Statistics and Computing

Robert A. Muenchen

R for SAS and SPSS Users

Second Edition

Springer

Statistics and Computing

For further volumes:
http://www.springer.com/series/3022

Robert A. Muenchen

R for SAS and SPSS Users

Second Edition

Springer

Robert A. Muenchen
University of Tennessee
Research Computing Support
109 Hoskins Library
1400 W. Cumberland
Knoxville, TN 37996-4005
USA
muenchen.bob@gmail.com

ISSN 1431-8784
ISBN 978-1-4614-0684-6 e-ISBN 978-1-4614-0685-3
DOI 10.1007/978-1-4614-0685-3
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011933470

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

While SAS and SPSS have many things in common, R is very different. My goal in writing this book is to help you translate what you know about SAS or SPSS into a working knowledge of R as quickly and easily as possible. I point out how they differ using terminology with which you are familiar, and show you which add-on packages will provide results most like those from SAS or SPSS. I provide many example programs done in SAS, SPSS, and R so that you can see how they compare topic by topic.

When finished, you should know how to:

Install R, choose a user interface, and choose and install add-on packages.

Read data from various sources such as text or Excel files, SAS or SPSS data sets, or relational databases.

Manage your data through transformations, recodes, and combining data sets from both the add-cases and add-variables approaches and restructuring data from wide to long formats and vice versa.

Create publication-quality graphs including bar, histogram, pie, line, scatter, regression, box, error bar, and interaction plots.

Perform the basic types of analyses to measure strength of association and group differences, and be able to know where to turn to learn how to do more complex methods.

Who This Book Is For

This book teaches R requiring no prior knowledge of statistical software. However, you know SAS or SPSS this book will make learning R as easy as possible by using terms and concepts that you already know. If you do not know SAS or SPSS, then you will learn R along with how it compares to the two most popular commercial packages for data analysis. Stata users would be better off reading *R for Stata Users* [41].

An audience I did not expect to serve is R users wanting to learn SAS or SPSS. However, I have heard from quite a few of them who have said that by explaining the differences, it helped them learn in the reverse order I had anticipated. Keep in mind that I explain none of the SAS or SPSS programs, only the R ones and how the packages differ, so it is not ideal for that purpose.

Who This Book Is Not For

I make no effort to teach statistics or graphics. Although I briefly state the goal and assumptions of each analysis along with how to interpret their output, I do not cover their formulas or derivations. We have more than enough to discuss without tackling those topics too.

This is also not a book about writing complex R functions, it is about using the thousands that already exist. We will write only a few very short functions. If you want to learn more about writing functions, I recommend Jones et al.'s *Introduction to Scientific Programming and Simulation Using R* [31]. However, reading this book should ease your transition to more complex books like that one.

Practice Data Sets and Programs

All of the programs, data sets, and files that we use in this book are available for download at http://r4stats.com. A file containing corrections and clarifications is also available there.

Regarding the Second Edition

As the first edition went to press, I began planning the second edition with the main goal of adding more statistical methods. However, my readers quickly let me know that they needed far more information about the basics. There are many wonderful books devoted to statistics in R. I recommend some in Chap. 17. The enhancements to this edition include the following:

1. Programming code has been updated throughout.
2. It is easier to find reference material using the new list of tables and list of figures.
3. It is easier to find topics using the index, which now has four times as many entries.
4. The glossary defines more R terms.
5. There is a new Sect. 3.6, "Running R in SAS and WPS," including *A Bridge to R* and *IML Studio*.
6. There is a new Sect. 3.9, "Running R from within Text Editors."

7. There is a new Sect. 3.8, "Running R in Excel," complete with R Commander menus.
8. There is a new Sect. 3.10 on integrated development environments, including RStudio.
9. There is a new Sect. 3.11.1 on the `Deducer` user interface and its Plot Builder (similar to IBM SPSS Visualization Designer).
10. New Sect. 3.11.4 on Red-R, a flowchart user interface like SAS Enterprise Miner or IBM SPSS Modeler (Clementine).
11. Chapter 5, "Programming Language Basics," has been significantly enhanced, including additional examples and explanations.
12. There is a new Sect. 5.3.4 on matrix algebra with table of basic matrix algebra functions.
13. There is a new Sect. 5.6, "Comments to Document Your Objects."
14. Chapter 6, "Data Acquisition," includes improved examples of reading SAS and SPSS data files.
15. There is a new Sect. 6.2.3, "Reading Text from a Web Site."
16. There is a new Sect. 6.2.4, "Reading Text from the Clipboard."
17. There is a new Sect. 6.2.6, "Trouble with Tabs," on common problems when reading tab-delimited files.
18. Section 6.3, "Reading Text Data Within a Program," now includes a simpler approach using the `stdin` function.
19. There is a new Sect. 6.4 "Reading Multiple Observations per Line."
20. There are new sections on reading/writing Excel files.
21. There is a new Sect. 6.9, "Reading Data from Relational Databases.
22. There is a new Sect. 7.11.1, "Selecting Numeric or Character Variables," (like VAR A-numeric-Z; or A-character-Z).
23. There is a new Sect. 8.4, "Selecting Observations using Random Sampling."
24. Chapter 9, "Selecting Variables and Observations," has many more examples, and they are presented in order from most widely used to least.
25. There is a new Table 10.2, "Basic Statistical Functions."
26. There is a new Sect. 10.2.3 "Standardizing and Ranking Variables."
27. Section 10.14, "Removing Duplicate Observations," now includes an example for eliminating observations that are duplicates only on key variables (i.e., PROC SORT NODUPKEY).
28. There is a new Sect. 10.16, "Transposing or Flipping Data Sets" (tricky with character variables).
29. There is a new Sect. 10.20, "Character String Manipulations," using the `stringr` package.
30. There is a new Sect. 10.21, "Dates and Times," which covers date/time manipulations using the `lubridate` package.
31. The new Chap. 11, "Enhancing Your Output," covers how to get publication quality tables from R into word processors, Web pages or LaTeX.
32. The new Sect. 12.4, "Generating Values for Reading Fixed-Width Files," shows how to generate repetitive patterns of variable names and matching widths for reading complex text files.

33. There is a new Sect. 16.15, which shows how to make geographic maps.
34. There is a new Sect. 17.11 “Sign Test: Paired Groups.”
35. Appendix B, ”A Comparison of SAS and SPSS Products with R Packages and Functions,” is now far more comprehensive and changes so frequently that I have moved it from the appendix to http://r4stats.com.

Acknowledgments

I am very grateful for the many people who have helped make this book possible, including the developers of the S language on which R is based, John Chambers, Douglas Bates, Rick Becker, Bill Cleveland, Trevor Hastie, Daryl Pregibon and Allan Wilks; the people who started R itself, Ross Ihaka and Robert Gentleman; the many other R developers for providing such wonderful tools for free and all of the R-help participants who have kindly answered so many questions. Most of the examples I present here are modestly tweaked versions of countless posts to the R-help discussion list, as well as a few SAS-L and SPSSX-L posts. All I add is the selection, organization, explanation, and comparison to similar SAS and SPSS programs.

