An R Companion to POLITICAL ANALYSIS

SECOND EDITION

Philip H. Pollock III
Barry C. Edwards

An R Companion to Political Analysis

Second Edition

Sara Miller McCune founded SAGE Publishing in 1965 to support the dissemination of usable knowledge and educate a global community. SAGE publishes more than 1000 journals and over 800 new books each year, spanning a wide range of subject areas. Our growing selection of library products includes archives, data, case studies and video. SAGE remains majority owned by our founder and after her lifetime will become owned by a charitable trust that secures the company's continued independence.

Los Angeles | London | New Delhi | Singapore | Washington DC | Melbourne

An R Companion to Political Analysis

Second Edition

Philip H. Pollock III

University of Central Florida

Barry C. Edwards

University of Central Florida

SAGE | CQPRESS

FOR INFORMATION:

CQ Press
An imprint of SAGE Publications, Inc.
2455 Teller Road
Thousand Oaks, California 91320
E-mail: order@sagepub.com

SAGE Publications Ltd.
1 Oliver's Yard
55 City Road
London EC1Y 1SP
United Kingdom

SAGE Publications India Pvt. Ltd.
B 1/I 1 Mohan Cooperative Industrial Area
Mathura Road, New Delhi 110 044
India

SAGE Publications Asia-Pacific Pte. Ltd.
3 Church Street
#10-04 Samsung Hub
Singapore 049483

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data

Names: Pollock, Philip H., III., author. | Edwards, Barry C., author.

Title: An R companion to political analysis / Philip H. Pollock III, University of Central Florida, Barry C. Edwards, University of Central Florida.

Description: Second edition. | Thousand Oaks, California : CQ Press, [2018] | Includes bibliographical references and index.

Identifiers: LCCN 2017003186 | ISBN 9781506368849 (pbk. : alk. paper)

Subjects: LCSH: Political statistics—Computer programs—Handbooks, manuals, etc. | Analysis of variance—Computer programs—Handbooks, manuals, etc. | R (Computer program language)—Handbooks, manuals, etc.

Classification: LCC JA86 .P639 2017 | DDC 320.0285/5133—dc23
LC record available at https://lccn.loc.gov/2017003186

Acquisitions Editor: Carrie Brandon
Development Editor: Anna Villarruel
Editorial Assistant: Duncan Marchbank
eLearning Editor: John Scappini
Production Editor: Kelly DeRosa
Copy Editor: Christina West
Typesetter: C&M Digitals (P) Ltd.
Proofreader: Sarah J. Duffy
Cover Designer: Anupama Krishnan
Marketing Manager: Amy Whitaker

This book is printed on acid-free paper.

17 18 19 20 21 10 9 8 7 6 5 4 3 2 1

Contents

List of Boxes and Figures

Boxes

Figures

Preface

In many ways, the second edition of *An R Companion to Political Analysis* follows the template of the book that preceded it. Thus, this volume guides students in the use of R for constructing meaningful descriptions of variables and performing substantive analysis of political relationships, from bivariate cross-tabulation analysis to logistic regression. As before, all of the examples and exercises use research-quality data—including two survey datasets (the 2012 American National Election Study and the 2012 General Social Survey) and two aggregate-level datasets (one based on the 50 U.S. states and one based on countries of the world). And, as in the first edition, each chapter is written as a tutorial, taking students through a series of guided examples that they then use to perform the analysis.

The second edition improves upon the first in three ways. First, we have added an "Introduction to R" to familiarize students with the R environment and help them understand the logic of objects and functions. Second, we have repurposed Chapter 9 to emphasize how R's plotting functions can be used to show the results of regression analysis.

The third and most important change from the first edition is the development and release of an R package called "poliscidata" that bundles the functions and datasets used in this book. Students can now simply install this book's R package, load it in R, and then jump right into executing commands and analyzing results. The book's R package is freely available on the Comprehensive R Archive Network (CRAN). The installation process is detailed in Chapter 1.

Each chapter has been revised to reflect the updated datasets that accompany this book. Where possible, we've revised our examples and model solutions to offer students simpler, more intuitive approaches. Throughout the text, we emphasize simple solutions that accommodate missing data and allow the research to apply sampling weights. We've also made a special effort to show how to use R to create publication-level tables and figures. Data visualization is an especially exciting field and a relative strength of R. Because most students are visual learners, giving them the opportunity to see relationships in data and statistical concepts in action is also a great teaching tool.

We have updated the end-of-chapter exercises for the second edition of this book. In our exercises, we attempt to test students' understanding of the methods demonstrated in each chapter.

Students can log on to **edge.sagepub.com/pollock** to access datasets used in *An R Companion to Political Analysis* as well as tables and figures from the book to strengthen understanding of key terms and concepts.

ADVICE FOR INSTRUCTORS

This book is intended to help college students learn to apply political science research methods using the R program. We emphasize developing good writing habits, proper interpretation of statistics, and clear presentation of results. This book isn't a comprehensive reference to R's data analysis functions. This book is

intended to serve as a companion to textbooks that emphasize the general concepts of political science research. We hope this book helps your students use the R program to apply textbook and lecture concepts to solve problems and conduct research.

Those of us who teach political science research methods understand there are pros and cons to using different statistics programs. We think instructors should be aware of the advantages and disadvantages of using R and, if they choose R, work to maximize its advantage and minimize its potential problems.

The primary benefit of using R for teaching students to use political science research methods is that R is a free program that works well on both Windows and Mac OS platforms. Students don't have to work on certain computers on campus or under an expiring software license. In our experience teaching this class, students really like the convenience of being able to work on their own laptops, even though we have computer labs on campus. Working with R gives students the option of working on or off campus, at times that fit in their schedules. Although R is sometimes seen as a program reserved for hardcore quants, it may be more appropriate to view R as a program made for everybody. We think it's great that students can build a toolkit of R scripts over the course of a term and take it with them into other classes or the workplace. The only real limitation to using R is the willingness to learn how.

In this book, we try to identify the fundamental research methods used by political scientists and demonstrate the simplest ways of applying these methods using R. In a number of instances, we've written very simple functions to execute certain tasks with minimal coding. Of course, we recognize that there are many different ways to implement research methods in R. We think it makes sense to teach students how R functions are called, demonstrate the simplest possible solution to a problem, and encourage students to demonstrate their creativity and initiative by refining the basic solution or trying other solutions to the problem. If you've mastered different solutions to some of the problems we discuss in this book, we'd encourage you to teach R strategies that are familiar to you in place of, or as alternatives to, the strategies we demonstrate here.

As noted in the preface to the first edition, teaching students to use R presents a number of challenges. Students are used to using computer software for everyday tasks and entertainment. Chances are, they've never had to use an instruction manual to operate a computer or electronic device, so using a manual to operate a statistics program is an unfamiliar task. Our suggestion is to be frank with students about the pros and cons of R and explain why you're using it to teach research methods. We've found that many students (although often reluctant to admit it) actually enjoy the challenge of learning a new skill that demands precision and attention to detail. When students learn that R is widely used in the private sector and familiarity with R is a desirable skill to potential employers, they are likely to prefer using R to working with other statistics programs.

One specific suggestion we'd like to offer instructors who plan on using R to teach political science research methods is to consider devoting at least part of one class session to helping students get R and the R package that bundles the functions and datasets used in this book installed and operational. Encourage students to bring their personal laptops to this session to get them set up to work independently. If you've worked with R for a while, it's easy to forget how confusing the R environment appears to a new user. Help students get to the point where they can execute commands and observe R's response. Make sure your students are prepared to start making mistakes and learning from them; trial and error is essential, so you don't want students to get caught up on one-time, set-up issues.

If you think that learning how to use R is a learning objective in and of itself and not merely a means to other ends, consider incorporating some computer lab sessions into your course if time and facilities allow you to do so. One of us (Edwards) teaches research methods with equal parts lecture and lab sessions. In the lab sessions, students work on solving problem sets using R for statistical analysis. When students have questions, they raise their hands and receive one-on-one instruction. Edwards has been fortunate to work with some excellent graduate teaching assistants who join the class lab sessions to work one-on-one with students. He has also recruited top students to return to lab sessions in subsequent terms to help other students learn to use the R program. It's a lot of fun and the hands-on experience with R reinforces the general concepts from lectures and the textbook.

ACCOMPANYING CORE TEXT

Instructors will find that this book makes an effective supplement to any of a variety of methods textbooks. However, it is a particularly suitable companion to Pollock's own core text, *The Essentials of Political Analysis*, now in its fifth edition. The textbook's substantive chapters cover basic and intermediate methodological issues

and ideas: measurement, explanations and hypotheses, univariate statistics and bivariate analysis, controlled relationships, sampling and inference, statistical significance, correlation and linear regression, and logistic regression. Each chapter also includes end-of-chapter exercises. Students can read the textbook chapters, do the exercises, and then work through the guided examples and exercises in *An R Companion to Political Analysis*. The idea is to get students to experience political research firsthand, early in the academic term. An instructor's solutions manual, free to adopters, provides solutions for all the textbook and workbook exercises.

ACKNOWLEDGMENTS

We would like to thank the wonderful editorial team at SAGE Publications for their continued support and encouragement. It's a real pleasure to work with such a talented and professional group. We would also like to thank the R Development Core Team and the authors whose functions we use throughout this book.

Pollock would like to thank his co-author, Barry Edwards, who took on the lion's share of the revisions and who is responsible for making this edition much better than the last. Pollock would also like to extend a special thanks to Charisse Kiino, who saw to it that the first edition of this project got off the ground.

Edwards would like to thank, first and foremost, his co-author, Philip Pollock, for inviting him to help revise the first edition of this book. Pollock is a terrific co-author and has a great sense of humor. Edwards would also like to thank the graduate and undergraduate teaching assistants he's had the pleasure of teaching POS 3703 with: Christine Regnier-Bachand, David Shabat-Love, Jason Christensen, Preeti Prakash, Sydney Dotson, Ryan Allen, Marissa Hall, Bobby Sells, and Jessica Lago. He also thanks all of the University of Central Florida students who have endured all his corny jokes in Howard Phillips Hall and inspired him to strive for better and better ways to teach this material.

A Quick Reference Guide to R Companion Functions

FUNCTION ARGUMENTS

Symbol	*What It Means*
x	Variable; independent variable
y	Dependent variable
z	Control variable
w	Optional weight variable
dataset	Dataset (gss, nes, states, or world)
design.dataset	Dataset created with svydesign (gssD, nesD, statesD, or worldD)

FUNCTION USAGE

AdjR2*(tdf=total.df, null.dev=null.deviance, resid.dev=residual.deviane, k=#indepvars)*
CI95*(m=mean, se=standard.error)*; CI99 *(m=mean, se=standard.error)*
Colors()
compmeans*(x=y, f=x, w=w, plot=T/F …)*
CramersV*(chi=chi2.statistic, r=#rows, c=#columns, n=sample.size)*
crosstab*(dep=x, indep=y, weight=w)*
csv.get*("csv.dataset.csv")* [import data in .csv format)
cut2*(x=variable, cuts=cutpoints, m=min.obs, g=num.groups …)* [use cuts or g]
ddply*(.data, .variables, .function ...)* [see help(ddply) for special input format]
describe*(x=variable, weights=w ...)*
fit.svyglm(*svyglm=svyglm.model*)

(Continued)

(Continued)

freq*(x=variable, w=w, plot=T/F ...)*; freqC *(x=variable, w=w)*
imeansC*(function1=~y, function2=~x + z, data=design.dataset)*
lineType()
logregR2*(model=logit.model ...)*
orci*(model= logit.model ...)*
pchisqC*(reduced=reduced.logit.model, full=full.logit.model ...)*
plotChar()
plotmeans*(formula=y ~ x, data=dataset ...)*
plotmeansC*(data=dataset, formula2=~y, formula3=~x, formula4=y~x, w=~w ...)*
printC*(objx=table.output)*
prop.testC*(y=y, x=x, w=w)*
scatterplot*(formula=y~x, data=dataset ...)*
somersD*(formula~x+y=, data=design.dataset)*
sortC*(data=dataset, id=identifier/name, by=sort.criteria, descending=T/F)*
spss.get*("SPSS.dataset.sav")* [import SPSS dataset]
stata.get*("Stata.dataset.dta')* [import Stata dataset]
svyboxplot*(formula=y~x, design=design.dataset ...)*
svyby *(formula=~y, by=~x, design=design.dataset, FUN=function.applied ...)*
svychisq *(formula=y~x, design=design.dataset ...)*
svychisqC *(formula=y~x, design=design.dataset)*
svydesign(id=~1, data=*data,* weights=*~w ...)* [create *design.dataset]*
svyglm*(formula=binary.y ~ x l... xn, design=design.dataset,* family=quasibinomial)
svyglm*(formula=y ~ xl ... xn, design=desig n.dataset ...*)
svytable*(formula=y~x, design=design.dataset)*
welcome()
wtd.boxplot*(formula=y ~ x, weights=w ...)*
wtd.chi.sq*(var1=x, var2=y, weight=w ...)*
wtd.cor*(x=variable.matrix, weight=w ...*)
wtd.hist*(x=variable, weight=w ...)*
wtd.mean*(x=variable, weights=w ...)*
wtd.median*(x=variable, weights=w)*
wtd.mode*(x=variable, weights=w)*

wtd.quantile*(x=variable, weights=w ...)*
wtd.sd*(x=variable, weights=w)*
wtd.t.test*(x=variable, y=test.value, weight=w ...)* [One sample t-test]
wtd.t.test*(x=var1, y=var2, weight=w1, weighty=w2 ...)* [Two sample t-test]
wtd.ttestC*(f1=~y, f2=~x, data=design.dataset)*
wtd.var*(x=variable, weights=w)*
xtabC*(function1=y~x, data=dataset)*
xtp*(data=dataset, y=y, x=x , w=w ...)*
xtp.chi2*(data=dataset, y=y, x=x , w=w ...)*

For more detailed help files on these functions, enter **?** followed by the function's name or help(function_name) in R. Functions from base installation packages are not listed.

Introduction: Getting Acquainted with R

Objective	*Functions Introduced*	*Author or Source*
Demonstrating R capabilities	c {base} data.frame {base} seq {base} sqrt {base} mean {base} help {utils}	All functions by R Development Core Team[1]

As you have learned about political research and explored techniques of political analysis, you have studied many examples of other people's work. You may have read textbook chapters that present frequency distributions, or you may have pondered research articles that use cross-tabulation, correlation, or regression analysis to investigate interesting relationships between variables. As valuable as these learning experiences are, they can be enhanced greatly by performing political analysis firsthand—handling and modifying social science datasets, learning to use data analysis software, learning to describe variables, setting up the appropriate analysis for interesting relationships, and running the analysis and interpreting your results.

This book will guide you as you learn these practical and creative skills. Using R, powerful data analysis software, to analyze research-ready datasets, you will learn to obtain and interpret descriptive statistics (Chapter 2), to collapse and combine variables (Chapter 3), to perform cross-tabulation and mean analysis (Chapter 4), and to control for other factors that might be affecting your results (Chapter 5). Techniques of statistical inference (Chapters 6 and 7) are covered too. On the somewhat more advanced side, this book introduces correlation and linear regression (Chapter 8). You will learn how to create graphics that show relationships among variables and the results of regression analysis (Chapter 9). Chapter 10 provides an introduction to logistic regression, an analytic technique that has gained wide currency in recent years. Chapter 11 shows you how to code your own data, and it provides guidance on writing up your results. Virtually every chapter in this book places special emphasis on the graphic display of data, an area of increasing interest to the scholarly community.

---

[1] R Development Core Team. (2011). *R: A language and environment for statistical computing*. Vienna, Austria: Author. Available at http://www.R-project.org/

To get started with this book, you will need access to a computer with an Internet connection. After you set up your computer with the right software and add-ons, you'll be able to work offline. All of the necessary files are freely accessible on the Internet.

