file management is crucial for reproducible research. Apart from the fleeting situations where you have am email exchange (or even meet in person) someone interested in reproducing your research, the main information other researchers will have is stored across many files: data files, analysis files, and presentation files. If these files are well organized then replication will be easier. File management is also important for you as a researcher, because if your files are well organized you will be able to understand your work more easily. Remember two of the guidelines from Chapter 3:

- Reproducible research projects are many files explicitly tied together,

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

Using tools such as R and markup languages like LATEXrequires a bit more knowledge of how files are located–their **path**–in your computer and on the internet than just knowing which graphical folder they are located in. Though more difficult to use at first than the typical point-and-click graphical user interface file handling systems you are probably familiar, R and Unix-like shell programs allow us to control files–creating, deleting, moving them–in powerful and reproducible ways.

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 navigate through files in RStudio in the **Files** pane as well as some basic R and Unix-like shell commands for manipulating files. The skills we learn in this chapter will be heavily used in the next Chapter (Gathering Data with R) and throughout the book.

In this chapter we work with locally stored files, i.e. files stored on your computer. In the next chapter (Chapter 5) we will discuss various ways to store and access files stored remotely in the cloud.

4.1   Organizing your research project

Figure 4.1 gives an example of how the files in a simple reproducible research project could be structured. The project–called *Example Project*–is organized into three main parts: a data gathering section, an analysis section, and a presentation section. The results of the project are presented in an article, slideshow, and website.

FIGURE 4.1
Example Research Project File Tree

4.2   File paths & naming conventions

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

4.2.1   Root directories

4.2.2   Working directories

4.3   Operating system-specific naming conventions

Unix

Mac

Windows

4.4   File navigation in RStudio

The RStudio **Files** pane allows us to navigate and do some basic file manipulations. Figure 4.2 shows us what this pane looks like.

4.5   R file manipulation commands

All of the tasks we can accomplish in RStudio's **Files** pane can also be accomplished using command line R. This allows us to more easily replicate our actions.
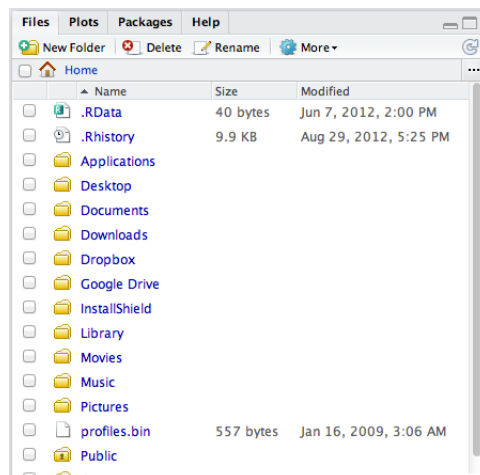
`setwd`

The `setwd` command sets the working directory.

`dir.create`

Sometimes we may want to create a new directory. We can use the `dir.create` command to do this.

FIGURE 4.2
The RStudio Files Pane

`unlink`

You can use the `unlink` command to delete a file, files, or directories.

4.6 Unix-like shell commands

Though this book is mostly focused on using R for reproducible research it can be useful to use a Unix-like shell program to manipulate files in large projects. A command line shell program is simply a program that allows you to type commands to interact with your computer's operating system.[8] We will especially return to shell commands near the end of the book when we discuss Make files for compiling large documents, and batch reports (Chapter 12). The syntax discussed here is also similar to the used in command line git (Chapter 5) and Pandoc (Chapter 12). We don't have enough space to properly get started with shell programs. For good introductions for Unix and Mac OS 10 computers see William E. Shotts Jr.'s book on the Linux command-line[8] and for Windows users Microsoft maintains a tutorial on Windows PowerShell at `http://technet.microsoft.com/en-us/library/hh848793`.