I am especially grateful to the people who provided advice, caught typos, and suggested improvements, including Raymond R. Balise, Patrick Burns, Glenn Corey, Peter Flom, Chun Huang, Richard Gold, Martin Gregory, Warren Lambert, Matthew Marler, Paul Miller, Ralph O’Brien, Wayne Richter, Denis Shah, Charilaos Skiadas, Andreas Stefik, Phil Spector, Joseph Voelkel, Michael Wexler, Graham Williams, Andrew Yee, and several anonymous reviewers.

My special thanks go to Hadley Wickham, who provided much guidance on his `ggplot2` graphics package, as well as a few of his other handy packages. Thanks to Gabor Grothendieck, Lauri Nikkinen, and Marc Schwarz for the R-Help discussion that led to Sect. 10.15: “Selecting First or Last Observations per Group.” Thanks to Gabor Grothendieck also for a detailed discussion that lead to Sect. 10.4, “Multiple Conditional Transformations.” Thanks to Garrett Grolemund for his help in understanding dates, times and his time-saving `lubridate` package. Thanks to Frank Harrell, Jr. for helping me elucidate the discussion of object orientation in final chapter.

I also thank SPSS, Inc. especially Jon Peck, for the helpful review of this book and Jon’s SPSS expertise, which benefited several areas including the programs for extracting the first/last observation per group, formatting date–time variables, and generating data. He not only improved quite a few of the SPSS programs, but found ways to improve several of the R ones as well!

At The University of Tennessee, I am thankful for the many faculty, staff, and students who have challenged me to improve my teaching and data analysis skills. My colleagues Michael Newman, Michael O’Neil, Virginia Patterson, Ann Reed, Sue Smith, Cary Springer, and James Schmidhammer have been

a source of much assistance and inspiration. Michael McGuire, provided assistance with all things Macintosh.

Finally, I am grateful to my wife, Carla Foust, and sons Alexander and Conor, who put up with many lost weekends while I wrote this book.

Robert A. Muenchen
muenchen.bob@gmail.com
Knoxville, Tennessee

About the Author

Robert A. Muenchen is a consulting statistician and, with Joseph Hilbe, author of the book *R for Stata Users* [41]. He is currently the manager of Research Computing Support (formerly the Statistical Consulting Center) at the University of Tennessee. Bob has conducted research for a variety of public and private organizations and has coauthored over 50 articles in scientific journals and conference proceedings.

Bob has served on the advisory boards of the SAS Institute, SPSS, Inc. the Statistical Graphics Corporation, and *PC Week Magazine*. His suggested improvements have been incorporated into SAS, SPSS, JMP, STATGRAPHICS, and several R packages.

His research interests include statistical computing, data graphics and visualization, text analysis, data mining, psychometrics, and resampling.

Linux is the registered trademark of Linus Torvalds.
MATLAB is a registered trademark of The Mathworks, Inc.
Macintosh and Mac OS are registered trademarks of Apple, Inc.
Oracle and Oracle Data Mining are registered trademarks of Oracle, Inc.
R-PLUS is a registered trademark of XL-Solutions, Inc.
RStudio is a registered trademark of RStudio, Inc.
Revolution R and Revolution R Enterprise are registered trademarks of Revolution Analytics, Inc.
SAS , SAS , AppDev Studio , SAS Enterprise Guide , SAS Enterprise Miner , and SAS/IML Studio are registered trademarks of the SAS Institute.
SPSS , IBM SPSS Statistics , IBM SPSS Modeler , IBM SPSS Visualization Designer , and Clementine , are registered trademarks of SPSS, Inc., an IBM company.

Stata is a registered trademark of Statacorp, Inc.
Tibco Spotfire S+ is a registered trademark of Tibco, Inc.
UNIX is a registered trademark of The Open Group.
Windows , Windows Vista , Windows XP , Windows XP , Excel , and Microsoft Word are registered trademarks of Microsoft, Inc.
World Programming System and WPS are registered trademarks of World Programming, Ltd.

Contents

List of Tables

List of Figures

1

Introduction

1.1 Overview

Norman Nie, one of the founders of SPSS, calls R [55] "The most powerful statistical computing language on the planet."[1] Written by Ross Ihaka, Robert Gentleman, the R Core Development Team, and an army of volunteers, R provides both a language and a vast array of analytical and graphical procedures. The fact that this level of power is available free of charge has dramatically changed the landscape of research software.

R is a variation of the S language, developed by John Chambers with substantial input from Douglas Bates, Rick Becker, Bill Cleveland, Trevor Hastie, Daryl Pregibon, and Allan Wilks.[2] The Association of Computing Machinery presented John Chambers with a Software System Award and said that the S language, "*. . . will forever alter the way people analyze, visualize, and manipulate data. . .*" and went on to say that it is, "*. . . an elegant, widely accepted, and enduring software system, with conceptual integrity. . . .*" The original S language is still commercially available as Tibco Spotfire S+. Most programs written in the S language will run in R.

The SAS Institute, IBM's SPSS Company, and other vendors are helping their customers extend the power of their software through R. They have added interfaces that allow you to use R functions from within their programs, expanding their capabilities. You can now blend SAS or IBM SPSS Statistics (hereafter referred to as simply SPSS) code with R, easily transferring data and output back and forth.

SAS and SPSS are so similar to each other that moving from one to the other is straightforward. R, however, is very different, making the transition confusing at first. I hope to ease that confusion by focusing on the similarities

[1] He said this after moving to Revolution Analytics, a company that sells a version of R.

[2] For a fascinating history of S and R, see Appendix A of *Software for Data Analysis: Programming with R* [12].

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_1, © Springer Science+Business Media, LLC 2011

and differences in this book. When we examine a particular analysis by, say, comparing two groups with a t-test, someone who knows SAS or SPSS will have very little trouble figuring out what R is doing. However, the basics of the R language are very different, so that is where we will spend most of our time.

For each aspect of R we discuss, I will compare and contrast it with SAS and SPSS. Many of the topics end with example programs that do almost identical things in all three. The R programs often display more variations on each theme than do the SAS or SPSS examples, making the R programs longer.

I introduce topics in a carefully chosen order, so it is best to read from beginning to end the first time through, even if you think you do not need to know a particular topic. Later you can skip directly to the section you need. I include a fair amount of redundancy on key topics to help teach those topics and to make it easier to read just one section as a future reference. The glossary in Appendix A defines R concepts in terms that SAS or SPSS users will understand, and provides parallel definitions using R terminology.

1.2 Why Learn R?

If you already know SAS or SPSS, why should you bother to learn R? Both SAS and SPSS are excellent packages for analyzing data. I use them both several times a week. However, R has many benefits:

R offers a vast array of analytical methods. There are several thousand add-on packages available for R on the Internet, and you can easily download and install them within R itself.

R offers new methods sooner. Since people who develop new analytic methods often program in R, you often get access to them years before the methods are added to SAS or SPSS.

Many analytic packages can run R programs. These include: SAS, SPSS, Excel, JMP, Oracle Data Mining, Statistica, StatExact, and others. This provides you the option of using R functions without having to learn its entire language. You can do all your data management in your preferred software, and call the R functions you need from within it.

R is rapidly becoming a universal language for data analysis. Books and journals frequently use R for their examples because they know everyone can run them. As a result, understanding R is important for your continuing education. It also allows you to communicate your analytic ideas with a wide range of colleagues.

R's graphics are extremely flexible and are of publication quality. They are flexible enough to overlay data from different data sets, even at different levels of aggregation. You are even free to completely replace R's graphics subsystem, as people have already done.