ABOUT R

What is R? R is free software developed in the public domain to analyze data. You can run R on a variety of operating systems. The base version of R performs many of the statistical procedures you will learn in this book. In addition, hundreds of users have written a large number of specialized programs for R, all of which are available from the Comprehensive R Archive Network (CRAN), a clearinghouse for R resources of all kinds.[2]

In the world of multi-faceted computer software, R is something of a youthful upstart—version 1.0.0 was released in early 2000—but its user base has steadily expanded.[3] Indeed, by 2014, R had an estimated 2 million regular users. Large corporations, such as Google and Facebook, use R for special applications, such as data visualization.

Powerful, flexible, richly supported, increasingly popular—and free. What's the downside? This: R is hard. The learning curve is steep. The R interface can be described as either retro or primitive, depending on how charitable you wish to be. Although a handful of promising graphical user interfaces (GUIs) for R exist, R's core power is unlocked by the keyboard, not the mouse. (Yes, R is command line.) Because different programmers have contributed to R's development, not all commands adhere to the same syntactical rules. Until you get the hang of it, you will find yourself frequently referring to the reference card provided with this book. Above all—and subsuming all these challenges—R's approach to computing, its *idea* of computing, is almost certainly different from the approach you have grown accustomed to. The R statistical environment takes some getting used to. However, when you get comfortable working with objects and using functions, you'll appreciate the program's flexibility and the wealth of tools available for data analysis.

INSTALLING R

There is no substitute for practical experience with R. Let's install R so we can begin seeing how R thinks and behaves.

To install R, follow these steps, illustrated in Figures I.1–I.5:

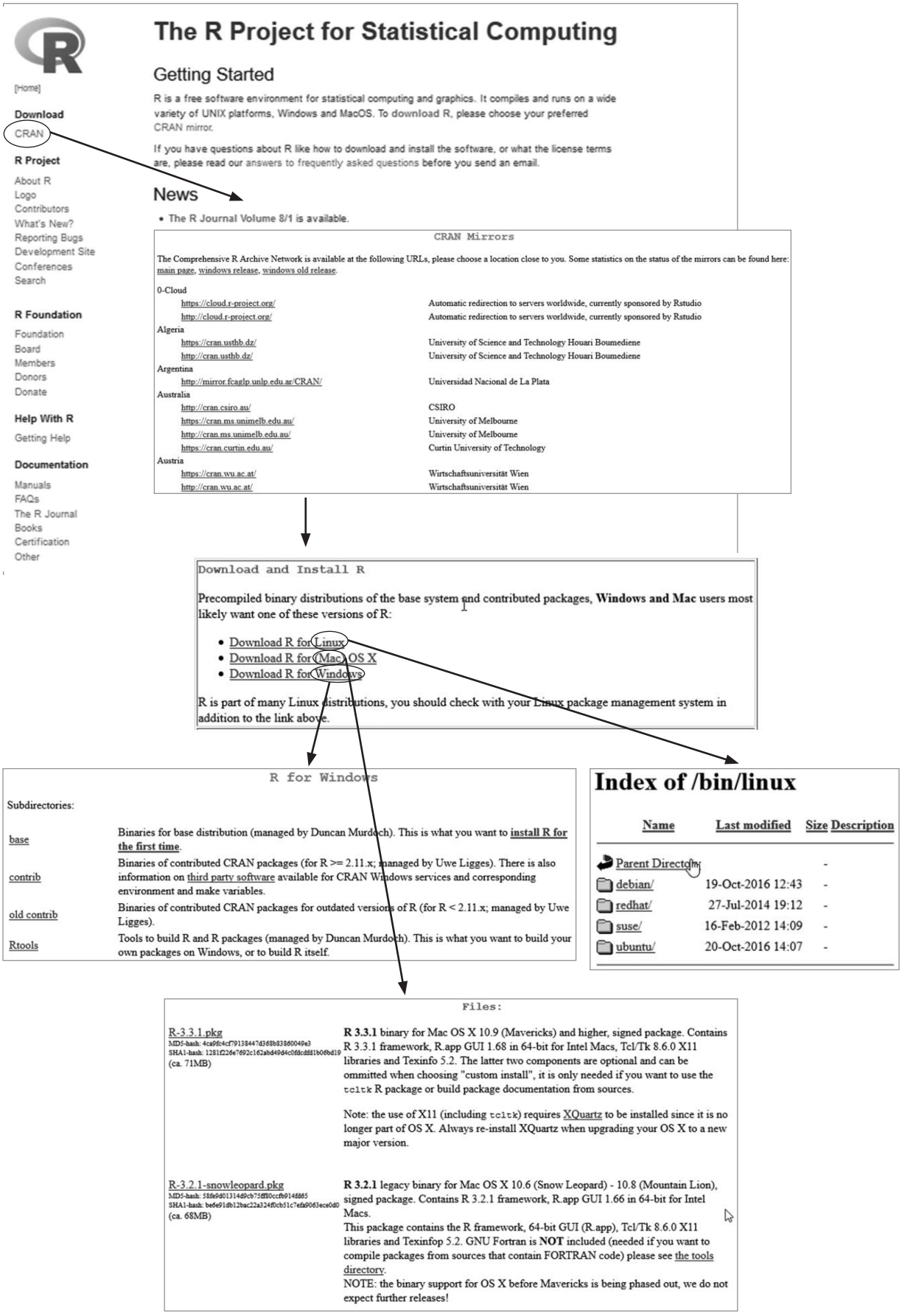
1. Open http://cran.r-project.org/, the home page of the R Project for Statistical Computing.
2. Under the "Download" heading on the left side of the home page, click the link for "CRAN" (the Comprehensive R Archive Network).
3. Select a repository near you from the list. The 0-Cloud options at the top of the list offer automatic redirection to servers worldwide, so they make a good default choice.
4. Under the heading "Download and Install R", select the link that corresponds to your computer's operating system.
    - *For Windows*: Click "base" or "install R for the first time" to install the basic version of the most recent version of R. *Note to Windows users*: The Windows installer should determine whether to install the 32-bit or the 64-bit version of R. However, if you need to determine your machine's bit count, find help here: http://support.microsoft.com/kb/827218.
    - *For Mac*: Select the most recent version of the R program your operating system can support. As of the time of this writing, the most recent version of R (3.3.1) requires Mac OS X 10.9 or higher. If your Mac OS is older than that, select the R version appropriate for your system.
    - *For Linux*: Follow instructions specific to your Linux distributor.
5. Follow normal installation procedures. Click through the installation dialogues. Accept the default settings.

---

[2] See http://cran.r-project.org/

[3] Ashlee Vance, "Data Analysts Captivated by R's Power," *New York Times*, January 6, 2009.

Figures I.1–I.5 R Project for Statistical Computing Home Page, Location of Repositories, Windows Download Options, Mac OS Download Options, Linux Download Options

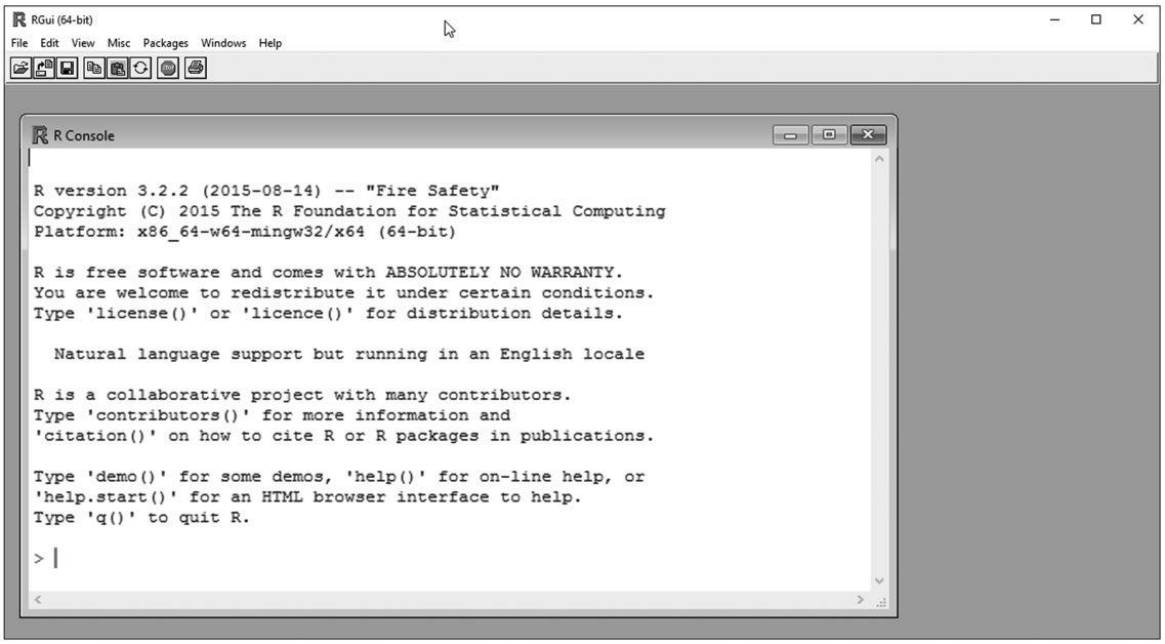


A QUICK TOUR OF THE R ENVIRONMENT

Before we start entering commands, let's take a look around the R program. Double-click the R icon. The window that opens on the left side of screen is called the R Console. Above the R Console, at the top of the screen, you'll see a row of drop-down menus. You can edit some settings to customize your R environment, but the drop-down menus are pretty spare. If you're running R on Mac OS or Linux, your R environment may look different than how it's depicted in Figure 1.6. (You have some options to customize the look and feel of your R environment with the "GUI preferences . . . " option under the Edit menu tab.)

Notice the > sign on the last line of the R Console in Figure 1.6? R is awaiting your commands.

Figure I.6 R Console



Now that you've got R running and know where you can enter commands, let's see what R can do. It can be helpful to think of R as an overgrown programmable calculator. Like a calculator, if you ask R to calculate a number like "2 + 2", it will return the answer, 4, to you.

```
2 + 2                                  # Enter "2+2" and R returns "4"
```

```
[1] 4
```

Notice that R's response to the command "2 + 2" starts with the [1]. Rather than clear your command, R *indexes* its answer, "[1]", and returns it on the next line. In this case, the answer is just one number, but we'll soon see that R can work with long series of numbers, in which case indexing helps us make sense of results.

In our simple 2 + 2 example, we see an example of an *operator* used by the R program. The + sign is a mathematical operator that adds numbers together. As you might guess, R also uses the familiar mathematical operators: – (dash, to subtract), / (forward slash, to divide), * (asterisk, to multiply), and ^ (caret, to raise to a power). The equal sign (=) is particularly important in the R environment and we will focus on it in the next section.

Comparing R to a calculator helps us get started, but it only scratches the surface of what R can do. To start unlocking R's potential, we need to learn about *objects* and *functions*.

To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.

— John Chambers (a co-creator of R)

OBJECTS

Objects are used to store information in an accessible manner. Just as all things are nouns in the English language, all things are objects in R. Some objects encapsulate just one value; other objects store vast arrays of data. Objects in R store different things, but they all are equally accessible in the workspace, ready to be put to use—no opening, entering, creating, saving, or exiting required. You can think of objects as all-purpose containers for information, much like the contacts list in your phone. You could key in a friend's number every time you want to call them, but it's easier to retrieve their phone number by associating it with their name. The contact object may have several attributes, such as an e-mail address, mailing address, and photo. Similarly, in R you can create objects and assign numbers and text—even other objects—to the object. We use the *assignment operator* to assign values to objects. The equal sign (=) is the intuitive choice, although R traditionalists prefer the classic assignment operator (<-), which does provide some advantages when it comes to writing functions.[4]

```
phoneNumber = 4078232608              # Assigns number to "phoneNumber"
phoneNumber <- 4078232608             # Alternate assignment operator
```

Object names must be one word, no spaces. If you were to insert a space in the middle of an object name, R would think you are referring to two different objects. There are a few limits on the names you can give objects. Names cannot begin with a digit or with a period followed by a digit. Some objects are already defined by the system, so you should avoid using them: T, F, and pi are examples. (For a complete list, type "?Reserved" at the R prompt.) As a general rule, avoid using single-character object names because they are not clear, descriptive names for the values of their contents. There is no character limit on object names, so you don't need to sacrifice clarity to save computer memory.

Just as you assigned a number to the object phoneNumber, you may also want the value of an object to be a word or phrase. Use quotation marks to assign text as a value to an object, otherwise R thinks you are referring to an object or doing math. (If we wanted to store the phone number in the above example with formatting, that is, as 407-823-2608, we would need to put the value in quotation marks.)

```
name = "UCF Poli Sci Dept"          # Assigns text in quotes to "name"
```

Successful object assignments work quietly in R. If you make an error, such as forgetting the closing quotation mark, R will let you know. If you type in the name of the object, R will return its assigned value.

Objects are the building blocks of the R environment. Two or more objects can be combined to produce a more complex and information-rich object. Suppose we wanted to create a new object, "directory", that combines the two objects created previously, "name" and "phoneNumber". The following assignment would do the trick. (This assignment statement uses a function, data.frame. We will take a closer look at functions in the next section.)

```
directory = data.frame(name, phoneNumber)    # Object from objects
```

Now enter "directory" to retrieve the contents of the combined object:

```
> directory
               name phoneNumber
1 UCF Poli Sci Dept  4078232608
```

[4] See http://www.r-bloggers.com/assignment-operators-in-r-%E2%80%98%E2%80%99-vs-%E2%80%98-%E2%80%99/

The new object is a *data frame*, which stores data in two dimensions: rows and columns. To be sure, our "directory" object is pretty sparse; it is a dataset with only one row and two columns. (In the next section, we will look at how to add names and phone numbers.) Even so, our tiny directory can illustrate how to use *brackets* to access values stored in objects. To access parts of the directory, you specify the row, column, or both the row and column, or you use the "$" sign to specify a variable in a dataset.

```
directory[1, ]              # returns first row
directory[, 1]              # returns first column
directory[, 2]              # returns second column
directory[1, 1]             # returns value of first row, first column
directory[1, 2]             # returns value of first row, second column
directory$name              # returns value of "name" variable
directory$phoneNumber       # returns value of "phoneNumber" variable
```

Note especially the role played by the dollar sign symbol, "$". This symbol tells R exactly where to locate an attribute stored in an object. Thus, the statement "directory$name" means, "Look in the object named 'directory' and output the attribute 'name'." The "$" symbol is important syntax, as we will see in Chapter 2, when we start working with variables.

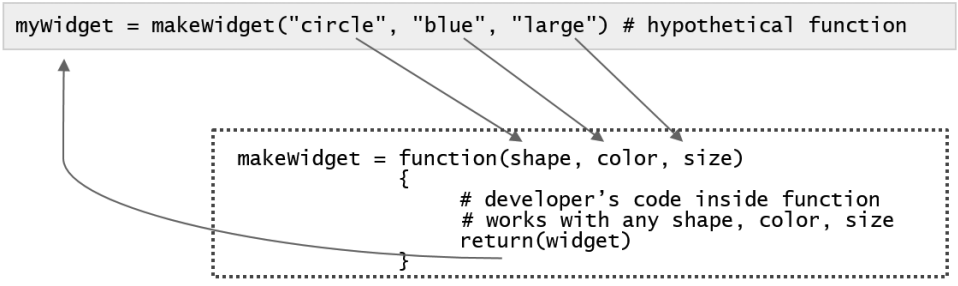
R recognizes six types of objects. For the purposes of this book, the most important are data frames (such as "directory") and *vectors*, which are strings of numbers (1, 2, 7, -4), words ("one", "two", "seven", "negative four"), or logical operators (TRUE, TRUE, TRUE, FALSE).

FUNCTIONS

Functions perform a defined sequence of actions. If objects are the equivalent of nouns, then functions are the equivalent of verbs. Functions are *generic*, meant to be used in a wide range of similar, but not identical, tasks. Functions are called by name, followed by a set of parentheses. A name without parentheses is an object. Good developers give their functions descriptive names. Function names must be one word, no spaces. If you insert a space in the middle of a function name, R will think you're referring to an object and a function and this will cause an error.