The one piece of general instruction I will give now is to highlight an important difference in the syntax between R and shell commands. In shell commands you don't need to put parentheses around your arguments. For

example if we want to change our working directory to my Mac Desktop in a shell using the `cd` command we simply type:
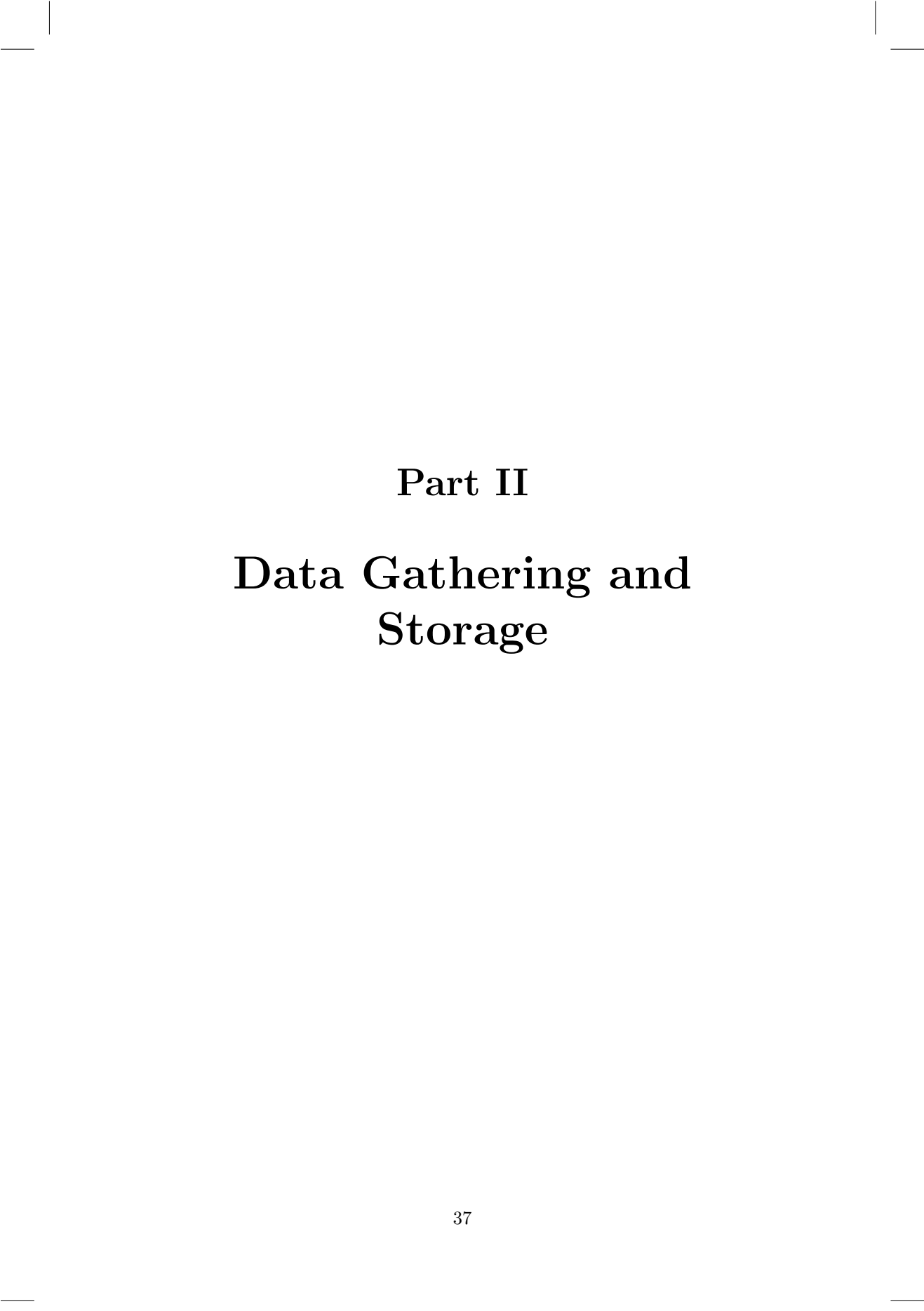
```
cd ~/Desktop
```

cd

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

rm

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

Part II

Data Gathering and Storage

5

Storing, Collaborating, Accessing Files, Versioning

CONTENTS

A stumbling block to actually reproducing a piece of research is getting a hold of the datasets and the codebooks that describe the data used in an analysis.

Researchers often face a number of data management issues that, beyond making their research difficult to reproduce, can make doing the initial research difficult.

First, there is the problem of **storing** the data so that it is protected against computer failure–virus infections, spilling coffee on your laptop, and so on.

Fourth, we almost never create a data set or write a paper perfectly all at once. We may make changes and then realize that we liked an earlier version, or parts of an earlier version better. This is a particularly important issue in data management where we may transform our data in unintended ways and want to go back to an earlier version. Collaborative projects can have regular incidents of one author accidentally deleting something in a file that another author needed, for example.