R is very flexible in the type of data it can analyze. While SAS and SPSS require you to store your data in rectangular data sets, R offers a rich variety of data structures that are much more flexible. You can perform analyses that include variables from different data structures easily without having to merge them.

R has object oriented abilities. This provides many advantages including an ability to "do the right thing." For example, when using a categorical variable as a predictor in a linear regression analysis, it will automatically take the proper statistical approach.

If you like to develop your own analytic methods, you'll find much to like in the power of R's language. The vast array of add-ons for R demonstrates that people who like to develop new methods like working in R.

R's procedures, called *functions*, are open for you to see and modify. This makes it easier to understand what it is doing. Copying an existing function and then modifying it is a common way to begin writing your own function.

Functions that you write in R are automatically on an equal footing with those that come with the software. The ability to write your own completely integrated procedures in SAS or SPSS requires using a different language such as C or Python and, in the case of SAS, a developer's kit.

R has comprehensive matrix algebra capabilities similar to those in MATLAB. It even offers a MATLAB emulation package [48].

R runs on almost any computer, including Windows, Macintosh, Linux, and UNIX.

R is free. This has an obvious appeal to corporate users. Even academics who purchase software at substantial discounts for teaching and internal use will appreciate the fact that they can consult with outside organizations without having to purchase a commercial license.

1.3 Is R Accurate?

When people first learn of R, one of their first questions is, "Can a package written by volunteers be as accurate as one written by a large corporation?" Just as with SAS and SPSS, the development of the main R package, referred to as *Base R plus Recommended Packages*, is handled with great care. This includes levels of beta testing and running validation suites to ensure accurate answers. When you install R, you can ask it to install the *Test Files*, which includes the `tools` package and a set of validation programs. See the *R Installation and Administration Manual* [56] on the R help menu for details.

The various quality assurance steps used with each version of R are outlined in [19]. R's Base and Recommended Packages currently consist of the following packages: `base`, `boot`, `class`, `cluster`, `codetools`, `datasets`, `foreign`, `graphics`, `grDevices`, `grid`, `KernSmooth`, `lattice`, `MASS`, `methods`, `mgcv`, `nlme`, `nnet`, `rpart`, `spatial`, `splines`, `stats`, `stats4`, `survival`, `tcltk`, `tools` and `utils`. Those packages, and the functions they contain, are roughly the equivalent to Base SAS, SAS/GRAPH, SAS/STAT and SAS/IML. Compared to SPSS products, they cover similar territory as IBM SPSS Statistics Base, IBM SPSS Advanced Statistics, and IBM SPSS Regression. The help files show in which each package each function resides.

Just as with SAS or SPSS programs or macros that you find on the Internet, R's add-on packages may or may not have been put through rigorous testing. They are often written by the university professors who invented the methods the packages implement. In that case, the work has usually passed the academic journal peer-review process with three experts in the field checking the work. However, a package could have been written by some poor programmer who just learned R.

One way you can estimate the quality of a given package is to see how people rate it at http://crantastic.org. You can also search the R-help archives to see what people are saying about a package that interests you. For details on R-help, see Chap. 4, ”Help and Documentation”.

It is to their credit that the SAS Institute and SPSS, Inc. post databases of known bugs on their Web sites, and they usually fix problems quickly. R also has open discussions of its known bugs and R's developers fix them quickly, too. However, software of this complexity will never be completely free of errors, regardless of its source.

The most comprehensive study of R's accuracy to date was done by Keeling and Pavur [33]. They compared nine statistics packages on the accuracy of their univariate statistics, analysis of variance, linear regression, and nonlinear regression. The accuracy of R was comparable to SAS and SPSS and, by the time the article was published, Bolker [9] found that R's accuracy had already improved.

Another study by Almiron et al. [1] replicated the Keeling and Pavur results, verified that R's accuracy had improved, and found R to be more accurate than several other open source packages.

1.4 What About Tech Support?

When you buy software from SAS or SPSS, you can call or e-mail for tech support that is quick, polite, and accurate. Their knowledgeable consultants have helped me out of many a jam.

If you use the free version of R, you do not get a number to call, but you do get direct access to the people who wrote the program and others who know it well via e-mail. They usually answer your question in less than an hour. Since they are scattered around the world, that support is around the clock.

The main difference is that the SAS or SPSS consultants will typically provide a single solution that they consider best, while the R-help list responders will often provide several ways to solve your problem. You learn more that way, but the solutions can vary quite a bit in level of difficulty. However, by the time you finish this book, that should not be a problem. For details on the various R e-mail support lists, see Chap. 4, “Help and Documentation.”

There are companies that provide various types of support for a fee. Examples of such organizations are Revolution Analytics, Inc., RStudio, Inc., and XL-Solutions Corporation.

1.5 Getting Started Quickly

If you wish to start using R quickly, you can do so by reading fewer than fifty pages of this book. Since you have SAS, the SAS-compatible World Programming System (WPS), or SPSS to do your basic descriptive statistics, you are likely to need R’s modeling functions. Here are the steps you can follow to use them.

1. Read the remainder of this chapter and Chap. 2, “Installing and Updating R.” Download and install R on your computer.
2. Read the part of Chap. 3, “Running R,” that covers your operating system and running R from within either SAS, WPS or SPSS.
3. In Chap. 5, “Programming Language Basics,” read Sect. 5.3.2 about factors, and Sect. 5.3.3 about data frames.
4. Also in Chap. 5, read Sect. 5.7.1, “Controlling Functions with Arguments,” and Sect. 5.7.3, “Controlling Functions with Formulas,” including Table 17.1, “Example formulas in SAS, SPSS, and R.”
5. If you do not have SAS/IML Studio, or MineQuest’s A Bridge to R, read Sect. 6.10, “Reading Data from SAS.”

After reading the pages above, do all your data management in SAS, WPS or SPSS, stripping out observations containing any missing values. Then save the data to a new file to pass to SAS or SPSS using their internal links to R. Assuming your variables are named y, x1, x2, . . . , your entire R program will look something like this:

```
library("TheLibraryYouNeed")  # If you need any.
mymodel <- TheFunctionYouNeed(y ~ x1 + x2, data = mydata)
summary(mymodel)
plot(mymodel)  # If your function does plots.
```

We will discuss what these commands mean shortly. The ones that begin with “#” are comments.

1.6 The Five Main Parts of SAS and SPSS

While SAS and SPSS offer hundreds of functions and procedures, they fall into five main categories:

1. Data input and management statements that help you read, transform, and organize your data.
2. Statistical and graphical procedures to help you analyze data. You could certainly consider these two as separate categories, but they share a similar syntax and you can use them in the same parts of programs;
3. An output management system to help you extract output from statistical procedures for processing in other procedures or to let you customize printed output. SAS calls theirs the Output Delivery System (ODS) and SPSS calls theirs the Output Management System (OMS);
4. A macro language to help you use sets of the above commands repeatedly;
5. A matrix language to add new algorithms (SAS/IML and SPSS Matrix).

SAS and SPSS handle each of these five areas with different systems that follow different rules. For simplicity's sake, introductory training in SAS or SPSS typically focuses on only the first two topics. Perhaps the majority of users never learn the more advanced topics. However, R performs these five functions in a way that completely integrates them all. The integration of these five areas gives R a significant advantage in power and is the reason that most R developers write procedures using the R language.