Some functions create objects, others don't. Some functions take input from the user. When you call a function, you may specify the values of the function's arguments, separating each argument with a comma, inside the parentheses that follow the function name. The argument values you specify in your function call are passed to the function and affect what the function does. As an R user, you should think about functions in terms of what you can input and what functions will output when you call them. To illustrate how you interact with functions, consider a hypothetical R function, makeWidget. The function allows you to specify the shape, color, and size of the widget you want made. Someday you may want to discover exactly how widgets are made but, for now, ignore the processes inside the box we put around the makeWidget function. What's important is the function's output, a widget made to your specifications, and to which we assign the name "myWidget".

Figure I.7 Hypothetical "makeWidget" Function



The author of a function defines how it is to be used and how any user-supplied information is processed within the function. The pieces of information that the user supplies the function, like the shape, color, and size of the widget to make, are called *arguments*. Some arguments are required, others are optional. You could imagine, for example, additional options to customize your widget beyond the standard features. A good developer would write a function that easily creates simple widgets with sensible default options, but also gives users access to advanced settings to customize their widgets. For the purposes of this book, you need not be too concerned about the code that is inside the R functions included in the base installation of the program or in the packages you install to expand R's functionality. Instead, you should focus on the flow of information between the function call and the function.

In the preceding section, we used the data.frame() function to create a data frame object, "directory", from two other objects, "phoneNumber" and "name", each of which contain only one value. Now we will take a look at another useful function, concatenate. It has a simple name: c. Suppose you wanted to create an object with several phone numbers. You could use the concatenate function to create a vector, which is an object with multiple values. The following example uses concatenate to produce two vectors, "name" and "phoneNumber". The two vectors are then combined to update the "directory" data frame.[5]

```
name        = c("UCF Poli Sci", "HPH lab", "Orlando") #name has three values
phoneNumber = c(4078232608, 3215555252, 2025678901)   #phoneNumber's 3 values
directory   = data.frame(name, phoneNumber)           #updates data frame
directory                                             #outputs data frame
```

```
         name phoneNumber
1 UCF Poli Sci  4078232608
2      HPH lab  3215555252
3      Orlando  2025678901
```

Another function, seq(), provides an opportunity to learn about function arguments. This is a very useful function, but we must supply it with a few vital pieces of information. Take a look at this function's usage statement:

```
seq(from, to, by)
```

The seq function creates a sequence of numbers, provided that you supply it with the start-from number, the go-to number, and the count-by increment. Supply that information inside the parentheses that follow the name of the function. Notice that, because it allows the user to set the parameters, the seq function becomes more versatile. That is, we don't need one function to count up, another to count down, another to count by twos, and so on. The seq function will perform any of those actions. The following code creates a vector object, "vec1", that ranges from 1 to 49 in increments of 3:

```
vec1 = seq(from = 1, to = 49, by = 3)   # using function arguments by name
vec1 = seq(1, 49, 3)                    # using function arguments by position
vec1
```

```
[1] 1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
```

[5] If we had more phone numbers than names (or vice versa), we could not store these vectors together in a data frame; instead, we would need to use another function, such as list (), that works with vectors of different lengths. If you want to create large data frames or lists with many values, you don't want to create these objects and assign them values in an R script; instead, you'd want to create a spreadsheet-like file for your data and read that file as an object (loading external data files is discussed in Chapter 11).

Notice that the first two statements above produce the same result. If we do not specify the names of function arguments, R will use positional matching and assume that the first value in parentheses corresponds to the first argument in the definition of the function, the second value in parentheses to the second argument in the defined usage of the function, and so on. (Consult the help page for a function to see how the function is defined and what its required and optional arguments are.) Now study these next lines of code and try to predict what they'll do when you enter them.

```
seq(3, 50, 1)                # what sequence will this generate?
seq(by=3, to=50, from=1)     # will this generate same output?
seq(10, 2, from=1)           # what's wrong with this command?
```

It is important to understand how functions are defined because one of the most common mistakes is to not supply function inputs in the right order or with the correct syntax. Remember, R is open source software with many contributors; there is no single, centralized authority to enforce uniform practices, so you will see different expressions of the same idea across packages and functions. Because R is open source, many people write functions for the program, which helps explain its rapid growth and incredible versatility. However, R's radically decentralized development also means that authors are not required to adhere to consistent function definitions.

When you are working with simple functions—functions with only a few essential arguments—positional matching is usually fine. For more complex functions, you might want to use keyword matching. Our first use of the seq() function used keyword matching because we used the arguments named in the definition of the function. Throughout this book, when we discuss a particular function, we'll show you how to call the function correctly.

Sometimes the output of one function is the input to another function. When you nest one function inside another, pay particular attention to your use of parentheses. In the first example below, we use R's sqrt() function to compute the square roots of a sequence of numbers created by the seq() function. The second example uses the mean() function to calculate the mean of the same sequence.

```
sqrt(seq(1, 50, 3))          # nested function outputs a vector
mean(seq(1, 50, 3))          # nested function outputs a single number
```

Notice what happens when you input these commands. One returns a new series of numbers, the other just one number. Why? R is calculating the square root of each number in the sequence. The mean() function, by definition, calculates one number from a set of numbers. If we wanted to calculate the square root of the sum, or square root of the mean, we'd use parentheses to establish the order of operations.

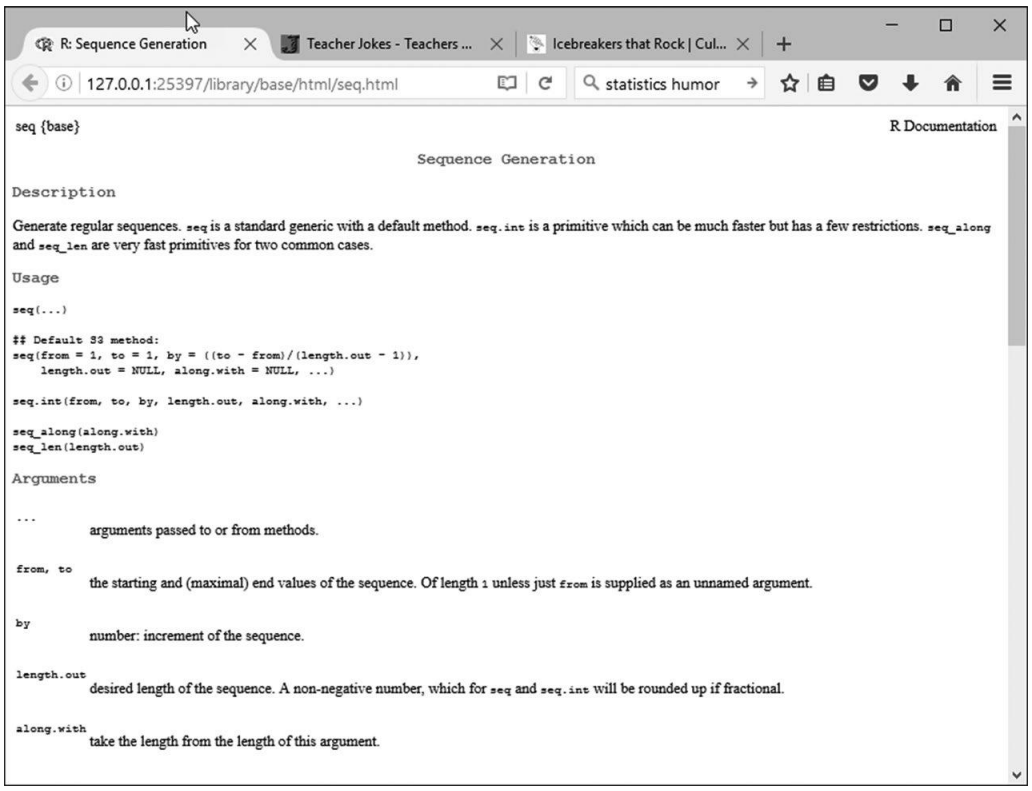
Don't think of functions as formulas you should memorize. There are far more functions written for R than you could possibly memorize. (At the time of this writing, there are nearly 10,000 packages written for R containing approximately 200,000 different functions.) It's much more important to understand the general logic of functions. While functions do many different things, you use them the same way. You execute a function by its name followed by a set of parentheses. Inside the parentheses, you may specify the values of arguments used by the function. The information you specify in parentheses is supplied to the function and processed within the function, and the result of the operation is returned to you. When you're working with a new function, try executing it in the simplest manner possible before fine-tuning your function usage by setting optional arguments.

GETTING HELP

To obtain information on a function from a package that is installed and loaded, type '?*function.name*' or help(function.name) at the prompt.[6] For example, if you want to know more about the seq function, type: '?seq' or 'help(seq)'. Because the base package is loaded, R will show us the R documentation for the seq function:

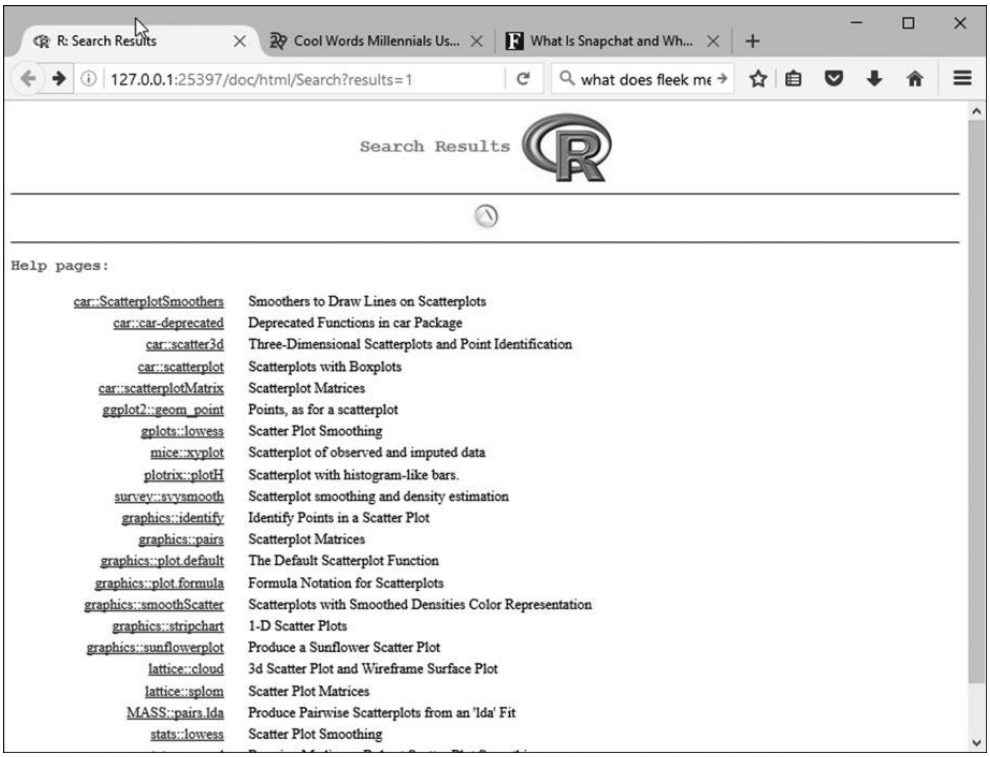
[6] For an alphabetical list of R packages, see https://cran.r-project.org/web/packages/available_packages_by_name.html

Figure I.8 Function Help File



Double question marks, '??*function. name*', extend the scope of the inquiry to R documentation that includes your search term. Below, we show the results of entering "??scatterplot" to learn more about R's impressive graphics capabilities.

Figure I.9 Extended Search Results



R documentation is highly technical and, truth be told, is not always helpful for beginners. Even so, if you are working with a new function, the usage section of the help file will show you the arguments you can include inside the parentheses. It may seem like a long list, but many of a function's arguments are optional. If you are relying on positional matching, make sure you put the arguments in the expected order. If you're using keyword matching, make sure you have the correct argument names. The arguments section of a function help file can tell you whether you have argument values in the right format; for example, you might need to set the value of an argument to a number rather than quoted text.

One of the best things about R is its enthusiastic online community. There are excellent resources available to help you learn about R. A particularly accessible source is Quick-R, http://www.statmethods.net/, created by Robert I. Kabacoff. With Quick-R, you can learn about the methods introduced in this book in greater detail as well as methods beyond the scope of this book. There are also some excellent video tutorials available. On YouTube.com, you can find concise, well-produced R tutorial videos from Phil Chan, Mike Marin (Marin Stats Lectures), and Lynda.com. For an in-depth treatment, the entire series of lectures from Emory University Professor Courtney Brown's "Statistics With R" course is available online. When you encounter problems, there's a great chance that someone has encountered the same problem and has published a helpful solution already.

EXERCISES

1. Which of the following are advantages of using the R statistical program? (Check all that apply.)

   - ❑ Free to use
   - ❑ Produces high-quality graphics
   - ❑ Thousands of user-contributed packages extend functionality
   - ❑ Live tech support available from the R Corporation

2. Create another object called "myName" and use an assignment operator to assign your name to this object. Be sure to use quotation marks around your name. Create an object called "myAge" and assign it your age in years. Next, apply the data.frame function to these objects to start a data frame object called "quickBio". Have R display the contents of your quickBio by entering "quickBio" in the R Console and copy the output here:

   _____

   _____

3. Consider the following R Commands:

```
thisNumber = 8
anotherNumber = thisNumber / 4 ^ 3 * 2
nextNumber = sqrt(anotherNumber)
theAnswer = nextNumber + thisNumber
```

   A. What is the value of "theAnswer"? (Circle one.)

   16 8.5

   8.25 12

   B. Which of the following objects has the largest value? (Circle one.)

   thisNumber anotherNumber

   nextNumber theAnswer

C. Which of the following objects has the smallest value? (Circle one.)

thisNumber anotherNumber

nextNumber theAnswer

4. Consider the following R Commands:

```
seq1 = seq(from=1, to=10, by=1)

theSolution = max(seq1) - length(seq1[1:5])
```

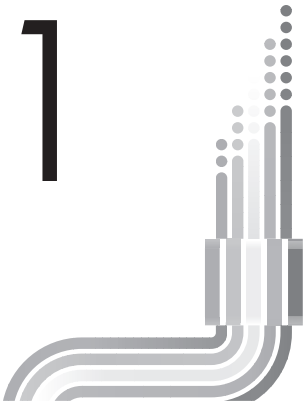
What is the value of "theSolution"? (Circle one.)

1 5

10 0

5. Everything that exists in the R environment is an object. Everything that happens is a function call. (Circle one.)

True False



1 The R Companion Package

Objective	*Functions Introduced*	*Author or Source*
Installing and loading the poliscidata package	install.packages {utils} library {base} welcome {poliscidata}	R Development Core Team R Development Core Team Philip Pollock and Barry Edwards
Exploring package contents	ls {base}	R Development Core Team
Demonstrating R capabilities	freq {descr} printC {poliscidata} getwd {base}	Jakson Aquino[1] Philip Pollock and Barry Edwards R Development Core Team

In the preceding Introduction, you became acquainted with some R basics: objects, functions, vectors, and data frames. For this book, we have developed a more specialized collection of additional objects and functions that will permit you to analyze, present, and interpret data. A specialized collection of R elements is called a *package*. The package we have created for this book, "poliscidata", contains the functions and datasets we use in this book and is available through an online repository. In this section, you will (1) run the install.packages function to install the poliscidata package, (2) run the library function to load the poliscidata package contents, and (3) run the poliscidata package's welcome function to produce some basic information about your working environment.