To deal with these issues we need to store our data in a system that has **version control**. Version control systems keep track of changes we make to our files and allow us to access previous versions if we like.

the data set can often grow and become disorganized. Perhaps even during a data transformation This creates problems

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 data:

- storage,

- accessing,

- collaboration,

- version control.

5.1 Saving data in reproducible formats

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

R is able to read (and write) a very wide variety of file formats, mostly through the `foreign` package in `base` R. This includes

5.2 Storing data in the cloud

Storing data locally–on your computer–or on a flash drive is generally more prone to loss than storing data on remote servers, often referred to as 'the cloud'.

5.3 Dropbox

The easiest types of cloud storage for your research are services like Dropbox and Google Drive. These services typically involve a folder based on your computer's hard drive that is automatically synced with a similar folder on a cloud-based server. Typically you can sign up for the service for free and receive a limited amount of storage space (usually a few gigabytes, which should be plenty if your research is made up of text files.).

Most of these services not only store your data in the cloud, but also provide some way to share files and maybe even includes basic version control. I am going to focus on using Dropbox because it currently offers a complete set of features that allow you to store, version, collaborate, and access your data.

5.3.1 Version control

Dropbox has a simple version control system. Every time you save a document on Dropbox a new version is created. One the Dropbox website

5.3.2 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 access from 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. The key factor is whether or not the files are located in your **Dropbox**'s *Public* folder. Files in the *Public* folder can be downloaded directly into R. Files not in the *Public* folder have to be downloaded manually.[1]

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

If the file is not in your *Public* folder you also go to Dropbox after right-clicking. 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. When you hover your curser over a file (or folder) name you will see a chain icon appear on the far right. Clicking on this icon will get you the link.

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

Note that you cannot save files through the URL link. You must save files in the Dropbox folder on your computer.

---

[1]This is not completely true. It is possible to create a web scraper (see Chapter GET) that could download a data file from a file not in your *Public* folder. However, this is kind of a hassle and not practical, especially since the accessing files from the *Public* folder is so easy.

5.4   GitHub

Dropbox does a fine job of meeting our four basic criteria for reproducible data storage. GitHub meets these criteria and more.

GitHub was not explicitly designed to host research projects or even data. It was designed to host 'socially coded' computer programs. It built an interface on top of the git version control system that makes it easy relatively easy for a number of collaborators to work together to build a computer program. This seems very far from reproducible research.

However, remember that as reproducible researchers we are just building projects out of interconnected text files. This is exactly the same as computer programming. and like computer programers, 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.

As is usually the case, GitHub's added features mean that it is takes a longer time than Dropbox to set up and become familiar. So we need good reasons to want to invest the time needed to learn GitHub rather than just sticking with Dropbox or a similar service. Here is a list of GitHub's key features relative to Dropbox for reproducible research:

- Git is directly integrated into RStudio projects (**RStudio** also supports the subversion version control system, but I don't cover that here).

- Dropbox's version control system only lets you the see the file names, the times they were created, who created them, and revert back to specific versions. git tracks every change you make in a way that makes it relatively easy to find the version you want. The GitHub website and GUI programs for Mac and Windows provide nice interfaces for examining specific changes. You can also use the command line to see changes.

- Dropbox creates a new version every time you save a file, which can make it difficult to actually find the version you want. git's version control system only creates a new version when you tell it to.

- Dropbox does not merge conflicting versions of a file together. This can be annoying when you are collaborating on project and more than one author is making changes to documents. GitHub identifies conflicts and lets you reconcile them.

- The GitHub website as an ''Issues" area where you can to note and discuss issues you have while doing your research. Basically this is an interactive to-do list for your research project.

5.4.1 Setting Up GitHub

There are a number of ways to set up GitHub on your computer. I will briefly cover both the command line version (available for Windows, Mac, and Linux) and the GUI[2] version currently available only for Windows and Mac.

5.4.2 Version Control in GitHub

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

Reverting to an old version of a file

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

```
git checkout COMMITREF FILENAME
```

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

Let's break down the code. `FILENAME` is the name of the file that you want to change[3] and `COMMITREF` is the reference that git gave to the commit you want to revert back to. The reference is easy to find and copy in GitHub. On the file's GitHub page click on the `History` button. This will show you all of the commits. By clicking on `Browse Code` you can see what the file at that commit looks like. Above this button is another with a series of numbers and letters. This is the commit's SHA (Secure Hash Algorithm). For our purposes, it is the commit's reference number. Click on the `Copy SHA` button to the left of the SHA to copy it. You can then paste it as an argument to your `git checkout` command.