While we will focus on topics 1 and 2 when discussing SAS and SPSS, we will discuss some of all five regarding R. Since SAS and SPSS procedures tend to print all of their output at once, a relatively small percentage of their users take advantage of their output management systems. Virtually all R users use output management. That is partly because R shows you only the pieces of output you request, and partly because R's output management is easier to use. For example, you can create and store a linear regression model using the `lm` function.

```
myModel <- lm(y ~ x)
```

You can then get several diagnostic plots with the `plot` function.

```
plot(myModel)
```

You can compare two models using the `anova` function.

```
anova(myModel1, myModel2)
```

That is a very flexible approach! It requires fewer commands than SAS or SPSS and it requires almost no knowledge of how the model is stored. The `plot` and `anova` functions have a built-in ability to work with models and other data structures.

The price R pays for this output management advantage is that the output to most procedures is sparse and does not appear as publication quality within R itself. It appears in a monospace font without a word-processor-style table structure or even tabs between columns. Variable labels are not a part of the core system, so if you want clarifying labels, you add them in other steps. You can use functions from add-on packages to write out HTML, ODF, or

LaTeX files to use in word processing tools. SPSS and, more recently, SAS make output that is publication quality by default, but not as easy to use as input to further analyses.

On the topic of matrix languages, SAS and SPSS offer them in a form that differs sharply from their main languages. For example, the way you select variables in the main SAS product bears no relation to how you select them in SAS/IML. In R, the matrix capabilities are completely integrated and follow the same rules.

1.7 Our Practice Data Sets

Throughout much of this book we will use a small artificial data set named *mydata*. This allows me to show you how to enter and manipulate it in many ways without much work on your part. The data set is a pretend survey of students who attended some workshops to learn statistical software. It records which workshop they took, their gender, and their responses to four questions:

q1 – The instructor was well prepared.
q2 – The instructor communicated well.
q3 – The course materials were helpful.
q4 – Overall, I found this workshop useful.

The values for the workshops are 1, 2, 3, and 4 for R, SAS, SPSS, and Stata respectively. In the smallest form of these data, only the R and SAS workshops appear. Here is mydata:

```
  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2     NA  3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

The letters “NA” stand for Not Available, or missing.

In Chap. 5 we will create various small R objects. They are all stored in a file named myall.RData.

When we study missing data, we will use a version of these data named *mydataNA*. That file contains many missing values coded in different ways.

For examples that require more data the data set *mydata100* has 100 observations in the same form, plus two additional variables, pretest and posttest. A version of this data set that adds variable labels is *mydata100L*, with the “L” standing “labeled.”

Here are the first few observations from mydata100:

```
  workshop gender q1 q2 q3 q4 pretest posttest
1        R Female  4  3  4  5      72       80
2     SPSS   Male  3  4  3  4      70       75
3     <NA>   <NA>  3  2 NA  3      74       78
4     SPSS Female  5  4  5  3      80       82
5    Stata Female  4  4  3  4      75       81
6     SPSS Female  5  4  3  5      72       77
```

We will occasionally treat the survey questions as interval-level data, which is a bit of a stretch. In a more realistic setting, we would have several items for each topic and we would create mean scores containing many more values than simply 1, 2, 3, 4, and 5.

Finally, when learning to read and manipulate dates and character strings, we will use a very small data file containing some famous people from the field of statistics:

```
                born,        died
R.A. Fisher,    2/17/1890,  7/29/1962
Carl Pearson,   3/27/1857,  4/27/1936
Gertrude Cox,   1/13/1900, 10/17/1978
John Tukey,     6/16/1915,  7/26/2000
William Gosset, 6/13/1876, 10/16/1937
```

1.8 Programming Conventions

The example programs are set to look for their matching data files in a folder named *myRfolder*, but that is easy to change to whatever location you prefer. Each program begins by loading the data as if it were a new session. That is not required if you already have the data loaded, but it makes it easier to ensure that previous programming does not interfere with the example. It also allows each program to run on its own.

Each example program in this book begins with a comment stating its purpose and the name of the file it is stored in. For example, each of the programs for selecting variables begin with a comment like the following.

```
# R Program for Selecting Variables.
# Filename: SelectingVars.R
```

R uses the "#" symbol at the beginning of comments used to document programs. The filename in the practice files will always match, so the three files for this topic are *SelectingVars.sas*, *SelectingVars.sps*, and *SelectingVars.R*. Each R data object in this book is available in a single file. Its name is the same as is used in the book, with the extension ".RData." For example, our most widely used data object, mydata, is stored in *mydata.RData.* Also, the objects we create and use frequently, data and functions, are all stored in *myWorkspace.RData*.

1.9 Typographic Conventions

All programming code and the names of all R functions and packages are written in `this Courier font`. The names of other documents and menus is in *this italic font*. Menus appear in the form *File> Save as*, which means "choose *Save as* from the *File* menu."

When learning a new language, it can be hard to tell the commands from the names you can choose (e.g., variable or data set names). To help differentiate, I CAPITALIZE statements in SAS and SPSS and use lowercase for names that you can choose. However, R is case-sensitive, so I have to use the exact case that the program requires. Therefore, to help differentiate, I use the common prefix "my" in names like *mydata* or *mySubset*.

R uses "`>`" to prompt you to input a new line and "`+`" to prompt you to enter a continued line. When there is no output to see, I delete the prompt characters to reduce clutter. However, when examples include both input and output, I leave the input prompts in place. That helps you identify which is which. So the first three lines below are the input I submitted and the last line is the mean that R wrote out.

```
> q1 <- c(1, 2, 2, 3,
+         4, 5, 5, 5, 4)

> mean(q1)

[1] 3.4444
```

R tends to pack its input and different sections of output tightly together. This makes it harder to read when you are learning it. Therefore, I also add spacing in some places to improve legibility. In the example above, I added a blank line on either side of the line containing "`> mean(q1)`".

2

Installing and Updating R

When you purchase SAS, WPS or SPSS, they sell you a "binary" version. That is one that the company has compiled for you from the "source code" version they wrote using languages such as C, FORTRAN, or Java. You usually install everything you purchased at once and do not give it a second thought. Instead, R is modular. The main installation provides Base R and a recommended set of add-on modules called packages. You can install other packages later when you need them. With thousands to choose from, few people need them all.

To download R itself, go to the Comprehensive R Archive Network (CRAN) at `http://cran.r-project.org/`. Choose your operating system under the web page heading, *Download and Install R*. The binary versions install quickly and easily. Binary versions exist for many operating systems including Windows, Mac OS X, and popular versions of Linux such as Ubuntu, RedHat, Suse, and others that use either the RPM or APT installers.

Since R is an Open Source project, there are also source code versions of R for experienced programmers who prefer to compile their own copy. Using that version, you can modify R in any way you like. Although R's developers write many of its analytic procedures (or at least parts of them) using the R language, they use other languages such as C and FORTRAN to write R's most fundamental functions.

Each version of R installs into its own directory (folder), so there is no problem having multiple versions installed on your computer. You can then install your favorite add-on packages for the new release.

2.1 Installing Add-on Packages

While the main installation of R contains many useful functions, many additional packages, written by R users, are available on the Internet. The main site for additional packages is at the CRAN web site under *Packages*. The section labeled *Task Views* organizes packages by task, such as Bayesian, Cluster Analysis, Distribution, Econometrics, and so on. While CRAN is a good place

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_2, © Springer Science+Business Media, LLC 2011

to read about and choose packages to install, you usually do not need to download them from there yourself. As you will see, R automates the download and installation process. A comparison of SAS and SPSS add-ons to R packages is presented at this book's web site, http://www.r4stats.com. Another useful site helps you to find useful packages and write reviews of packages you like: Crantastic at http://crantastic.org/.

Before installing packages, your computer account should have administrative privileges and you must start R in a manner that allows administrative control. If you do not have administrative privileges on your computer, you can install packages to a directory to which you have write access. For instructions, see the FAQ (*F*requently *A*sked *Q*uestions) at http://www.r-project.org/.

To start R with administrative control on Windows Vista or later, right-click its menu choice and then choose *Run as administrator*. Window's User Account Control will then ask for your permission to allow R to modify your computer.

On the R version for Microsoft Windows, you can choose *Packages> Install package(s)* from the menus. It will ask you to choose a CRAN site or "mirror" that is close you:

```
CRAN mirror
  Australia
  Austria
  Belgium
  Brazil (PR)
  ...
  USA (TX 2)
  USA (WA)
```

Then it will ask which package you wish to install:

```
Packages
  abc
  abd
  abind
  AcceptanceSampling
  ...
  zipcode
  zoo
  zyp
```

Choose one of each and click OK.

If you prefer to use a function instead of the menus, you can use the `install.packages` function. For example, to download and install Frank Harrell's `Hmisc` package [32], start R and enter the command:

```
install.packages("Hmisc")
```

R will then prompt you to choose the closest mirror site and the package you need. If you are using a graphical user interface (GUI), you click on your choice, then click *OK*. If not, R will number them for you and you enter the number of the mirror.

A common error is to forget the quotes around the package name:

```
> install.packages(Hmisc)  # Quotes are missing!

Error in install.packages(Hmisc) : object  Hmisc  not found
```

Older versions of R also required the argument `dependencies = TRUE`, which tells R to also install any packages that this package "depends" on and those that its author "suggests" as useful. That is now the default setting and so it is usually best to avoid adding that. However, a few packages still require that setting. The best known of these packages is Fox's R Commander user interface. So you would install it using:

```
install.packages("Rcmdr", dependencies = TRUE)
```

After a package is installed, you can find out how to cite it using the `citation` function. Note that you call this function with the package name in quotes:

```
> citation("Rcmdr")

To cite package  Rcmdr  in publications use:

  John Fox <jfox@mcmaster.ca>, with
  contributions from ...(2010). Rcmdr: R
  Commander. R package version 1.6-2.
  http://CRAN.R-project.org/package=Rcmdr

A BibTeX entry for LaTeX users is

  @Manual{,
    title = {Rcmdr: R Commander},
    author = {John Fox and with contributions from ...
```

If you use simply `citation()` it will tell you how to cite R itself.

2.2 Loading an Add-on Package

Once installed, a package is on your computer's hard drive in an area called your *library*. However, it is not quite ready to use. Each time you start R, you also have to *load* the package from your library into your computer's main memory before you can use it. The reason for this additional step is twofold. It makes efficient use of your computer's memory and it keeps different packages conflicting with each other, or with base R functions. You can see what packages are installed and ready to load with the library function:

```
> library()

R Packages available
Packages in library  C:/PROGRA~1/R/R-212~1.1/library :

anchors   Statistical analysis of surveys with...
arules    Mining Association Rules and Frequent Itemsets
base      The R Base Package
...
xtable    Export tables to LaTeX or HTML
xts       Extensible Time Series
Zelig     Everyone s Statistical Software
```

If you have just installed R, this command will show you the *Base and Recommended Packages*. They are the ones that are thoroughly tested by the R Core Team. The similar installed.packages function lists your installed packages along with the version and location of each.

You can load a package you need with the menu selection, *Packages> Load packages*. It will show you the names of all packages that you have installed but have not yet loaded. You can then choose one from the list.

Alternatively, you can use the library function. Here I am loading the Hmisc package. Since the Linux version lacks menus, this function is the only way to load packages.

```
library("Hmisc")
```

With the library function, the quotes around the package name are optional and are not usually used. However, other commands that refer to package names – such as install.packages – require them.

Many packages load without any messages; you will just see the ">" prompt again. When trying to load a package, you may see the error message below. It means you have either mistyped the package name (remember capitalization is important) or you have not installed the package before trying to load it. In this case, Lemon and Grosjean's prettyR [38] package name is typed accurately, so I have not yet installed it.

```
> library("prettyR")

Error in library("prettyR") :
  there is no package called  prettyR
```

To see what packages you have loaded, use the search function.

```
> search()

 [1] ".GlobalEnv"        "package:Hmisc"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

We will discuss this function in detail in Chapter 13, "Managing Your Files and Workspace."

Since there are so many packages written by users, two packages will occasionally have functions with the same name. That can be very confusing until you realize what is happening. For example, the `Hmisc` and `prettyR` packages both have a `describe` function that does similar things. In such a case, the package you load last will *mask* the function(s) in the package you loaded earlier. For example, I loaded the `Hmisc` package first, and now I am loading the `prettyR` package (having installed it in the meantime). The following message results:

```
> library("prettyR")

Attaching package:  prettyR
  The following object(s) are masked from package:Hmisc :
     describe
```

Since people usually want to use the functions in the package they loaded most recently, this is rarely a problem. However, if warnings like these bother you, you can avoid them by detaching each package as soon as you are done using it by using the `detach` function. For details, see Section 13.4, "Loading Packages."

If your favorite packages do not conflict with one anther, you can have R load them each time you start R by putting the commands in a file named *".Rprofile"*. That file can automate your settings just like the *autoexec.sas* file for SAS. For details, see Appendix C.

2.3 Updating Your Installation

Keeping your add-on packages current is very easy. You simply use the `update.packages` function.

```
> update.packages()

graph :
 Version 1.15.6 installed in C:/PROGRA~1/R/R-26~1.1/library
 Version 1.16.1 available at
   http://rh-mirror.linux.iastate.edu/CRAN

Update (y/N/c)?  y
```

R will ask you if you want to update each package. That can get tedious if you have a lot of packages to install. You can avoid that starting the update process with:

```
update.packages(ask = FALSE)
```

If you enter "y," it will do it and show you the following. This message, repeated for each package, tells you what file it is getting from the mirror you requested (Iowa State) and where it placed the file.

```
trying URL http://rh-mirror.linux.iastate.edu
  /CRAN/bin/windows/contrib/2.6/graph_1.16.1.zip
Content type application/zip length 870777 bytes (850 Kb)
opened URL
downloaded 850 Kb
```

This next message tells you that the file was checked for errors (its sums were checked) and it says where it stored the file. As long as you see no error messages, the update is complete.

```
package  graph  successfully unpacked and MD5 sums checked

The downloaded packages are in
        C:/Documents and Settings/muenchen/Local Settings/
   Temp/Rtmpgf4C4B/downloaded_packages
updating HTML package descriptions
```