To install the poliscidata package, enter the following command:

```
install.packages("poliscidata")      # install bundled datasets and
                                     # functions for R companion
```

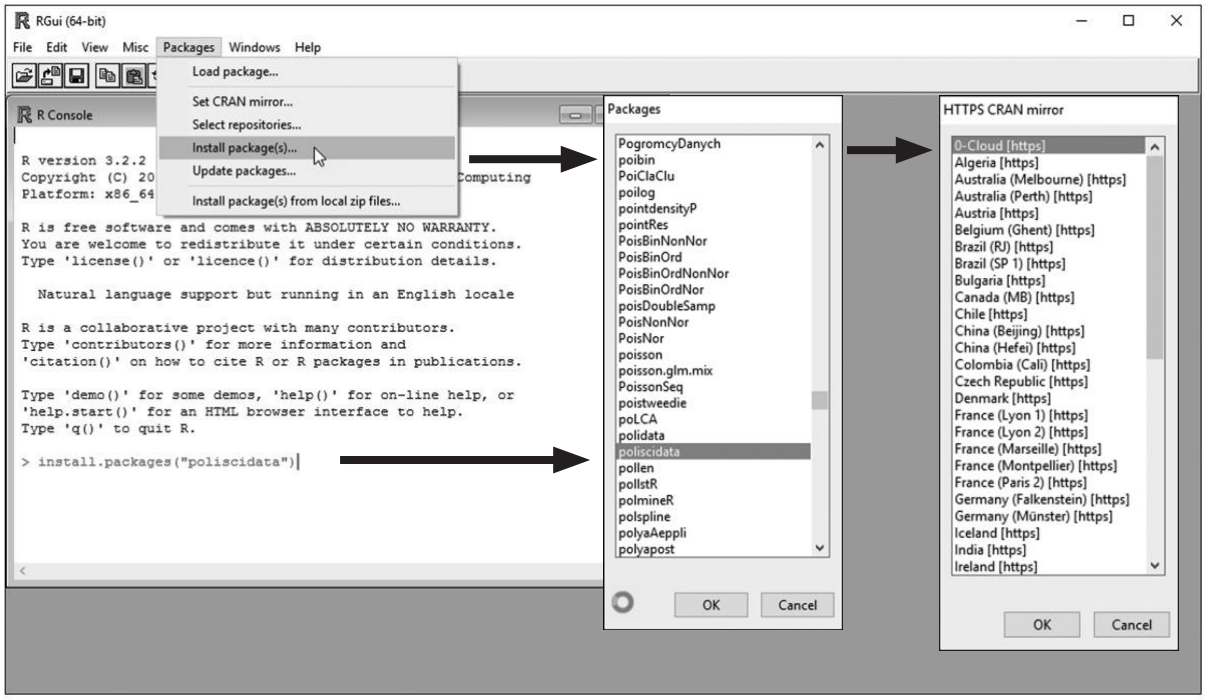
This command will prompt you to select a repository from which to download the poliscidata package. The repositories mirror one another, but you may want to select a repository close to you to save download time. See Figure 1.1.

You can also download an R package by selecting the "Install package(s) . . . " option under the Packages menu. This method will also prompt you to select a nearby repository to download the package from, and then select the "poliscidata" package from the long alphabetical list of R packages.

You will need to install the poliscidata package on each computer you use. When you install the poliscidata package, R will automatically install the functions and datasets you'll be using as well as the packages that the poliscidata package requires. The installation process may take a couple of minutes.

[1] Aquino, J. (2012). *descr: Descriptive statistics* (R package version 0.9.8). Includes R source code and/or documentation written by Dirk Enzmann, Marc Schwartz, and Nitin Jain. Available at http://CRAN.R-project.org/package=descr

Figure 1.1 Installing R Companion's Package



You might wonder why the R program does not come with all the packages you need. There are thousands of different packages available to extend R's capabilities. Chances are, you will use only a fraction of them (even if you become a lifetime R user). So the R Project keeps the base version of the program relatively light and allows users to add on functionality based on their individual preferences.

Box 1.1 Missing Packages

When you install the poliscidata package, R should automatically install all the packages that our package depends on. We've found, however, that R sometimes fails to install all the required dependencies. If this happens, you will see an error message that you are missing a required package. Don't panic. You can fix this problem pretty easily. You just need to install the missing packages manually. You can either select the "Install package(s) . . . " option from the Packages drop-down menu, select a repository near you, and select the missing package from the very long list of available packages, or you can type:

```
install.packages("name_of_missing_package")
```

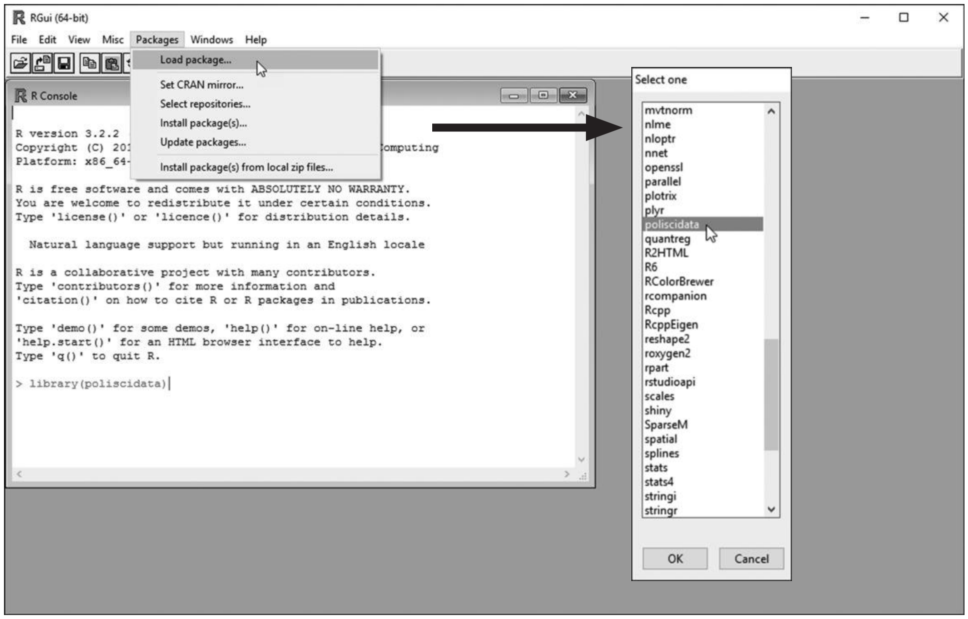
on the Console command line, substituting the name of the missing package where indicated. If R reports that it is missing another package, keep installing missing packages until the missing package error messages go away. You will not have to do all this each time you use R. It is simply a set-up issue.

Now that you've downloaded the poliscidata package, you need to load the package in your current R session using the library command. When you download R packages, they aren't automatically available every time you use R. The program allows you to selectively load installed packages so you can make efficient use of your computer's memory. It might be helpful to think about R packages likes apps you download to your phone; your phone doesn't come with all available apps pre-installed: It lets you pick and choose which ones you want. After you've downloaded an app, you have to open it to use it; you don't want all your phone apps to open and run automatically.

After you've installed the poliscidata package, you load it with the library command. You can also load the poliscidata package by selecting the "Load package . . . " option under the Packages menu. See Figure 1.2.

```
library(poliscidata)            # Loads R companion package in session
```

Figure 1.2 Loading R Companion's Package



When you execute the library(poliscidata) command, it may look like R didn't do anything. Actually, there is a lot going on behind the scenes, but R won't output any messages to the console unless there is a problem. We created the poliscidata package to make getting started with R as simple and as straightforward as possible. At this point, you should be ready to go.

To acquaint you with the R working environment and the contents of the poliscidata package, we've written a special function called welcome. This command will generate a welcome message, output some basic information about your R session, and list the objects and functions in the poliscidata package.

```
welcome()                      # introduction to the companion environment
```

```
 __        __     _
 \ \      / /___ | |  ___  ___   _ __   ___
  \ \ /\ / // -_)| | / _| / _ \ | '  \ / -_)
   \_/\_/  \___| _|_|_\__|_\___/ |_|_|_|\___|
_|"""""|_|"""""|_|"""""|_|"""""|_|"""""|_|"""""|_|"""""|
"`-0-0-'"`-0-0-'"`-0-0-'"`-0-0-'"`-0-0-'"`-0-0-'"`-0-0-'

The poliscidata package bundles the datasets and functions featured in
Pollock and Edwards' R Companion to Political Analysis, 2nd Ed.

Your current working directory is: C:/Users/User/Documents
Use the setwd() function to change your working directory

These are the functions and objects in the companion package:

These are the functions and objects in the companion package:
 [1] "AdjR2"        "CI95"          "CI99"         "Colors"
 [5] "compmeans"    "CramersV"      "crosstab"     "csv.get"
 [9] "cut2"         "ddply"         "describe"     "fit.svyglm"
[13] "freq"         "freqC"         "gss"          "gssD"
[17] "imeansC"      "inverse.logit" "iplotC"       "lineType"
[21] "logregR2"     "nes"           "nesD"         "orci"
[25] "pchisqC"      "plotChar"      "plotmeans"    "plotmeansC"
[29] "printC"       "prop.testC"    "scatterplot"  "somersD"
[33] "sortC"        "spss.get"      "stata.get"    "states"
[37] "statesD"      "svyboxplot"    "svyby"        "svychisq"
[41] "svychisqC"    "svydesign"     "svyglm"       "svymean"
[45] "svyplot"      "svytable"      "welcome"      "world"
[49] "worldD"       "wtd.boxplot"   "wtd.chi.sq"   "wtd.cor"
[53] "wtd.hist"     "wtd.mean"      "wtd.median"   "wtd.mode"
[57] "wtd.quantile" "wtd.sd"        "wtd.t.test"   "wtd.ttestC"
[61] "wtd.var"      "xtabC"         "xtp"          "xtp.chi2"

For help, type ? followed by the function's name, or help(function_name)
```

The circled objects on the list above are the four datasets that you will analyze: gss, nes, states, and world. (For a detailed description of the datasets, see Box 1.2.) You'll also notice that the package contains four objects with similar names as our four datasets: gssD, nesD, statesD, and worldD. These are special design datasets that are used by a useful suite of functions that analyze weighted data.

The list of objects and functions in the poliscidata package may look pretty long at first, but we'll introduce them gradually and, with some practice, you'll learn how to use all sorts of R functions to analyze politics.

Box 1.2 The *Companion* Datasets

The poliscidata package has four datasets.

1. **gss**. This dataset has selected variables from the 2012 General Social Survey, a random sample of 1,974 adults aged 18 years or older, conducted by the National Opinion Research Center and made available through the Inter-university Consortium for Political and Social Research (ICPSR) at the University of Michigan. Some of the scales in gss were constructed by the authors. The variables in the gss dataset are described in the Appendix (Table A.1).
2. **nes**. This dataset includes selected variables from the 2012 National Election Study, a random sample of 5,916 citizens of voting age, conducted by the University of Michigan's Institute for Social Research and made available through ICPSR. See the Appendix (Table A.2).
3. **states**. This dataset includes variables on each of the 50 states. Most of these variables were compiled by the authors from various sources. A complete description of variables in the states dataset is found in the Appendix (Table A.3).
4. **world**. This dataset includes variables on 167 countries of the world. These variables are based on data compiled by Pippa Norris, John F. Kennedy School of Government, Harvard University, and made available to the scholarly community through her Internet site. See the Appendix (Table A.4) for a complete description of variables in the world dataset.

The four datasets included in the R package that accompanies this book contain a wealth of information about political behavior and institutions. We'll use these datasets to demonstrate a variety of research methods, but we hope your curiosity will be sparked to explore variables and relationships that we don't address here. To see the names of variables contained in the datasets, you can use the names function. For example, the following command will return the names of all the variables in the world dataset.

```
names(world)                # list names of variables in dataset
```

```
 [1]   "country"           "gini10"              "dem_level4"
 [4]   "dem_rank14"        "dem_score14"         "lifeex_f"
 [7]   "lifeex_m"          "literacy"            "oil"
[10]   "pop_0_14"          "pop_15_64"           "pop_65_older"
[13]   "fertility"         "govregrel"           "regionun"
[16]   "religoin"          "spendeduc"           "spendhealth"
[19]   "spendmil"          "hdi"                 "pop_age"
[22]   "sexratio"          "pop_total"           "pop_urban"
[25]   "gender_unequal"    "gender_unequal_rank" "arda"
[28]   "lifeex_total"      "debt"                "colony"
[31]   "confidence"        "decent08"            "dem_other"
[34]   "dem_other5"        "democ"               "democ11"
```

```
 [37]  "democ_regime"       "democ_regime08"       "district_size3"
 [40]  "durable"            "effectiveness"        "enpp3_democ"
 [43]  "enpp3_democ08"      "dnpp_3"               "eu"
 [46]  "fhrate04_rev"       "fhrate08_rev"         "frac_eth"
 [49]  "frac_eth2"          "frac_eth3"            "free_business"
 [52]  "free_corrupt"       "free_finance"         "free_fiscal"
 [55]  "free_govspend"      "free_invest"          "free_labor"
 [58]  "free_monetary"      "free_property"        "free_trade"
 [61]  "free_overall"       "free_overall_4"       "gdp08"
 [64]  "gdp_10_thou"        "gdp_cap2"             "gdp_cap3"
 [67]  "gdpcap2_08"         "gdpcap3_08"           "gdpcap08_2"
 [70]  "gdppcap08"          "gdppcap08_3"          "gender_equal3"
 [73]  "gini04"             "gini08"               "hi_gdp"
 [76]  "indy"               "muslim"               "natcode"
 [79]  "oecd"               "pmat12_3"             "polity"
 [82]  "pr_sys"             "protact3"             "regime_type3"
 [85]  "rich_democ"         "unions"               "unnetgro"
 [88]  "unnetuse"           "unpovnpl"             "unremitp"
 [91]  "unremitt"           "vi_rel3"              "votevap00s"
 [94]  "votevap90s"         "women05"              "women09"
 [97]  "women13"            "ipu_wom13_all"        "womyear"
[100]  "womyear2"           "dem_economist"        "democ.yes"
[103]  "country1"
```

An important note to commit to long-term memory: Each time that you open a new session to work with the poliscidata package, you will need to execute the following command:

```
library(poliscidata)
```

As we noted above, you can also load the poliscidata package using the "Load package" option under the Packages menu tab. We encourage you to use R commands when possible because you can save a series of commands in a script file (more on using scripts below).

We've designed this material so you can start analyzing real political science data with R quickly and easily. You may still encounter some problems or receive some unexpected warnings from the R program. You might also see a warning message that one or more packages were built under an earlier version of R than the one you are running. This issue does not seem to pose any serious problem.

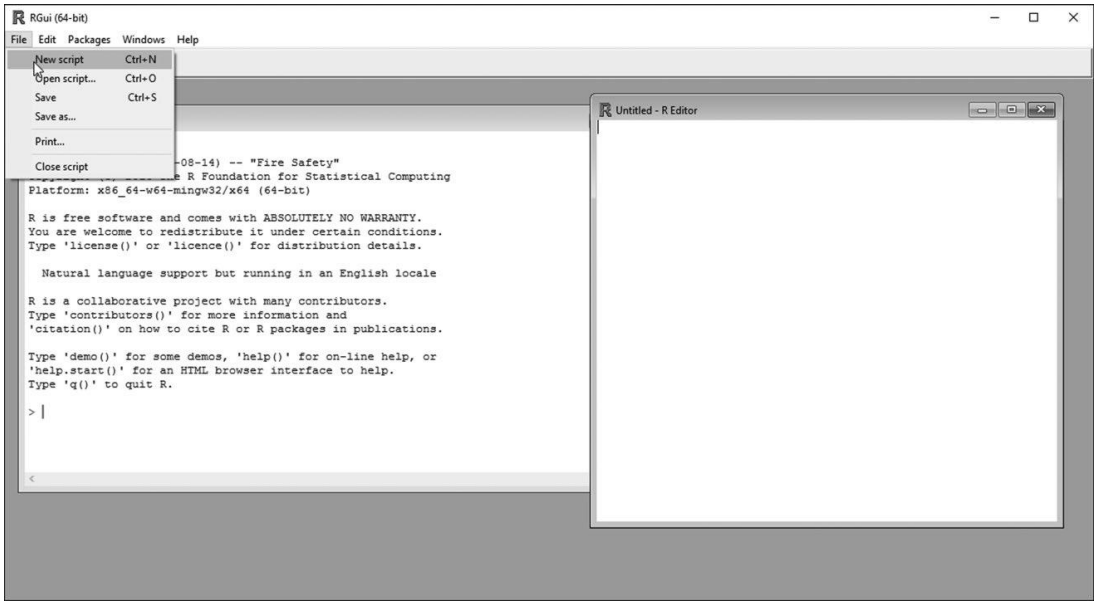
RUNNING SCRIPTS

In this book, you will create, run, and save R scripts. R scripts (called R documents on Mac OS) are documents that contain the lines of code you want R to execute (as well as comments that make your scripts easier to read and understand). You can think of an R script as a set of step-by-step instructions for the R program.