More Practice with Command Line GitHub

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

---

[2]Graphical User Interface, i.e. not the command line version, but the one with windows that you navigate with your mouse.

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

6

Gathering Data with R

CONTENTS

There are many practical issues involved in gathering data that can make replication easier or harder. As with all of the steps in this book: document everything. Replication will be easier if your documentation–source code–can be understood and executed by a computer. Of course there are data gathering situations that simply require manually pointing and clicking, talking with subjects in an experiment, and so on. The best we can do in these situations is just describe our data gathering process in detail CITE. Nonetheless, R's automated data gathering capabilities are extensive and often under utilized. Learning how to take full advantage of them greatly increases replicability and can even save researchers considerable time and effort.

6.1   Organize Your Data Gathering: Make files

Before getting into the details of using R to automate data gathering, lets's start from where all data gathering should start: a plan to organize the process. Clearly organizing your data gathering process from the start of a research project improves the possibility of replicability and can save significant effort over the course of the project.

    A key principle of replicable data gathering with R, like replicable research in general is segmenting the process into discrete files that can be run by a

common Make file. The Make file's output is the data set(s) that we use in the statistical analyses. There are two types of files that the Make file runs: data clean up files and merging files. Data clean up files bring raw (the rawer the better) individual data sources into R and transform them into something that can be merge with data from the other sources. Some of the R tools for data clean up were covered in Chapter 3. In this chapter we mostly cover the ways to bring raw data into R. We don't explicitly cover the process of merging data sets together in this book GET CITE. Merging files are executed by the Make file after it runs the clean up files.

Data gathering Make files 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 some file.[1] If we plan to gather data from two different data sources–DATA1 and DATA2–stored in the directory DIRECTORY our Make file might look like this:

```
# Example Make file
setwd("~/DIRECTORY/")
# Clean up raw data files.
source("CleanDATA1.R")
source("CleanDATA2.R")

# Merge cleaned data files
source("MergeDATA1.DATA2.R)
```

You can save the output data set using the `write.table` command placed in the merge file or the Make file.

---

[1] The `source` command is used more in the Chapter 8.

6.2 Importing locally stored data sets

6.2.1 Single files

6.2.2 Looping through multiple files

6.3 Importing data sets from the internet

6.3.1 Data from non-secure (`http`) URLs

6.3.2 Data from secure (`https`) URLs

6.3.3 Compressed data stored online

Sometimes data files can be very large, making them difficult to store and download without compressing them. There are a number of compression methods such as Zip and tar archives. Zip files have the extension `.zip` and tar archives use extensions such as `.tar` and `.gz`. In most cases[2] we can easily download, decompress, and create dataframe objects from these files directly in **R**.

To do this we need to:[3]

- create a temporary file with `tempfile` to store the zipped file which we will 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**,[4]

- 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 more than one file as well as metadata.

Let's download a compressed file called *uds_summary.csv* from [7]. It is in a zipped file called *uds_summary.csv.gz*. The file's URL address is `http://www.unified-democracy-scores.org/files/uds_summary.csv.gz`, but I shortened it[5] to `http://bit.ly/S0vxk2` to cut down on the text I have to include in the code.

---

[2]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 we create for unzipped files.

[3]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`, accessed 16 July 2012.

[4]To find a full list of commands type `?connections` in to the **R** console.

[5]I used the website `bitly.com` to shorten the URL.