Moving to a whole new version of R is not as easy. First, you download and install the new version just like you did the first one. Multiple versions can coexist on the same computer. You can even run them at the same time if you wanted to compare results across versions. When you install a new version of R, I recommend also installing your add-on packages again. There are ways to point your new version to the older set of packages, I find them more trouble than they are worth. You can reinstall your packages in the step-by-step fashion discussed previously. An easier way is to define a character variable like "myPackages" that contains the names of the packages you use. The following is an example that uses this approach to install most of the packages we use in this book[1].

```
myPackages <- c("car", "hexbin", "Hmisc", "ggplot2",
  "gmodels", "gplots", "reshape2", "prettyR", "xtable")
```

[1] R Commander is left out since it requires `dependencies = TRUE`.

```
install.packages(myPackages)
```

We will discuss the details of the `c` function used above later. We will also discuss how to store programs like this so you can open and execute them again in the future. While this example makes it clear what we are storing in myPackages, a shortcut to creating it is to use the `installed.packages` function:

```
myPackages <- row.names( installed.packages() )
```

You can automate the creation of myPackages (or whatever name you choose to store your package names) by placing either code example that defines it in your .Rprofile. Putting it there will ensure that myPackages is defined every time you start R. As you find new packages to install, you can add to the definition of myPackages. Then installing all of them when a new version of R comes out is easy. Of course, you do not want to place the `install.packages` function into your .Rprofile. There is no point in installing package every time you start R! For details, see Appendix C.

2.4 Uninstalling R

When you get a new version of any software package, it is good to keep the old one around for a while in case any bugs show up in the new one. Once you are confident that you will no longer need an older version of R, you can remove it.

In Microsoft Window, uninstall it in the usual way using *Start> Control Panel*, then *Programs and Features*. To uninstall R on the Macintosh, simply drag the application to the trash. Linux users should use their distribution's package manager to uninstall R.

2.5 Uninstalling a Package

Since uninstalling R itself also removes any packages in your library, it is rarely necessary to uninstall packages separately. However, it is occasionally necessary. You can uninstall a package using the `uninstall.packages` function. First, though, you must make sure it is not in use by detaching it. For example, to remove just the `Hmisc` package, use the following code:

```
detach("package:Hmisc")  # If it is loaded.

remove.packages("Hmisc")
```

2.6 Choosing Repositories

While most R packages are stored at the CRAN site, there are other repositories. If the Packages window does not list the one you need, you may need to choose another repository. The *Omegahat Project for Statistical Computing* [59] at `http://www.omegahat.org/` and *R-Forge* [61] at `http://r-forge.r-project.org/` are repositories similar to CRAN that have a variety of different packages available. There are also several repositories associated with the *BioConductor project*. As they say at their main web site, `http://www.bioconductor.org/`, "BioConductor is an open source and open development software project for the analysis and comprehension of genomic data" [23].

To choose your repositories, choose *Packages> Select repositories...* and the *Repositories* window will appear:

```
Repositories
  CRAN
  CRAN (extras)
  Omegahat
  BioC software
  BioC annotation
  BioC experiment
  BioC extra
  R-Forge
  rforge.net
```

The two CRAN repositories are already set by default. Your operating system's common mouse commands work as usual to make contiguous or noncontiguous selections. In Microsoft Window, that is Shift-click and Ctrl-click, respectively.

You can also select repositories using the `setRepositories` function:

```
> setRepositories()
```

If you are using a GUI the result will be the same. If you are instead working without a graphical user interface, R will number the repositories and prompt you to enter the number(s) of those you need.

2.7 Accessing Data in Packages

You can get a list of data sets available in each loaded package with the `data` function. A window listing the default data sets will appear:

```
> data()
```

```
R data sets
```

```
Data sets in package datasets :

  AirPassengers  Monthly Airline Passenger Numbers 1949-1960
  BJsales        Sales Data with Leading Indicator
  CO2            Carbon Dioxide Uptake in Grass Plants
  ...
  volcano        Topographic Information on Auckland s...
  warpbreaks     The Number of Breaks in Yarn during Weaving
  women          Average Heights and Weights for American Women
```

You can usually use these practice data sets directly. For example, to look at the top of the CO2 file (capital letters C and O, not zero!), you can use the `head` function:

```
> head(CO2)

  Plant   Type  Treatment conc uptake
1   Qn1 Quebec nonchilled   95   16.0
2   Qn1 Quebec nonchilled  175   30.4
3   Qn1 Quebec nonchilled  250   34.8
4   Qn1 Quebec nonchilled  350   37.2
5   Qn1 Quebec nonchilled  500   35.3
6   Qn1 Quebec nonchilled  675   39.2
```

The similar `tail` function shows you the bottom few observations.

Not all packages load their example data sets when you load the packages. If you see that a package includes a data set, but you cannot access it after loading the package, try loading it specifically using the `data` function. For example:

```
data(CO2)
```

If you only want a list of data sets in a particular package, you can use the `package` argument. For example, if you have installed the `car` package [21] (from Fox's *C*ompanion to *A*pplied *R*egression), you can load it from the library and see the data sets only it has using the following statements:

```
> library("car")

> data(package = "car")

Data sets in package car :
  AMSsurvey  American Math Society Survey Data
  Adler      Experimenter Expectations
  Angell     Moral Integration of American Cities
  Anscombe   U. S. State Public-School Expenditures
  Baumann    Methods of Teaching Reading Comprehension
  Bfox       Canadian Women s Labour-Force Participation
  Blackmoor  Exercise Histories of Eating-Disordered...
  Burt       Fraudulent Data on IQs of Twins Raised Apart
  ...
```

You could then print the top of any data set using the `head` function:

```
> head(Adler)

  instruction expectation rating
1        GOOD        HIGH     25
2        GOOD        HIGH      0
3        GOOD        HIGH    -16
4        GOOD        HIGH      5
5        GOOD        HIGH     11
6        GOOD        HIGH     -6
```

To see all of the data sets available in all the packages you have installed, even those not loaded from your library, enter the following function call:

```
data(package = .packages(all.available = TRUE))
```

3

Running R

There are several ways you can run R:

Interactively using its programming language: You can see the result of each command immediately after you submit it;

Interactively using one of several GUIs that you can add on to R: Some of these use programming while others help you avoid programming by using menus and dialog boxes like SPSS, ribbons like Microsoft Office, or flowcharts like SAS Enterprise Guide or SPSS Modeler (formerly Clementine);

Noninteractively in batch mode using its programming language: You enter your program into a file and run it all at once.

From within another package, such as Excel, SAS, or SPSS.

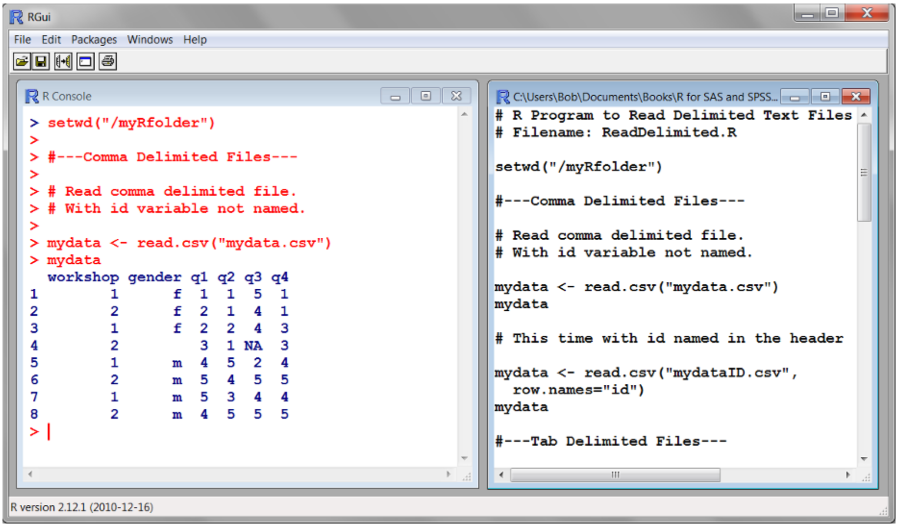
You can ease your way into R by continuing to use SAS, SPSS, or your favorite spreadsheet program to enter and manage your data and then use one of the methods below to import and analyze them. As you find errors in your data (and you know you will), you can go back to your other software, correct them, and then import them again. It is not an ideal way to work, but it does get you into R quickly.