By this point, you probably have already executed some R commands successfully from the console's command line prompt, so why bother opening another window in the program and creating a script file? Sometimes getting R to do what you want it to do is tricky, so when you figure out what works, it is a good idea to save your work so you won't make the same mistakes again. If you are going to execute the same commands repeatedly, like the lines of code you need to execute each time you work with the *R Companion*, save those commands in a script file, making it easy to repeat them. Just like you save someone's phone number so you can call or text them at the touch of a button, rather than manually entering each digit of their number every time you want to contact them, saving your work in well-written script files saves time and prevents mistakes.

To create an R script, select the "New script" option under the File menu tab (or press Ctrl-N). If you're running R on a Mac OS, your version of R will say "New document" rather than "New script".

Figure 1.3 R Script Editor



The unassuming R script editor should now appear. Click in the script editor and type a couple of lines (refer to the Introduction for sample lines of R code). When you finish typing a line of code and start a new line, the line you've completed isn't automatically executed. You can run lines one at a time by clicking on the line and pressing Ctrl-R. (If you are running Mac OS, you'll *execute* a line of code by pressing Command-Enter.) You can also select the "Run line or selection" option from the Edit menu tab or right-click the line and select "Run line or selection" from the pop-up menu, but you'll find the keyboard shortcut a time-saving practice. R will execute the line of code the cursor is on and get ready to run the next line.

When you execute a line of code from the script editor, R reacts just like you entered that line of code at the command prompt in the console widow. In fact, if you look closely at the Console output, you'll see that when you execute a line of code from your script, the line you ran appears in the Console window.

To run multiple lines of code at once, select (highlight) the lines of code you want to run and press Ctrl-R (or the "Run line or selection" options discussed above). You can run an entire script at once by pressing Ctrl-A and then Ctrl-R (or the "Run all" option from the Edit menu).

Sometimes it's very helpful to run just a fragment of a line of code. We often do this to debug a line of code that's not working the way we expected. You can run part of a line of code by highlighting part of the line and pressing Ctrl-R (or the alternatives discussed above).

After you've had the chance to write and run a couple lines of test code from the script editor, let's learn how scripts are saved and re-opened. This is the big advantage of writing a script, rather than entering command line statements: You can save your work and pick up from where you left off later. To save your test script, press Ctrl-S, the ubiquitous keyboard shorthand for save. Give the script a descriptive filename, such as "testScript.R". Make sure to type the entire filename, including the .R extension. You might want to create a folder for your R scripts; you'll hopefully be developing a nice, well-organized collection soon. Feel free to close your script and even quit the R program.

To re-open a script and continue working with it, you'll need to start the R program first and select "Open script . . . " under the File menu. The R program doesn't automatically launch when you double-click an .R file the way double-clicking a word document launches a word processor. The .R extension helps the R program recognize a script file, so it is a good script-naming practice to follow. Your test script may not be especially useful moving forward, but getting in the habit of writing good R script files will pay dividends.

TEN TIPS FOR WRITING GOOD R SCRIPTS

Just as you can learn to write good essays, you can learn to write good code for a computer program. In fact, many of the principles you've learned for composing effective prose apply just as well to writing good R code. Following these suggestions will save you time and aggravation.

1. Good scripts are user-friendly. We write R code for our benefit, not the computer's. (To the computer, it all becomes a stream of 0s and 1s.) So you should write scripts that are clear and comprehensible to you.
2. When you create objects or generate new variables, give your creations clear, descriptive names. You aren't limited to names with a limited number of characters, like some old computer programming languages. Avoid the temptation to give objects fanciful, humorous, or arbitrary names (they won't be very amusing when they give you problems).
3. Understand how R treats white space and line breaks. R will interpret the space between words as the separation between objects. If you want to give an object a multi-word name, use underscores or periods to connect the words, or use camelCase (capitalizing the first letter of each subsequent word). If you enclose words in parentheses, R will interpret the quoted expression as a value to be assigned to an object or passed to an argument in a function. If you want to include a quotation mark, or some other special characters, as part of a quoted expression, you need to use special escape sequences. (Enter ?Quotes for more information.) R will interpret line breaks as the start of new commands. Sometimes, you'll want to execute long lines of code that are more easily read and edited if broken into several lines. You can enclose multiple lines of code in parentheses and R will then interpret everything enclosed in parentheses as belonging to the same command. We'll take advantage of this feature to demonstrate how to use functions with multiple arguments.
4. Lines of code that work together to complete a particular task should appear like single-space text in a script. A block of code is a set of instructions that complete a single task, are run together, and look like a block in a script file. Lines of code that complete another task should be separated into another block. For example, your script should keep a block of statements that transform a dataset variable together, a block of statements that create a graphic together, and a block of statements that estimate a statistical model together. It's the same logic you follow when you use paragraphs to organize an essay. You use several sentences to express some idea in a single paragraph and when you're ready to move on to a new idea, you start a new paragraph.
5. Easy on the eyes, easy on the brain. Use comments, white space, and line breaks to write subheadings and create visual separation in long scripts. Comments are statements intended for human readers that R does not attempt to execute. Anything you write on a line to the right of a # sign is strictly commentary (including more # signs).

```
# ------------- Create Plot of Multiple Regression Results ---------------
```

   Format longer scripts using comments, spaces between lines, and indentations, just as you would use subheadings, paragraphs, and punctuation to organize words in an essay. You want to be able to quickly skim a script to understand its basic design and purpose. This will help you locate particular lines that you want to copy or revise.

6. Use comments like "sticky note" reminders to yourself. For example, one of the early lessons in this book is how to generate descriptive statistics for different types of variables. When you learn how to produce a frequency table for an ordinal-level variable, insert a comment in your script like:

```
# Create frequency table for ordinal-level variable
# The w and plot arguments are optional
freq(x=nes$budget_deficit_x, w=nes$wt, plot=FALSE)
```

There's a good chance you'll be asking yourself at the end of the term how you created the frequency tables you made early in the term. If you follow these suggestions, you'll find a script called something like "describingVariables.R" on your computer and when you open it, you'll see your comment telling you which line(s) of code create the descriptive statistics you need. (Feel free to pat yourself on the back at this point.) What may seem like a few minutes of extra, unnecessary work in the moment will save you hours of time in the long run.

7. Save your scripts with names that clearly describe what the script does. For example, when you write an R script that makes comparisons between groups, save that script with a name like "makingComparisons.r". Don't save it with a name like "homeworkForClass.r" or "assignment4.r" because those names aren't going to help you find the code you want to use to solve a problem in the future. Write separate scripts for separate projects, just like you have different work documents for different papers.
8. Set a working directory to keep your files organized. Use a separate working directory for each project. If you're using this book as part of a class, create a directory for your class. If, subsequent to taking this class, you want to apply some of the research methods you've practiced to analyze data in a paper or for a project, you'll have a well-organized toolkit at your disposal.
9. Type the name of objects and variables as seldom as possible. Each time you type the name of an object or variable, there's a chance you type it incorrectly. Instead, to use an object declared earlier in your script, highlight the object's name, copy it, and paste the copied name where you need to reference it. If you are executing a statement that's similar to one you've already written, copy and paste what you've already written and then edit only those parts of the copied code that needed to be changed to complete the task at hand. If you're working on a project that's similar to one you've worked on before, re-use your earlier work as much as possible.
10. Save your scripts frequently in case R stops responding. It's a stable program, but it will occasionally seize up. Even better, save your work on some kind of Cloud storage so it's convenient for you to work on multiple machines.

MANAGING R OUTPUT: GRAPHICS AND TEXT

For practically all of the examples and exercises in this book, you will produce and interpret text output—frequency distributions, cross-tabulations, tables of regression coefficients, and so on. Quite often, you will want to create an accompanying graph or chart, such as a mosaic plot or scatterplot. R graphics are remarkably easy to work with: Create them, print them, or copy/paste them into a document, such as a Word document. By contrast, nicely formatted text requires a bit more work.

To illustrate, we will use the freq command (from the descr package) to obtain a frequency distribution (text) and bar chart (graphic) of nes$pid_x, a measure of party identification.[2] The nes dataset needs to be weighted, so we will include the weight variable, nes$wt, in the freq command. (Be sure to read Box 1.3, A Special Note on Weights). At the prompt or in the script file, type and run the following function call:

```
freq(nes$pid_x, nes$wt)          # Example: graphics and text output
```

```
nes$pid_x
       Frequency  Percent  Valid Percent
StrDem   1156.02  19.5405         19.61
WkDem     890.31  15.0492         15.10
IndDem    690.86  11.6778         11.72
Ind       839.33  14.1875         14.23
IndRep    720.81  12.1841         12.22
WkRep     731.41  12.3632         12.40
StrRep    867.52  14.6640         14.71
NA's       19.74   0.3337
Total    5916.00 100.0000        100.00
```

[2] Chapter 2 covers the freq command in detail.

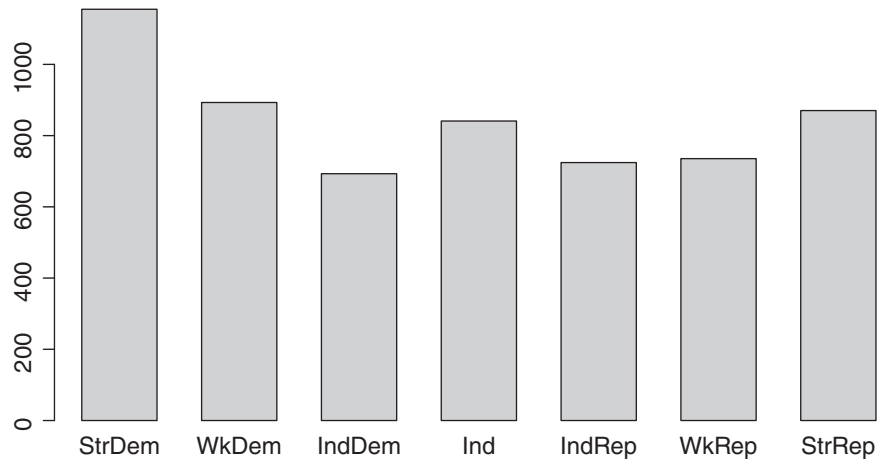
Box 1.3 A Special Note on Weights

The states and world datasets are unweighted. In analyzing unweighted data, you do not need to adjust for sampling bias, because each state or country is equally and adequately represented in the dataset. For example, to calculate the average percentage of women in parliaments of the world (recorded in the variable world$women09), you would ask R to sum the percentages for each country and divide by the number of countries.

By contrast, the gss and nes datasets must be weighted. Why is this? In unweighted form, these datasets contain sampling bias—that is, some groups are over- or under-represented when compared with the overall population of adults. So, for example, if you wanted to calculate the average age of respondents in the nes dataset, the unweighted average would be distorted, because not all age groups are equally and adequately represented in the dataset. To correct for this bias, survey designers provide sampling weights. Therefore, in order to obtain accurate results from the two survey datasets, gss and nes, you will need to weight your analyses by the appropriate sampling weight. For nes, the weight variable is nes$wt; for gss, it is gss$wtss.

Most of the base R functions do not permit sampling weights. Fortunately, the extra packages you installed in this chapter contain procedures that can be used with weighted data (such as gss and nes) or unweighted data (such as states and world). On rare occasion, however, you will learn separate procedures, one for weighted data and one for unweighted data.

Figure 1.4 Sample R Graphics Output



Consider the results. As you can see, R creates a graphic, a bar chart of nes$pid_x, and displays it in a separate window. If you're using Windows, you can right-click on the graphic to copy or save the figure in a desired format, or you can print it directly. Not too much to it. An editable version of the console's frequency distribution table, on the other hand, requires a few additional steps. There are a couple of ways to manage R Console output. You'll frequently want to incorporate the results of your analysis into documents.

To incorporate R Console output into informal documents, like rough drafts of papers or class assignments (check with your instructor though), you can copy text from the R Console and paste it into a word processor. The result typically looks disorganized and confusing because the pasted text appears in your word processor's default font, which is typically a proportional font (such as Times New Roman). If you change the font used to display R Console out in a Word document to a monospace font (such as Courier New or Lucida Console), you can replicate the basic formatting of tabular results you see in the R Console.

Figure 1.5 Simple Table Formatting

Default Word Format

nes$pid_x

	Frequency	Percent	Valid Percent
StrDem	1156.02	19.5405	19.61
WkDem	890.31	15.0492	15.10
IndDem	690.86	11.6778	11.72
Ind	839.33	14.1875	14.23
IndRep	720.81	12.1841	12.22
WkRep	731.41	12.3632	12.40
StrRep	867.52	14.6640	14.71
NA's	19.74	0.3337	
Total	5916.00	100.0000	100.00

Monospace Font, Single Spaced

```
nes$pid_x
       Frequency  Percent Valid Percent
StrDem   1156.02  19.5405          19.61
WkDem     890.31  15.0492          15.10
IndDem    690.86  11.6778          11.72
Ind       839.33  14.1875          14.23
IndRep    720.81  12.1841          12.22
WkRep     731.41  12.3632          12.40
StrRep    867.52  14.6640          14.71
NA's       19.74   0.3337
Total    5916.00 100.0000         100.00
```

For more formal presentations, like final drafts of papers or any analysis you plan on sharing with an audience, you should edit and format Console output. The printC function will export R tabular output as an .html file to your working directory. These files can subsequently be opened in a web browser, copied/pasted into a document file such as Word, and then edited for appearance and readability. The printC function will create an .html file, named "Table.Output.html", that will be the repository for all the tables you wish to export, edit, and print. To create an editable table using the printC function, insert the desired command within printC's parentheses.[3] For example, to print the frequency distribution table for nes$pid_x using the printC function, enter the following:

```
printC(freq(nes$pid_x, nes$wt))     # Print table output to html file
```

This statement quietly exports the frequency distribution to Table.Output.html in the working directory. If you don't know where to find the Table.Output.html file, enter the getwd() command. (See the section "Creating Tables of Regression Results" in Chapter 8 for more instruction on the printC command and formatting tables for formal works.)