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

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

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

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

# Delete the temporary file.
unlink(temp)
```

6.3.4 Data APIs & feeds

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

• The *openair* package, which beyond providing a number of tools for analysing air quality data also has the ability to directly gather data directly from sources such as Kings College London's London Air (`http://www.londonair.org.uk/`) database with the `importKCL` command.

6.4 Basic web scraping

6.4.1 Scraping tables

6.4.2 Gathering and parsing text

7

Preparing Data for Analysis

CONTENTS

Once we have gathered the raw data that we want to include in our statistical analyses we generally need to clean it and merge it into a single data files CAWELY QOUTE ABOUT HOW IT CAN BE BAD TO USE DATA FROM DIFFERENT DATA FRAMES.This chapter covers some of the basics of how to clean data files and merge them using R.

The two main suggestions for cleaning and merging data are to:

- always versions of the original–non-cleaned–data in as raw a state as possible,

- again document everything.

It's a good idea to keep data your original data in as raw a version as possible because it makes reconstructing the steps you took to create your data set easier. Also, while cleaning and merging your data you may transform it in an unintended way, for example, accidentally deleting some observations that you had intended to keep. Having the raw data makes it easy to go back and correct your mistake. Documenting everything also helps you achieve these two goals. Also it makes updating the data set easier if, for example, new data becomes available. MAYBE EXPLAIN MORE.

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

7.1   Cleaning data for merging

7.2   Sorting data

7.3   Merging data sets

7.4   Subsetting data

Part III

Analysis and Results

8

Statistical Modelling and knitr

CONTENTS

8.1   Incorporating analyses into the markup

8.1.1   Full code in the main document

8.1.1.1   LaTeX

8.1.1.2   Markdown

8.1.2   Showing code & results inline

Sometimes we want to have some R code or output to show up in the text of our documents. We may want to include stylized code in our text when we discuss how we did an analysis. We may want to report the mean of some variable in our text.

8.1.2.1   LaTeX

Inline static code

If we just want to include a code snippet in out text we can simply use the LaTeXcommand \tt. This sets our text to 'typewriter' font, the standard font for inline code in LaTeX(I use it in this book, as you have probably noticed).

Inline dynamic code

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

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

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

## [1] 591
```

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

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

This will produce 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 document we enclose the code in single backticks (``). For example typing ` MeanRiver <- mean(rivers) ` produces MeanRiver <- mean(rivers).

Inline dynamic code

To include dynamic code in an R Markdown document we use the backticks as be fore but include a the letter r after the first one.

8.1.3 Sourcing R code from another file

There are a number of reasons that you might want to have your R source code located in a separate file from your markup even if you plan to 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 collapsing and chunk management options. There are just too many things going on in one document.

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

Third, other researchers trying to replicate your work might only be interested in specific parts of your analysis. If you have the analysis broken into separate and clearly labeled files it is easier for these researchers to find the specific bits of code that they are interested compared to digging through long markup files.

8.1.3.1 Source from a local file

Usually in the early stages of research you may want to source analysis iles located on your computer. Doing this is simple. The *knitr* syntax is the same as above. The only change is that instead of writing all of our code in the chunk we save it to its own file and use the `source` command in *base* R to access it. For example:

8.1.3.2 Source from a non-secure URL (`http`)

Sourcing from your local 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 replicate your code, you need to use another method.[1]

The simplest solution to these issues is to host the replication code in your Dropbox public folder. You can find the file's public URL the same way we did in Chapter 6. Now use the `source` command the same way as before. For example:

---

[1]You can 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. However, this usually just adds an extra complicating step that makes replication harder. It is also a pain if you are collaborating and each author has to constantly change the directories.
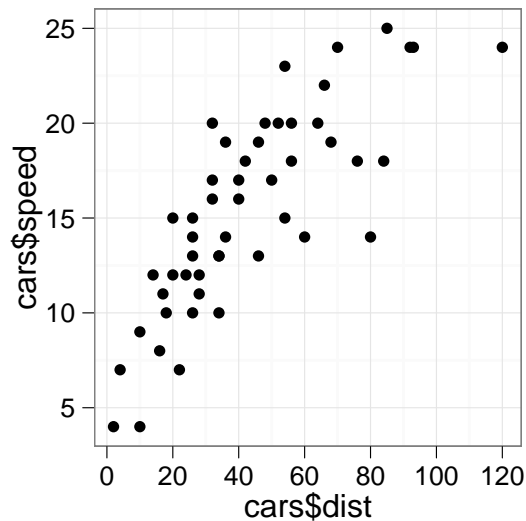
8.1.3.3   Source from a secure URL (`https`)

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

```
# Load library
library(devtools)

# Create object to hold URL
URL <- "http://bit.ly/Ny1n6b"

# Run the source code to create the scatter plot
source_url(URL)
```


───────────────
[2]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.

8.2 Saving output objects for future use

8.3 Literate Programming: Including highlighted syntax in the output

8.3.1 LaTeX

8.3.2 Markdown/HTML

8.4 Debugging

9

Showing Results with Tables

CONTENTS

??

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

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

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

Warning: Automating table creation removes the possibility of adding errors to our analyses by incorrectly copying R output, which is a big potential problem in hand-created tables. Be warned, it is not an error free process. We

---

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

could easily create inaccurate tables through coding errors. For example, we may incorrectly merge together columns in so that our id variables no longer match the data they are supposed to.