3.1 Running R Interactively on Windows

You can run R programs interactively in several steps:

1. Start R by double-clicking on its desktop icon or by choosing *Start> All Programs> R> R x.x.x* (where x.x.x is the version of R you are using). The main R console window will appear looking like the left window in Fig. 3.1. Then enter your program choosing one of the methods described in steps 2 and 3 below.

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_3, © Springer Science+Business Media, LLC 2011


RGui
File Edit Packages Windows Help
R Console
C:\Users\Bob\Documents\Books\R for SAS and SPSS...
R version 2.12.1 (2010-12-16)

```
> setwd("/myRfolder")
>
> #---Comma Delimited Files---
>
> # Read comma delimited file.
> # With id variable not named.
>
> mydata <- read.csv("mydata.csv")
> mydata
  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2         3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
>
```

```
# R Program to Read Delimited Text Files
# Filename: ReadDelimited.R

setwd("/myRfolder")

#---Comma Delimited Files---

# Read comma delimited file.
# With id variable not named.

mydata <- read.csv("mydata.csv")
mydata

# This time with id named in the header

mydata <- read.csv("mydataID.csv",
  row.names="id")
mydata

#---Tab Delimited Files---
```

Fig. 3.1. R graphical user interface in Microsoft Windows with the console on the *left* and an open script editor window on the *right*

2. Enter R commands into the R console. You can enter commands into the console one line at a time at the ">" prompt. R will execute each line when you press the Enter key. If you enter commands into the console, you can retrieve them with the up arrow key and edit them to run again. I find it much easier to use the program editor described in the next step. If you type the beginning of an R function, such as "`me`" and press Tab, R will show you all of the R functions that begin with those letters, such as mean or median. If you enter the name of a function and an open parenthesis, such as "`mean(`," R will show you the parameters or keywords (R calls them arguments) that you can use to control that function.
3. Enter R programming commands into the R editor. Open the R editor by choosing *File> New Script*. R programs are called *scripts*. You can see one on the right side of Fig. 6.1. You can enter programs as you would in the SAS Program Editor or the SPSS Syntax Editor.
4. Submit your program from the R editor. To submit just the current line, you can hold the Ctrl key down and press "r," for *r*un, or right-click on it and choose *Run line or selection*, or using the menus choose, *Edit> Run line or selection*. To run a block of lines, select them first, and then submit them the same way. To run the whole program, select all lines by holding the Ctrl key down and pressing "a" and then submit them the same way.
5. As you submit program statements, they will appear in the R Console along with results or error messages. Make any changes you need and submit the program again until finished. You can clear the console results

by choosing *Edit*> *Clear console* or by holding the Ctrl key down and pressing “l” (i.e., Ctrl-l). See *Help*> *Console* for more keyboard shortcuts.

6. Save your script (program). Click on the R editor window to make it active and choose *File*> *Save to file*. Unlike most Windows programs, R will not automatically add the “.R” extension to files saved in the program editor. You must actually type the extension yourself. If you forget, later when you go to open the program R will not see the file, making you wonder if you actually saved it!
7. Save your output. Click on the console window to make it active and choose *File*> *Save to file*. The console output will contain the commands and their output blended together like an SPSS output file rather than the separate log and listing files of SAS. It will simply be text so giving it a file extension of “.txt” is good. Again, you will have to actually type the extension if later you want to be able to double-click on the file and open it with your default text editor.
8. Save your data and any functions you may have written. The data or function(s) you created are stored in an area called your workspace. You can save them with the command *File*> *Save Workspace*. . . . In a later R session you can retrieve it with *File*> *Load Workspace*. . . . You can also save your workspace using the `save.image` function:

   ```
   save.image(file = "myWorkspace.RData")
   ```

   Again note that you need to type the extension “.RData” at the end of the filename. Later, you can read the workspace back in with the command:

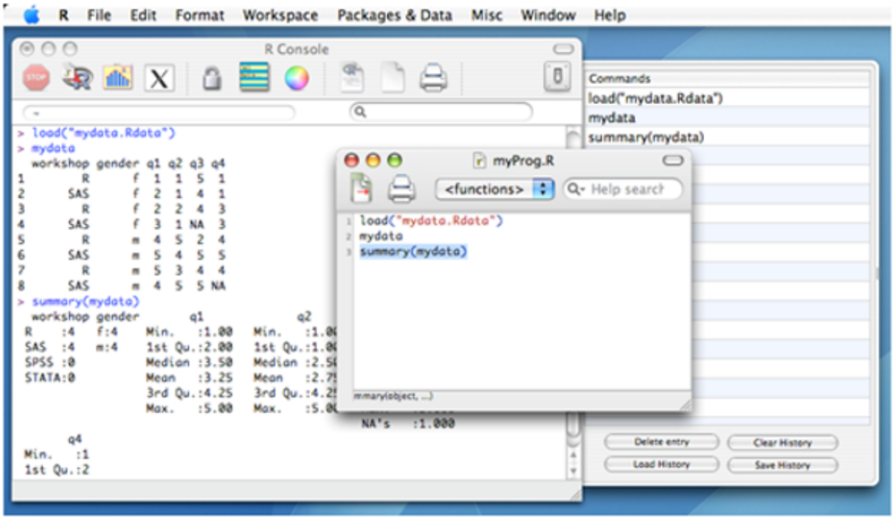
   ```
   load("myWorkspace.RData")
   ```

   For details, see Chap. 13, “Managing Your Files and Workspace.”
9. Optionally save your history. R has a history file that saves all of the commands you submit in a given session. This is just like the SPSS journal file. This is similar to the SAS log except that the history contains input and no output or system messages. If you are working with the R editor, your program is already saved in a more organized form, so I rarely save the command history.

   You can save the session history to a file using *File*> *Save History*. . . and you can load it in a future session with *File*> *Load History*. . . . You can also use R functions to do these tasks.