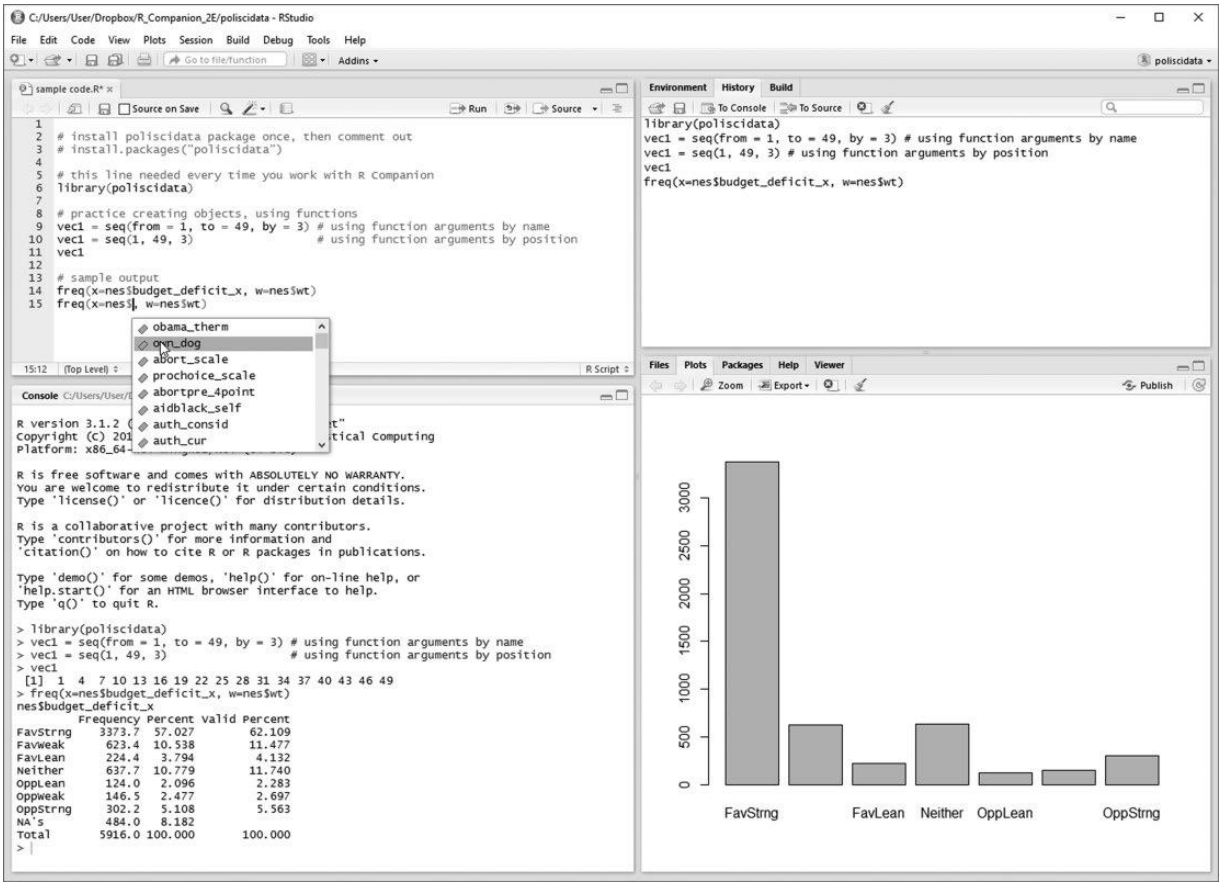
ADDITIONAL SOFTWARE FOR WORKING WITH R

In this section, we discuss some options to making the R environment easier to use. As we've discussed, the R environment is relatively spare and efficient. Its graphical user interface is limited and little analysis can be conducted using its pull-down menus. Fortunately, some software developers are working to address this void and make R more intuitive and user-friendly. We'll take a look at two of these developments, R Studio and R Commander.

R Studio is an interface for R that is available for Windows, Mac OS, and Linux. It's a free program (commercial enterprises may pay more for technical support). You can download R Studio and learn more about it from its website: https://www.rstudio.com/. For new R users, R Studio has some excellent features. We particularly like R Studio's ability to suggest and auto-complete code. If you look closely at the screenshot in Figure 1.6, you'll see that when we type "nes$" we get a pull-down menu of variables in the nes dataset, a very helpful feature. Other nice features include an enhanced Editor with line numbers and smart text coloring, a command history pane, a help file pane, and some nice options for saving graphics.

[3] If you get an error message that says "function not found", that means you either haven't loaded the companion packages or didn't type the name of the function correctly.

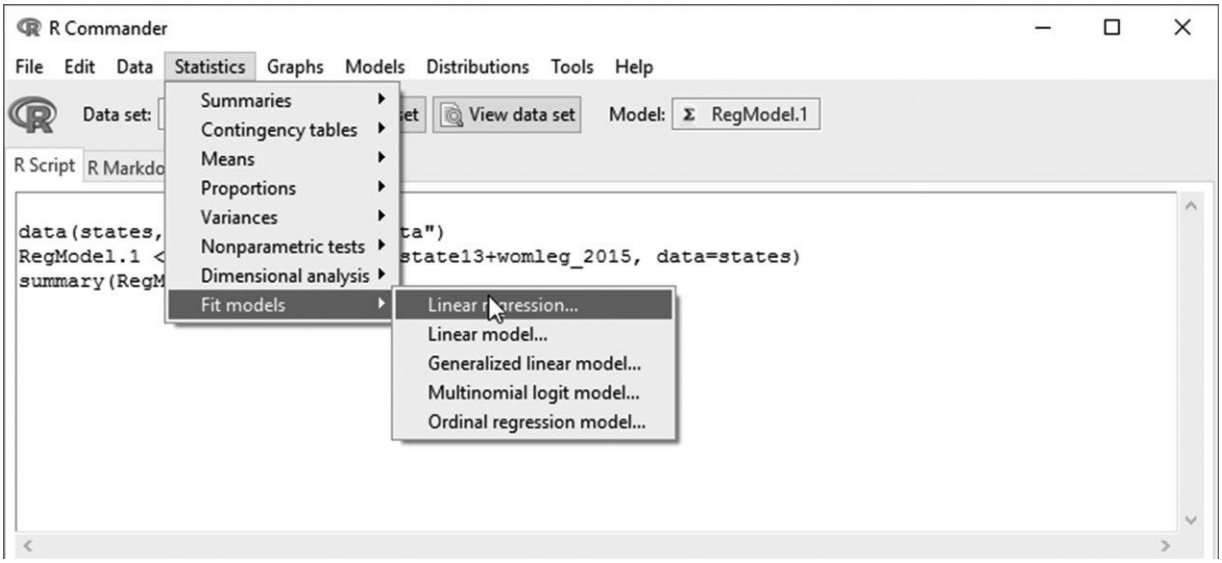
Figure 1.6 R Studio Screenshot



R Commander is an R package that allows users to execute a suite of commands using drop-down menus and a graphical user interface. If you have used statistics programs like Stata or SPSS before, you might like the look and feel of R Commander. Once you have R up and running, it's easy to install and load R Commander:

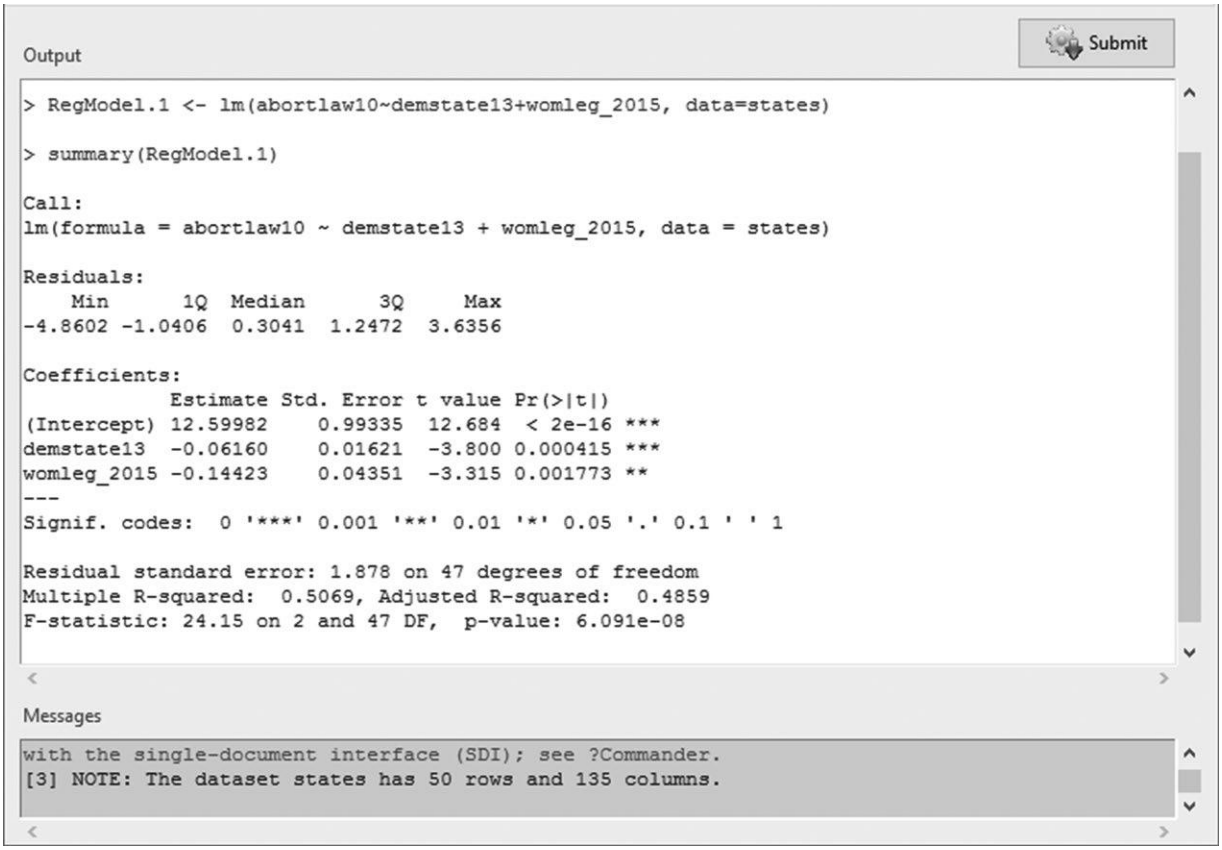
```
# Install and load the R Commander package
install.packages("Rcmdr")
library(Rcmdr)
```

Figure 1.7 R Commander Screenshot



(Continued)

Figure 1.7 (Continued)



Of course, not all R packages and functions are integrated into R Commander, but the package makes some of the most commonly used statistical methods easy to use.

DEBUGGING R CODE

When you execute a statement in R, you might get an error message telling you that an action you performed did not achieve the desired result. In fact, this happens all the time. Learning how to identify and correct mistakes is how you practice and develop your skills as an R user. Error messages are different from—and worse than—warning messages, in which R simply makes note of something it encountered while executing a command, such as missing data.

In our experience working with students, most errors are caused by typos and minor syntactical mistakes. When you type the name of an object or function incorrectly, you typically get a "Function not found" or "Object not found" message. If you see these error messages, carefully check how you've spelled the name of the function or object that's not found. Remember that R is case sensitive and will interpret a space between words as the beginning of a new object or function.

Test lines of code as you write them. Don't wait until you've written all the commands you think necessary to complete a task before running the code. Remember that R can run code one line at a time, just part of a line, or all lines at once.

If you are working with a function that has many arguments, start by executing the function in its most basic form and add arguments incrementally. Most R users start by finding a working example that's similar to what they want to do and adapting the example to suit their needs. It's an iterative process with a lot of trial and error. Functions that create graphics are a good example of this. The best way to create beautiful graphics in R is to start with a basic working figure and then refine that figure by defining values for optional arguments and adding layers of information.

If your function call isn't working, test each executable component of a function call. For example, if you're applying a function to a variable, highlight the name of the variable and run it to see its contents. Are you applying the function to a variable that isn't there or is not in the form you expected? One of the most common errors that a user makes when working with a variable is not specifying the dataset in which it resides.

What if you run a function and you see a plus sign (+), rather than the > prompt? The plus sign is the continuation prompt, meaning that R is waiting for more user input. The most common cause for this error is having more open parentheses signs than closed parentheses signs in a statement, which tells R you hasn't finished calling a function yet. Either execute the right number of closing parentheses signs or click the "STOP" button on the console window. (To wake up the STOP button, click in the Console window.) Parentheses, brackets, and quotation marks come in pairs. Make sure each opening parenthesis "(" and opening bracket "[" has a corresponding closing parenthesis ")" and closing bracket "]". This can get a little confusing when you write complex statements with nested functions. Develop the habit of typing a set of parentheses, quotations marks, brackets, or braces anytime you use them, then move the cursor back inside the set to fill in values, arguments, and so forth. Some script editors will do this automatically and that can be helpful.

If you copy and paste sample code from a Word document or web page, beware of curly quotation marks. When you use single or double quotation marks in a word processor, the program uses "curly" quotation marks for style. In contrast, your R script editor uses "straight" quotation marks. This can be a difficult bug to spot in code.

If you run a number of statements at once and get a lot of error messages, locate the first line in your script that prompted an error message—that's probably where you need to start debugging. A small typo early in a script can set off a chain reaction of errors. Don't be alarmed by a cascade of errors and warnings: Just locate the start of the error messages, read the message for any helpful information, and address one problem at a time.

In the following chapter of this book, we are going to show you how to conduct some fundamental analysis using R. For particular methods, we will feature one function or set of functions. As mentioned above, there are thousands of R packages and hundreds of thousands of functions. It should be no surprise, then, that there is a lot of overlap among functions and there is often more than one way to solve a particular problem. In this book, we emphasize functions with sensible default values, including handling of missing data, that allow researchers to use sampling weights. We have only scratched the surface of R's capabilities, but we believe the best way to learn how to use R is hands-on experience solving problems with the program.

EXERCISES

1. This chapter described R's names function. Use the names function to find out which variables are contained in the states dataset. Which of the following variables are in the states dataset? (Check all that apply.)

   - ❑ cigarettes
   - ❑ denom
   - ❑ gunlaw_scale
   - ❑ rep_therm
   - ❑ partyid3
   - ❑ attend_pct

2. Which of the following uses correct form in telling R where to locate the variable named gini10 in the world dataset? (Check one.)

   - ❑ gini10
   - ❑ gini10$world
   - ❑ world$gini10

3. The states dataset contains abortlaw10, the number of restrictions that each state puts on access to an abortion. Values range from 0 (no restrictions) to 10 (ten restrictions). Use the freq command to obtain a

frequency distribution and bar chart. (*Hint*: The states dataset does not require weighting, so you do not need to include a weight variable in the freq expression.)

A. Print the graph.

B. Following this chapter's printC example, create a nicely formatted table of the abortlaw frequency distribution in a word-processing file, such as Word. When you edit the table in your word processor, give it this title: "Number of Abortion Restrictions." Print the formatted table.

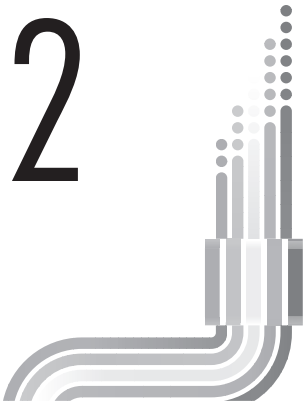
4. Each time you start an R session using the R package that bundles the functions and dataset used in this book, you must type and run which one of the following expressions? (Check one.)

   ❑ 'library(poliscidata)'

   ❑ 'welcome()'

   ❑ 'help( )'

2



Descriptive Statistics

Objective	*Functions Introduced*	*Author or Source*
Measuring central tendency	freq {descr}	Jakson Aquino[1]
	freqC {rcompanion}	Philip Pollock and Barry Edwards[2]
	wtd.mode {rcompanion}	Philip Pollock and Barry Edwards
	wtd.median {rcompanion}	Philip Pollock and Barry Edwards
	wtd.mean {Hmisc}	Frank E. Harrell, Jr.[3]
	describe {Hmisc}	Frank E. Harrell, Jr.
Measuring dispersion	wtd.hist {weights}	Josh Pasek[4]
	wtd.var {Hmisc}	Frank E. Harrell, Jr.
	wtd.sd {rcompanion}	Philip Pollock and Barry Edwards
Getting case-level information	sortC {rcompanion}	Quan Li[5]

Analyzing descriptive statistics is the most basic—and sometimes the most informative—form of analysis you will do. Descriptive statistics reveal two attributes of a variable:

- Central tendency (the variable's typical value)
- Dispersion (how spread out or varied the variable's values are)

---

[1] Aquino, J. (2012). *descr: Descriptive statistics* (R package version 0.9.8). Includes R source code and/or documentation written by Dirk Enzmann, Marc Schwartz, and Nitin Jain. Available at http://CRAN.R-project.org/package=descr

[2] The companion function, freqC, is a slightly modified version of freq.

[3] Harrell, F. E., Jr. (2012). *Hmisc: Harrell miscellaneous* (R package version 3.9-3). Contributions from many other users. Available at http://CRAN.R-project.org/package=Hmisc

[4] Pasek, J. (2012). *weights: Weighting and weighted statistics* (R package version 0.75). With some assistance from Alex Tahk and some code modified from R-core. Available at http://CRAN.R-project.org/package=weights

[5] Pollock, P. H. (2013). *An R companion to political analysis*. Thousand Oaks, CA: SAGE/CQ Press. Based on order {base}, R Development Core Team. (2011). *R: A language and environment for statistical computing*. Vienna, Austria: Author. Available at http://www.R-project.org/

The precision with which we can describe central tendency for any given variable depends on the variable's level of measurement. Nominal-level variables—for example, gender, race, or religious denomination—have values that simply differentiate categories: Women are in one category, men in a different category. R refers to nominal variables as *unordered factors*. For unordered factors, we can identify the *mode*, the most common value of the variable. Ordinal-level variables—a survey question gauging strength of partisanship, for example, or measuring level of support for or opposition to public policy—are *ordered factors*. Because ordinal variables, or ordered factors, have values that convey the relative amount of a characteristic—an individual who "strongly" supports a policy has a greater amount of support than does an individual who "somewhat" supports it—we can find the mode and the *median*, the value of the variable that divides the cases into two equal-size groups. For interval-level or *numeric* variables, we can obtain the mode, median, and arithmetic *mean*, the sum of all values divided by the number of cases.