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

9.1   Table Basics

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

9.1.1   Tables in LaTeX

9.1.2   Tables in Markdown/HTML

9.2   Creating tables from R objects

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

9.2.1.1   `xtable` for LaTeX

9.2.1.2   `xtable` for Markdown

9.2.2   `xtable` with non-supported class objects

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

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

1. find and extract the information from the unsupported class object that we want in the table,

---

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

2. convert this information into a matrix or dataframe where the rows and columns of the object correspond to the rows and columns of the table that we want,

3. use `xtable` with this object to create the table.

Imagine that we want to create a results table showing the covariate names, coefficient means, and quantiles for marginal posterior distributions from a Bayesian normal linear regression using the `zelig` command [2] and data from the *swiss* dataframe.[3] We run our model:

```
# Load required library
library(Zelig)

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

# Find NBModel's class
class(NBModel)

## [1] "MCMCZelig"
```

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

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

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

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

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

We created a new object of the class `summary.MCMCZelig`. We're still not there yet as this object contains not just the covariate names and so on but also information we don't want to include in our results table like the formula that we used. The second step is to extract a matrix from inside *NBModelSum* called *summary* with the component selector (`$`). This matrix is where the things we want in our table are located. I find it easier to work with dataframes, so we'll also convert the matrix into a dataframe.

---

[3]This dataframe is loaded by default.

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

Here is what our model results dataframe looks like:

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

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

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

TABLE 9.1
Coefficient Estimates Predicting Examination Scores in Swiss Cantons (1888)
Found Using Bayesian Normal Linear Regression

It may take a bit of hunting to find what you want, but a similar process can be used to create tables from objects of virtually any class.[4] Hunting for what you want is generally easier by clicking on the object in RStudio's workspace pane.

9.2.3 Basic `knitr` syntax for tables

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

The most important `knitr` chunk option for showing the markup created by these packages as tables is `results`. The `results` option can have three values:

- `markup`,

- `asis`,

- `hide`.

`hide` clearly hides the results of whatever we have in our code chunk; no results show up.

---

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

9.3   Tables with `apsrtable`

10

Showing Results with Figures

CONTENTS

10.1   Basic knitr figure options

10.2   Creating figures with plot and ggplot2

10.3   Animations

10.4   Motion charts and basic maps with GoogleVis

Part IV

Presentation Documents

11

Presenting with LaTeX

CONTENTS

??

11.1 The Basics

All commands in LaTeXstart with a \

11.1.1 Editors

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

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

11.1.2 The header & the body

All LaTeXdocuments require a header. The header goes before the body of the document and specifies what type of presentation document you are creating– an article, a book, a slideshow, and so on. LaTeXrefers to these as classes.

You also can specify what style it should be formatted in and load any extra packages you may want to use to help you format your document.[1]

The header is followed by the body of your document. You tell LaTeX where the body of your document starts by typing `\begin{document}`. The very last line of 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 header and body like this:

```
\documentclass{article}
\begin{document}
\end{document}
```

11.1.3   Headings

11.1.4   Footnotes & Bibliographies

11.1.4.1   Footnotes

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

11.1.4.2   Bibliographies

Citing R Packages with BibTeX

Researchers are pretty good about consistently citing others' articles and data. However, citing the R packages used in an analysis is very inconsistent. This is unfortunate not only because correct attribution is not being given but also because it makes reproducibility harder because it obscures important steps that were taken in the research process. Fortunately, R actually includes the tools to quickly generate citations, including the version of the package you are using. It can also add them directly to an existing bibliography file.

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

```
citation("ggplot2")

##
## To cite ggplot2 in publications, please use:
```

---

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

```
##
##   H. Wickham. ggplot2: elegant graphics for data
##   analysis. Springer New York, 2009.
##
## A BibTeX entry for LaTeX users is
##
##   @Book{,
##     author = {Hadley Wickham},
##     title = {ggplot2: elegant graphics for data analysis},
##     publisher = {Springer New York},
##     year = {2009},
##     isbn = {978-0-387-98140-6},
##     url = {http://had.co.nz/ggplot2/book},
##   }
```

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

```
toBibtex(citation("ggplot2"))

## @Book{,
##   author = {Hadley Wickham},
##   title = {ggplot2: elegant graphics for data analysis},
##   publisher = {Springer New York},
##   year = {2009},
##   isbn = {978-0-387-98140-6},
##   url = {http://had.co.nz/ggplot2/book},
## }
```