Finding a variable's central tendency is ordinarily a straightforward exercise. Simply read the output and report the numbers. Describing a variable's degree of dispersion or variation, however, often requires informed judgment.[6] Here is a general rule that applies to any variable at any level of measurement: A variable has no dispersion if all the cases—states, countries, people, or whatever—fall into the same value of the variable. A variable has maximum dispersion if the cases are spread evenly across all possible values of the variable such that the number of cases in one category equals the number of cases in every other category. For example, if observations take on one of two values of a variable, dispersion is greatest when half of the observations have one value and half, the other value. This general rule is particularly useful for variables measured at the nominal or ordinal level. When a variable is measured at the interval level, we can calculate statistical measures of dispersion, such as variance and standard deviation.

INTERPRETING MEASURES OF CENTRAL TENDENCY AND VARIATION

Central tendency and variation work together in providing a complete description of any variable. Some variables have an easily identified typical value and show little dispersion. For example, suppose you were to ask a large number of U.S. citizens what sort of economic system they believe to be the best: capitalism, communism, or socialism. What would be the modal response, the economic system preferred by most people? Capitalism. Would there be a great deal of dispersion, with large numbers of people choosing the alternatives, communism or socialism? Probably not. In other instances, however, you may find that one value of a variable has a more tenuous grasp on the label "typical." And the variable may exhibit more dispersion, with the cases more evenly spread out across the variable's other values. For example, suppose a large sample of voting-age adults were asked, in the weeks preceding a presidential election, how interested they are in the campaign: very interested, somewhat interested, or not very interested. Among your own acquaintances, you probably know a number of people who fit into each category. So even if one category, such as "somewhat interested," is the median, there are likely to be many people at either extreme: "very interested" and "not very interested." This would be an instance in which the amount of dispersion in a variable—its degree of spread—is essential to understanding and describing it.[7]

These and other points are best understood by working through some guided examples using the GSS dataset. In the examples that follow, you will become better acquainted with the freq function, introduced in Chapter 1. The freq command produces frequency distributions and bar charts for nominal, ordinal, or interval variables. In this chapter, you also will use the describe function to obtain descriptive statistics for interval-level variables. You will learn to use wtd.hist (from the weights package) to create histograms, graphic displays that enhance the description of interval variables. Finally, you will learn to sort a dataset to obtain case-specific information about interesting variables using the sortC function.

---

[6] In this chapter, we will use the terms *dispersion*, *variation*, and *spread* interchangeably.

[7] For elaboration on these points with additional examples, see Pollock, P. H. (2016). *The essentials of political analysis*, 5th ed. Thousand Oaks, CA: SAGE/CQ Press, Chapter 2.

DESCRIBING NOMINAL VARIABLES

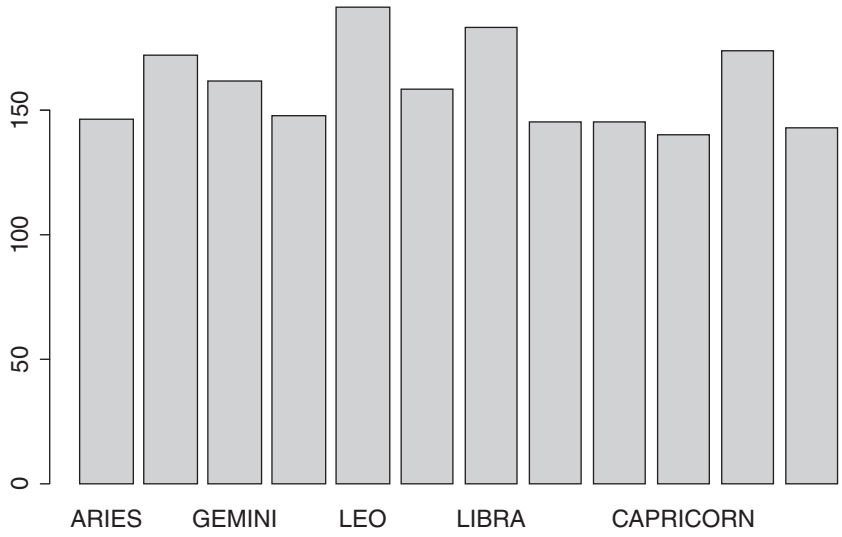
Nominal-level variables simply differentiate the unit of analysis into different groups or categories. One value of a nominal-level variable is no more or less than another value, they are just different values. In the R environment, nominal-level variables are classified as unordered factors.

In this section, you will obtain a frequency distribution for a nominal-level variable, zodiac, which records GSS respondents' astrological signs. The variable, zodiac, is in the GSS dataset, which requires a weight variable, wtss. Recall R's rule: To R, zodiac is gss$zodiac, and wtss is gss$wtss. To obtain a frequency distribution table and bar chart of zodiac, enter:

```
freq(gss$zodiac, gss$wtss)         # Describing a Nominal-Level Variable
```

```
             Frequency  Percent  Valid Percent
ARIES           145.78    7.381          7.649
TAURUS          171.59    8.688          9.003
GEMINI          161.40    8.172          8.469
CANCER          147.73    7.480          7.751
LEO             190.35    9.638          9.988
VIRGO           158.58    8.029          8.321
LIBRA           183.37    9.285          9.621
SCORPIO         145.12    7.348          7.614
SAGITTARIUS     145.36    7.360          7.627
CAPRICORN       140.29    7.104          7.361
AQUARIUS        173.52    8.786          9.104
PISCES          142.78    7.229          7.492
NA's             69.14    3.501
Total          1975.00  100.000        100.000
```

Figure 2.1 Distribution of Zodiac Signs in the GSS Dataset



R produces a frequency distribution table in the console window and a bar chart in the graphics window. If you want to generate descriptive statistics for a nominal-level variable without weighting observations (for instance, if you are analyzing a variable in the states or world datasets), simply omit the second argument in the function above.

The value labels for each astrological sign appear in the left-most column of the frequency distribution table, with Aries occupying the top row of numbers and Pisces the bottom row. There are three columns of numbers: Frequency, Percent, and Valid Percent. The Frequency column tells us the number of respondents—more accurately, the number of respondents weighted by the sampling weight—having each zodiac sign. Percent is the percentage of respondents in each category of the variable, counting missing cases (NA's). Valid Percent is the column to focus on. So, for example, ignoring NA's, about 172 respondents (171.59), or 9.003 percent of the sample, have Taurus as their astrological sign.

Consider the Valid Percent column of the frequency distribution table with the central tendency of this variable in mind. What is the mode, the most common astrological sign? For nominal variables, the answer to this question is (almost) always an easy call: Simply find the value with the highest percentage of responses. Leo is the modal sign. To simply identify a variable's mode, without consulting a frequency distribution table, try the wtd.mode function:

```
wtd.mode(gss$zodiac, gss$wtss)       # Finding the Modal Value
```

```
[1] "LEO"
```

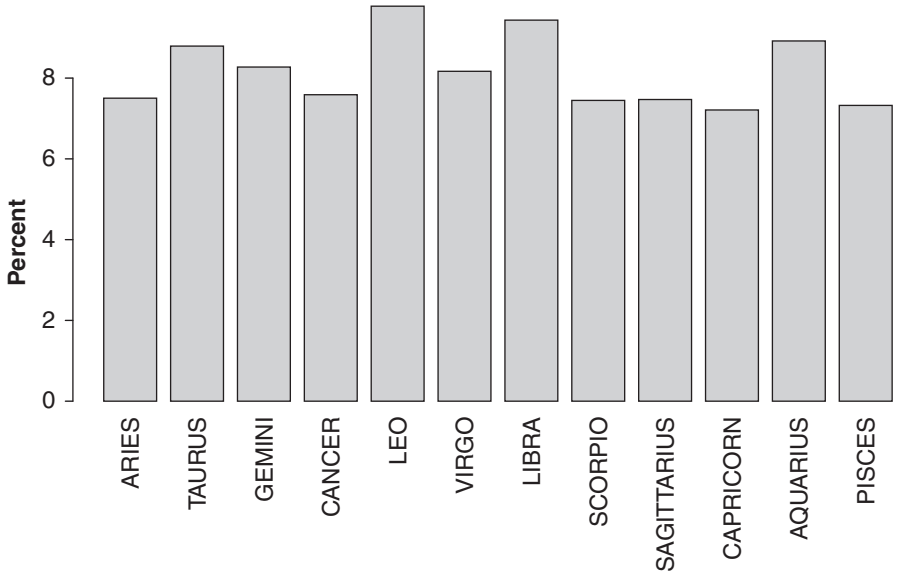
Do zodiac signs have little dispersion or a lot of dispersion? Take a close look at the Valid Percent column of the frequency distribution table and consider the height of the bars in the bar chart. Recall that a variable has no dispersion if the cases are concentrated in one value of the variable; there would be only one bar containing 100 percent of the cases. A variable has maximum dispersion if the cases are spread evenly across all values of the variable; all the bars would be the same height. Are most of the cases concentrated in Leo, with only one or two heavily populated bars? Or are there many cases in each value of zodiac, with many bars of roughly equal height? Since respondents are widely dispersed across the values of zodiac, we would conclude that zodiac has a high level of dispersion.

When you visually represent data, your plot or chart may need refinement. This is especially true for factor variables having a large number of categories (zodiac has 12) with long value labels. For example, notice that freq labeled only 5 of the 12 zodiac signs in the chart (you may see fewer or more labels depending on the size of your graphics window; Figure 2.1). Later in this chapter, and throughout the remainder of the book, you will learn how to fine-tune R's graphics, adding axis labels, titles, legends, line types, and so on. For present purposes, however, a slight variation on the freq function, freqC, comes in handy for factors with many possible values and long value labels. Try this:

```
freqC(gss$zodiac, gss$wtss)        # Describing a Nominal-Level Variable
                                   # Uses Modified Plot Settings
```

(*Hint*: In the script editor, copy your original freq command, paste it onto a new line, and edit 'freq' to read 'freqC'. Minimize typing to avoid introducing typos in your R code.) The frequency distribution reappears, accompanied by a bar chart in which all the values of zodiac are labeled (Figure 2.2). Also, the vertical axis records valid percentages instead of frequencies.[8]

Figure 2.2 Distribution of Zodiac Signs in the GSS Dataset



[8] If the x-axis labels are still cropped by the graphics window, try re-sizing your graphics window to a narrower shape and re-running the freqC command with the graphics window open. If this doesn't resolve the problem, you may need to add a line of code to specify the width of the outer margin around the bar chart. To do this, try executing this line of code *before* the freqC command: par(omi=c(.2, 0, 0, 0)). This code sets a graphics parameter for the outside margin size clockwise around the figure (below, left, above, right).

Bar charts can be a useful interpretive tool. Even so, you may not always want freq to produce one. You can suppress the chart by including the additional argument, 'plot=FALSE', which may be abbreviated, 'plot=F'. For example:

```
freq(gss$zodiac, gss$wtss, plot=F)   # Describing a Nominal-Level Variable
                                     # Suppresses Plot of Results
```

DESCRIBING ORDINAL VARIABLES

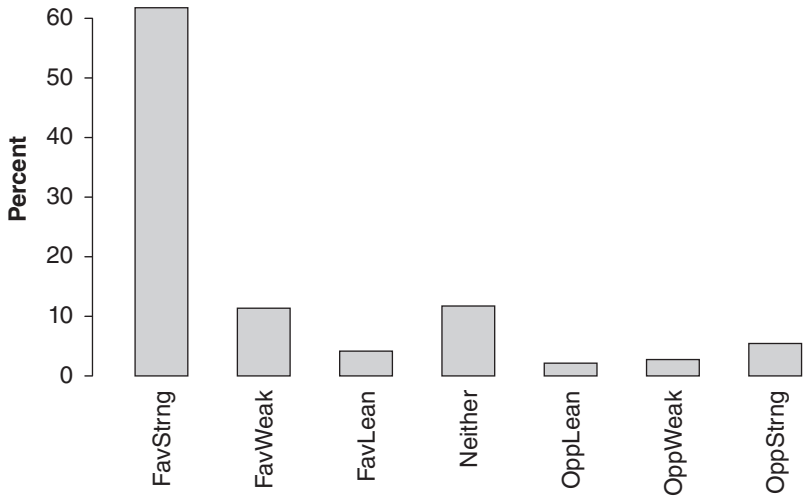
Next, you will analyze and describe ordinal-level variables, two of which have relatively little variation and a third which is more spread out. These variables appear in the NES dataset, which contains a wealth of survey data gauging individuals' opinions on a variety of public policies.

The NES variable, budget_deficit_x, asks whether respondents favor reducing the federal budget deficit. Similarly, the variable congress_job_x asks whether respondents approve of the way Congress does its job. On both questions, respondents could favor strongly, favor moderately, favor slightly, take a middle position, oppose slightly, oppose moderately, or oppose strongly. For seven-category ordered factors like these, we will run freqC. To obtain representative results, we should use the survey weights variable, nes$wt.

To create descriptive summaries of nes$budget_deficit_x and nes$congress_job_x, we execute the following lines of R code:

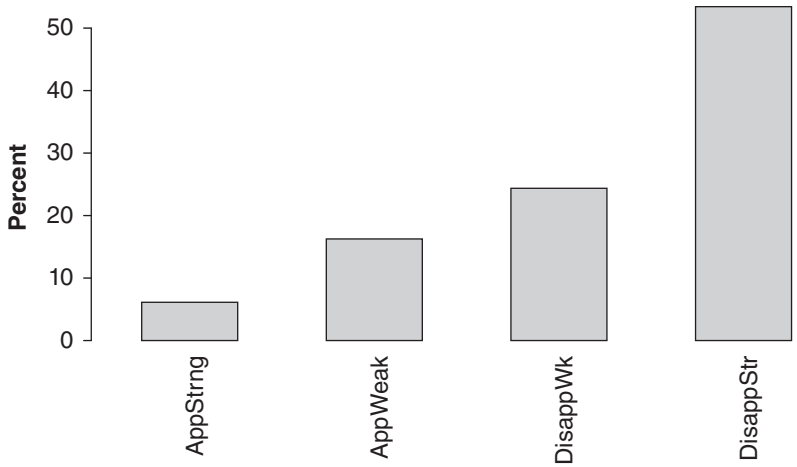
```
freqC(nes$budget_deficit_x, nes$wt)          # Describing Ordinal Variable

freqC(nes$congapp_job_x, nes$wt)             # Additional Example
                                             # Describing Ordinal Variable
```

Figure 2.3 Public Support for Reducing the Federal Deficit




```
         Frequency  Percent Valid Percent
FavStrng    3373.7   57.027        62.109
FavWeak      623.4   10.538        11.477
FavLean      224.4    3.794         4.132
Neither      637.7   10.779        11.740
OppLean      124.0    2.096         2.283
OppWeak      146.5    2.477         2.697
OppStrng     302.2    5.108         5.563
NA's         484.0    8.182
Total       5916.0  100.000       100.000
```

Figure 2.4 Public Opinion of Congress




```
           Frequency  Percent  Valid Percent
AppStrng       342.3    5.786          6.177
AppWeak        899.8   15.209         16.236
DisappWk      1343.9   22.717         24.250
DisappStr     2955.9   49.964         53.337
NA's           374.1    6.323
Total         5916.0  100.000        100.000
```

The results of these lines of code are similar to descriptive statistics we generated for the nominal variable zodiac above. In both cases, R produces a frequency distribution table in the console and a bar chart in the graphics window.