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

```
# Divert output to biblio.bib
sink(file = "biblio.bib",
     append = TRUE, type = c("output")
     )
toBibtex(citation("ggplot2"))
sink()
```

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

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

```
# Load package
library(knitcitations)

# Write Zelig citation and
# to biblio.bib
write.bibtex(entry = c("ggplot2"),
             file = "bibliography.bib", append = TRUE)
```

11.2   Presentations with Beamer

You can make slideshow presentations with LaTeX. FILL IN WITH INTRO

knitr largely the works the same way in in LaTeXslideshows as it does in article or book class documents. There are a few differences to look out for.

Slide frames

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

```
\frame{
}
```

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

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

Results

By default *knitr* hides the results or a code chunk. 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 LaTeXdocuments see this blog post by Pieter Belmans: `http://pbelmans.wordpress.com/2011/02/20/why-latex-beamer-needs-fragile-when-using-verbatim/`.

12

Large LATEXDocuments: Theses, Books, & Batch Reports

CONTENTS

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

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

12.1 Planning large documents

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

12.1.1 Planning theses and books

Books and theses have a natural parent-child structure, i.e. they are single documents comprise of multiple chapters. They often include other child-like features such as title pages, bibliographies, figures, and appendices. We could

include most of these features directly into one markup file. Clearly this file would become very large and unwieldy. It would be difficult to find one part or section to edit. If of your presentation markup are are difficult to find, they are difficult to reproduce.

12.1.2   Planning batch reports

COMPLETE

12.2   Combining Chapters

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

12.2.1   Parent documents

knitr global options

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

12.2.2   Child documents

Child documents in the same markup language

Child documents in a different markup language

You can use the command line program and 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 with R's `system` command.

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

---

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

into a LATEXfile. This file can then be input into my main document with the
LATEXcommand
`input`.

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

```
system("pandoc StylisticConventions.md
        -f markdown -t latex -o StyleTemp.tex")
```

The options `-f markdown` and `-t latex` tell Pandoc to convert *Stylistic-
Conventions.md* from Markdown to LATEXsyntax. `-o StyleTemp.tex` in-
structs Pandoc to save the resulting LATEXmarkup to a new file called
StyleTemp.tex.

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

CONTENTS

13.1 The Basics

13.1.1 Headings

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

13.1.2 Footnotes and bibliographies with MultiMarkdown

13.1.3 Math

13.1.4 Drawing figures with CSS

13.2 Simple webpages

13.2.1 RPubs

13.2.2 Hosting webpages with Dropbox

13.3 Presentations with `Slidify`

13.4 Reproducible websites

13.4.1 Blogging with Tumblr

13.4.2 Jekyll-Bootstrap and GitHub

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

13.4.3 Jekyll and Github Pages

13.5 Using Markdown for non-HTML output with Pandoc

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

Pandoc can help solve this problem. Pandoc is a command line program that can convert files written in Markdown, HTML, LaTeX, and a number of other markup languages[1] to any of the other formats.

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

For example,

---

[1]See the Pandoc website for more details: `http://johnmacfarlane.net/pandoc/`

14

Chapter 14:

CONTENTS

Bibliography

[1] Jake Bowers. Six steps to a better relationship with your future self. *Newsletter of the Political Methodology Section, APSA*, 18(2):2–8, 2011.

[2] Ben Goodrich and Ying Lu. *normal.bayes: Bayesian Normal Linear Regression*. 2007.

[3] Donald E. Knuth. *Literate Programming. Center for the Study of Language and Information–Lecture Notes*. Stanford, 1992.

[4] Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9.

[5] Norman Matloff. *The Art of Programming in R: A Tour of Statistical Programming Design*. No Starch Press, San Francisco, 2011.

[6] Jill P. Mesirov. Accessible Reproducible Research. *Science*, 327(5964):415–416, 2010.

[7] Daniel Pemstein, Stephen A. Meserve, and James Melton. Democratic compromise: A latent variable analysis of ten measures of regime type. *Political Analysis*, 18(4):426–449, 2010.

[8] William E Shotts Jr. *The Linux Command Line: A Complete Introduction*. No Starch Press, San Francisco, 2012.

[9] Yihui Xie. *knitr: A general-purpose package for dynamic report generation in R*, 2012. R package version 0.7.9.

Index