How would you describe the central tendency and dispersion of NES respondents' opinions about reducing the federal budget deficit or how Congress is doing its job? Because budget_deficit_x and congress_job_x are ordinal variables, we can report both their modes and their medians. The modal mode opinion regarding budget deficit reduction, clearly enough, is "FavStrng" (favor strongly), the option chosen by 62.11% of NES respondents (Figure 2.3). Fully 53.34% of respondents "DisappStr" (disapprove strongly) of the job being done by Congress (Figure 2.4). (As before, make sure to focus on the Valid Percent column.)[9]

What about the median values of these variables? For ordered factors, freq and freqC return a cumulative percent column ("Cum Percent").[10] This column reports the percentage of cases falling in *or below* each value of the variable. *The median for any ordinal or interval variable is the 50th percentile, the category below which 50 percent of the cases lie.* Is the first category, "favor strongly," the median public opinions about budget deficit reduction? Yes, it is. The 50th percentile must lie within this heavily populated response category. To simply identify a variable's median value, without consulting a frequency distribution table, try the wtd.median function, illustrated below:

```
wtd.median(nes$budget_deficit_x, nes$wt)          # Finding Median Value

wtd.median(nes$congapp_job_x, nes$wt)             # Additional Example
                                                  # Finding Median Value
```

```
[1] "FavStrng"

[1] "DisappStr"
```

[9] The encoded values for these variables are abbreviated in the dataset. While the abbreviated labels are useful, one might want to modify the value labels to produce a table and/or figure for an audience. In Chapter 3, we discuss methods for transforming and relabeling variable values.

[10] If the frequency distribution table that the freq or freqC functions generate in the R console, omit cumulative percentages and use the class function to determine whether the variable you are analyzing is classified as an ordered factor. If not, you can use the as.ordered function to reclassify the variable as an ordered factor; either nest the as.ordered command as the first argument to freq or freqC or create a new variable and use your new variable as the first argument to freq or freqC. See Chapter 3 for additional information on reclassifying variables.

The output from these commands should coincide with what you learned from studying the cumulative percentages in the frequency distribution table: The median NES respondent strongly favors reducing the budget deficit and strongly disapproves of the job being done by Congress.

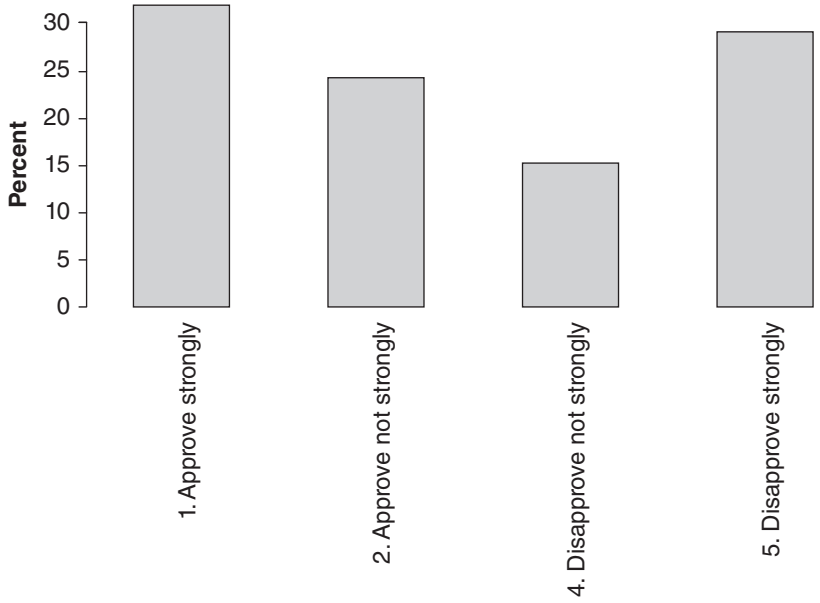
Does budget_deficit_x have a high or low degree of dispersion? If budget_deficit_x had a high level of variation, then the percentages of respondents holding each position would be about equal, much like the zodiac variable that you analyzed earlier. So roughly one-seventh, or 14 percent, would fall into each of the seven response categories. If budget_deficit_x had no dispersion, then all the cases would fall into one value. That is, one value would have 100 percent of the cases, and each of the other categories would have 0 percent. Which of these two scenarios comes closest to describing the actual distribution of respondents across the values of budget_deficit_x? It seems clear that budget_deficit_x is a variable with a relatively *low* degree of dispersion. Indeed, over three-quarters of all respondents fall on the "favor" side of this policy issue (cumulative percentage, 77.72), differing only in the strength of that opinion.

Now let's take a look at another NES variable, nes$presapp_war_x, an ordinal-level variable that encodes how NES respondents feel about President Barack Obama's handling of the war in Afghanistan. Execute the following code to generate a frequency distribution table and a bar graph that describe public opinion. Consider the distribution of public opinion presented here. Examine the Valid Percent column and the bar graph.

```
freqC(nes$presapp_war_x, nes$wt)          # Example, Descriptive Statistics
                                          # Describing Ordinal Variables
```

```
                          Frequency  Percent   Valid Percent
1. Approve strongly          1792.5   30.299           31.73
2. Approve not strongly      1362.0   23.023           24.11
4. Disapprove not strongly    857.9   14.502           15.18
5. Disapprove strongly       1637.5   27.679           28.98
NA's                          266.0    4.497
Total                        5916.0  100.000          100.00
```

Figure 2.5 Public Support for President's Handling of War in Afghanistan



Do common measures of central tendency such as the mode and the median accurately convey public sentiment about the president's handling of the war in Afghanistan? The two measures provide inconsistent impressions of public opinion. What is the mode? Technically, "approve strongly" (31.73 percent) is the mode, although "disapprove strongly" (at 28.98 percent) is a close rival for that designation (Figure 2.5). The median sentiment is "approve not strongly." Split results like this tell us that high variation, not central tendency, is the character trait to emphasize. One could say that public opinion is deeply divided on this controversial issue, with slightly more than half of the electorate on the "approve" side of the scale and slightly less than half on the disapprove side.

If you try to apply mathematical functions like mean, wtd.var, or wtd.sd to ordinal or nominal variables, you may see the "not meaningful for factors" error message. This error indicates you are attempting to use a function that is not intended for ordered factors. In some cases, changing the class of the variable to numeric solves the problem (assuming the variable can be treated as numeric data). In Chapter 3, we discuss methods for converting ordinal values to numeric values.

DESCRIBING THE CENTRAL TENDENCY OF INTERVAL VARIABLES

We now turn to the descriptive analysis of interval-level variables (classified as numeric data in R). An interval-level variable represents the most precise level of measurement. Unlike nominal variables, whose values stand for categories, and ordinal variables, whose values can be ranked, the values of an interval variable *tell us the exact quantity of the characteristic being measured*.

Because interval variables have the most precision, they can be described more completely than can nominal or ordinal variables. For any interval-level variable, we can report its mode, median, and arithmetic average, or *mean*. In addition to these measures of central tendency, we can make more sophisticated judgments about variation. The most common measures of the dispersion of interval variables are variance and standard deviation.

Additionally, one can determine if an interval-level distribution is *skewed*. What is skewness and how do you know it when you see it? Skewness refers to how symmetrical a distribution is. If a distribution is not skewed, the cases tend to cluster symmetrically around the mean of the distribution, and they taper off evenly for values above and below the mean. If a distribution is skewed, by contrast, one tail of the distribution is longer and skinnier than the other tail. Distributions in which a small number of cases occupy extremely high values of an interval variable—distributions with a longer, skinnier right-hand tail—have a *positive skew*. If the distribution has a few cases at the extreme lower end—the distribution has a longer, skinnier left-hand tail—then the distribution has a *negative skew*.

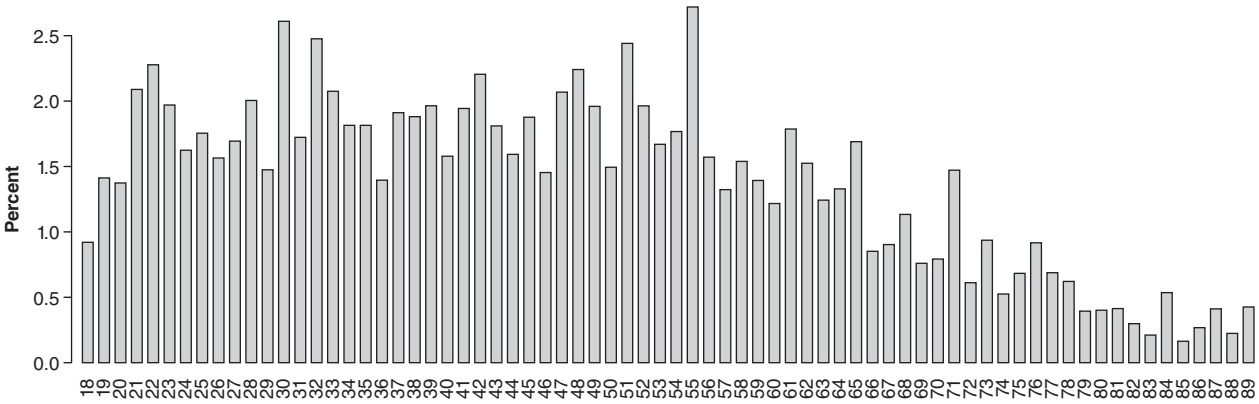
When a distribution is highly skewed, it is a good practice to use the median instead of the mean in describing central tendency. Skewness has a predictable effect on the mean. A positive skew tends to pull the mean upward; a negative skew pulls it downward. However, skewness has less effect on the median. Since the median reports the middle-most value of a distribution, it is not tugged upward or downward by extreme values.

To illustrate how we can use R to describe the central tendency and dispersion of an interval-level variable, we will analyze gss$age, a numeric variable. Age qualifies as an interval-level variable since its values impart each respondent's age in years. To obtain a frequency distribution table and bar chart, run freqC on gss$age, weighted by gss$wtss. (Notice that the R functions we used to generate descriptive statistics for nominal and ordinal-level variables also work for interval-level variables.)

```
freqC(gss$age, gss$wtss)          # Describing Interval Variables
```

```
      Frequency     Percent     Valid Percent
18       18.113      0.9171            0.9195
19       27.737      1.4044            1.4081
20       27.165      1.3754            1.3790
21       40.968      2.0743            2.0798
22       44.855      2.2711            2.2770
23       38.798      1.9645            1.9696
24       32.007      1.6206            1.6248
25       34.618      1.7528            1.7574
26       30.872      1.5631            1.5672
                  . . .
87        8.077      0.4090            0.4100
88        4.581      0.2320            0.2326
89        8.439      0.4273            0.4284
NA's      5.144      0.2604
Total  1975.001    100.0000          100.0000

Note: Table output edited to save space.
```

Figure 2.6 Ages of GSS Respondents



You can use the frequency distribution table and bar graph to identify the mode and median ages of GSS respondents, or use the wtd.mode and wtd.median functions introduced above.[11] (The modal age is 55 and median age is 45.)

Before we make further observations about the central tendency and dispersion of this variable, we will apply a different function, describe, to generate a bumper crop of information about this numeric variable. The generic syntax for the describe function is as follows:

```
describe(x, weights=optional.weight)
```

If you wish to include a weight variable, the argument, 'weights=', needs to be typed out.[12] To describe the age of GSS respondents, we would type:

```
describe(gss$age, weights=gss$wtss)    # Example, Descriptive Statistics
                                       # Describing Interval Variables
```

```
gss$age
       n       missing        unique          Info          Mean
  1969.9           5.1            72             1          46.1

     .05           .10           .25           .50           .75
    21.0          23.0          32.0          45.0          58.0

     .90           .95
    71.0          77.0

lowest: 18 19 20 21 22, highest: 85 86 87 88 89
```

The median is the 50th percentile, labeled ".50."

The top row of describe's output tells us the weighted numbers of valid cases and missing cases (1969.9 and 5.1, respectively), the number of unique values (72), and the mean age (46.1 years). Next, describe reports a series of percentiles. For example, the label, ".05," and its associated value of age, "21.0," tells us that 5 percent (.05) of the individuals in the survey are 21 years old or younger. The 50th percentile, labeled ".50", is the median age, 45.00. Half of the GSS respondents are younger than 45 and half are older than 45. Finally, the output of the describe function reports the five lowest and highest ages found in the GSS dataset.

To simply calculate an interval variable's mean value, without consulting all the summary information produced by the describe function, you can use the wtd.mean function. This function will yield the same mean value, but it is sometimes useful for a function to generate a result rather than extracting it from console output.

```
wtd.mean(gss$age, weights=gss$wtss)    # Describing Interval Variables
```

```
[1] 46.10235
```

[11] The frequency distribution table and bar graph produced from this sample code may represent the distribution in too much detail. It may be more useful to describe the relative distribution of different age groups in the GSS survey rather than break down each individual age. In Chapter 3, we'll discuss some techniques to create age groupings.

[12] To supply weights to the describe function, we need to use keyword matching. We cannot rely on positional matching because "weights" is not the second argument to this function as defined by the Hmisc package.

Describe is so meticulous in providing percentiles, the numbers permit us to determine the *interquartile range*, the values of a variable that bracket the "middle half" of a distribution, between the top of the lowest quartile (".25") and bottom of the highest quartile (".75"). For age, we can see that the middle half falls between 32 and 58 years of age. The interquartile range has limited analytic value for describing a single variable; however, interquartile ranges are quite useful when comparing two or more distributions. (This is illustrated in Chapter 4.)

We have discovered that the mean age, at 46.1, is higher than the median age of 45. What does this comparison tell us about the skewness of the distribution? When a distribution is perfectly symmetrical—no skew—its mean will be equal to its median. If the mean is lower than the median—that is, if a few extremely low values pull the mean down, away from the center of the distribution—the distribution has a negative skew.[13] If the mean is higher than the median, as is the case with our current analysis, the distribution has a positive skew.[14] The bar chart from the freq analysis (Figure 2.6) lends visual clarity. The skinnier right-hand tail is a tell-tale sign of positive skewness. Even so, the mean (46.1) and the median (45) are just over one year apart. In this case, it would not be a distortion of reality to use the mean instead of the median to describe the central tendency of the distribution.

DESCRIBING THE DISPERSION OF INTERVAL VARIABLES

Sometimes the mean value of an interval variable provides a misleading impression of a variable's typical value. To illustrate this point—and to introduce another useful graphic form—we will obtain descriptive statistics for a variable in the states dataset, hispanic10, the Hispanic percentage of each state's population (as of 2010). This time we will bypass freq and go directly to describe. (For unweighted data, like states or world, you might prefer R's summary function.)

```
describe(states$hispanic)          # Example, Descriptive Statistics
                                   # Describing Interval Variables
```

The states dataset does not need to be weighted, so simply omit the optional "weights=" argument.

```
states$hispanic10
       n  missing   unique     Info     Mean      .05      .10      .25      .50
      50        0       43        1    10.61    1.725    2.700    4.250    8.200
     .75      .90      .95
  12.225   22.900   34.000

lowest: 1.2 1.3 1.5 2.0 2.7, hightest: 22.5 26.5 29.6 37.6 46.3
```

The mean is equal to 10.61, much higher than the median, 8.200.

The mean percentage Hispanic, 10.61, is more than two units of measure higher than the median percentage Hispanic, 8.20, indicating a strong positive skew. The bottom row of describe's output provides a clue to the skew: The percentage of Hispanics in the five lowest-percentage states tops out at 2.7. The percentages of the five highest-percentage states range from 22.5 to 46.3. These high values pull the mean upward, off the median. In this case, the median, 8.20, is the more accurate measure of central tendency.[15]

What about graphic accompaniment for describe's numbers? We could ask freq (or freqC) for a bar chart, but because states$hispanic10 has so many unique values relative to the number of cases—according to describe,

---

[13] For a precise method of measuring the skew of a distribution, see the skewness function in the "moments" package.

[14] We don't observe a left-side tail of the age distribution because the GSS does not survey children.

[15] Many demographic variables are skewed, so their median values rather than their means are often used to give a clearer picture of central tendency. One hears or reads reports, for example, of median family income or the median price of homes in an area.