   ```
   savehistory(file = "myHistory.Rhistory")
   loadhistory(file = "myHistory.Rhistory")
   ```

   Note that the filename can be anything you like, but the extension should be “.Rhistory.” In fact the entire filename will be simply “.Rhistory” if you do not provide one.
10. To quit R, choose *File*> *Exit* or submit the function `quit()` or just `q()`. R offers to save your workspace automatically on exit. If you are using



```
> load("mydata.Rdata")
> mydata
  workshop gender q1 q2 q3 q4
1        R      f  1  1  5  1
2      SAS      f  2  1  4  1
3        R      f  2  2  4  3
4      SAS      f  3  1 NA  3
5        R      m  4  5  2  4
6      SAS      m  5  4  5  5
7        R      m  5  3  4  4
8      SAS      m  4  5  5 NA
> summary(mydata)
  workshop gender       q1             q2
 R    :4   f:4    Min.   :1.00   Min.   :1.0
 SAS  :4   m:4    1st Qu.:2.00   1st Qu.:1.0
 SPSS :0          Median :3.50   Median :2.5
 STATA:0          Mean   :3.25   Mean   :2.7
                  3rd Qu.:4.25   3rd Qu.:4.2
                  Max.   :5.00   Max.   :5.0
                                                NA's   :1.000

       q4
 Min.   :1
 1st Qu.:2
```

```
load("mydata.Rdata")
mydata
summary(mydata)
```

```
load("mydata.Rdata")
mydata
summary(mydata)
```

Fig. 3.2. R graphical user interface on Macintosh with the console on the *left*, script editor in the *center*, and the history window on the *right*

the `save.image` and `load` functions to tell R where to save and retrieve your workspace in step 4 above, you can answer *No*. If you answer *Yes*, it will save your work in the file ".RData" in your default working directory. The next time you start R, it will load the contents of the .RData file automatically. Creating an .RData file in this way is a convenient way to work. However, I recommend naming each project yourself, as described in step 4 above.

You can stop R from saving and restoring its own workspace by starting R with the options `--no-save --no-restore`. You can set these options by right-clicking on the R menu item or icon, choosing *Properties* and under *Target* appending the options to the string that appears there as follows:

```
"C:\Program Files...\Rgui.exe" --no-save --no-restore
```

Be careful not to change the *Properties* string itself. Then simply click *OK*. From then on, R will neither save the workspace to .RData nor load one automatically if it finds it.

3.2 Running R Interactively on Macintosh

You can run R programs interactively on a Macintosh in several steps.

1. Start R by choosing R in the Applications folder. The R console window will appear (see left window in Fig. 3.2). Then enter your program choosing one of the methods described in steps 2 and 3 below.

2. Enter R functions in the console window. You can enter commands into the console one line at a time at the ">" prompt. R will execute each line when you press the Enter key. If you enter commands into the console, you can retrieve them with the up arrow key and edit them to run again. I find it much easier to use the program editor described in the next step. If you type "me" at the command prompt and press Tab or hold the Command key down and press "." (i.e., CTRL-period), R will show you all of the R functions that begin with those letters, such as mean or median. When you type a whole function name, the arguments (parameters or keywords) that you can use to control the function will appear below it in the console window.
3. Enter R programming statements into the R editor. Open the R editor by choosing *File> New Document*. Although R programs are called *scripts*, here R uses the standard Macintosh term *document*. The R editor will start with an empty window. You can see it in the center of Fig. 3.2. You can enter R programs as you would in the SAS Program Editor or the SPSS Syntax Editor.
4. Submit your program from the R editor. To submit one or more lines, highlight them, then hold the Command key, and press Return, or choose *Edit> Execute*. To run the whole program, select it by holding down the Command key and pressing "a," and then choose *Edit> Execute*.
5. As you submit program statements, they will appear in the R Console along with results or error messages. Make any changes you need and submit the program again until finished.
6. Save your program and output. Click on a window to make it the active window and choose *File> Save as...*. The commands and their output are blended together like an SPSS output file rather than the separate log and listing files of SAS.
7. Save your data and any functions you may have written. The data or function(s) you created are stored in an area called your *workspace*. You can save your workspace with *Workspace> Save Workspace File...*. In a later R session you can retrieve it with *Workspace> Load Workspace File...*. You can also perform these functions using the R functions `save.image` and `load`:

   ```
   save.image(file = "myWorkspace.RData")
   load("myWorkspace.RData")
   ```

   For details, see Chap. 13, "Managing Your Files and Workspace."
8. Optionally save your history. R has a history file that saves all of the commands you submit in a given session (and not the output). This is just like the SPSS journal file. This is similar to the SAS log except that the history contains input and no output or system messages. If you are working with the R editor, your program is already saved in a more organized form, so I rarely save the command history.

You can view your history by clicking on the *Show/Hide R command history* icon in the console window (to the right of the lock icon). You can see the command history window on the right side of Fig. 3.2. Notice that it has alternating stripes, matching its icon. Clicking the icon once makes the history window slide out to the right of the console. Clicking it again causes it to slide back and disappear. You can see the various buttons at the bottom of the history, such as *Save History* or *Load History*. You can use them to save your history or load it from a previous session. You can also use R functions to do these tasks:

```
savehistory(file = "myHistory.Rhistory")
loadhistory(file = "myHistory.Rhistory")
```

Note that the filename can be anything you like, but the extension should be ".Rhistory." In fact the entire filename will be simply ".Rhistory" if you do not provide one.

9. Exit R by choosing *R> Quit R*. Users of any operating system can also quit by submitting the function `quit()` or just `q()`. R will offer to save your workspace automatically on exit. If you are using the `save.image` and `load` functions to tell R where to save/retrieve your workspace as recommended previously, you can answer *No*. If you answer *Yes*, it will save your work in the file ".RData" in your default working directory. The next time you start R, it will load the contents of the .RData file automatically. Some people find creating an .RData file this way a convenient way to work. However, I much prefer giving each project its own name.
   You can stop R from ever saving an .RData file by choosing the menu *R> Preferences> Startup* and under *Save workspace on exit from R* click *No*.

3.3 Running R Interactively on Linux or UNIX

You can run R programs interactively in several steps.

1. Start R by entering the command "R," which will bring up the ">" prompt, where you enter commands. For a wide range of options, refer to Appendix B, "An Introduction to R" [66], available at `http://www.r-project.org/` under *Manuals*, or in your R *Help* menu. You can enter R functions using either of the methods described in steps 2 and 3 below.
2. Enter R functions into the console one line at a time at the ">" prompt. R will execute each line when you press the Enter key. You can retrieve a function call with the up arrow key and edit it, and then press Enter to run it again. You can include whole R programs from files with the `source` function. For details, see Sect. 3.4, "Running Programs That Include Other Programs." If you type the beginning of an R function, such as "`me`" and press Tab, R will show you all of the R functions that begin with those

letters, such as mean or median. If you enter the name of a function and an open parenthesis, such as "`mean(`," R will show you the arguments (parameters or keywords) that you can use to control that function.

3. Enter your R program in an R-compatible text editor and submit your functions from there. Although R for Linux or UNIX does not come with its own GUI or program editor, a popular alternative is to use text editors that color-code their commands and automatically transfer them to R. See Sect. 3.9 for details.
4. Save your program and output. If you are working in a text editor (highly recommended), then saving your program is the usual process. You can save your output from the console window easily as well. However, if you are entering your program into the console directly, you may wish to route input and output to a file with the `sink` function. You must specify it in advance of any output you wish to save.

   ```
   sink("myTranscript.txt", split = TRUE)
   ```

   The argument `split = TRUE` tells R to display the text on the screen as well as route it to the file. The file will contain a transcript of your work. The commands and their output are blended together like an SPSS output file rather than the separate log and listing files of SAS.
5. Save your data and any functions you may have written. The data and and function(s) you created are stored in an area called your workspace. Users of any operating system can save it by calling the `save.image` function:

   ```
   save.image(file = "myWorkspace.RData")
   ```

   Later, you can read the workspace back in with the function call:

   ```
   load("myWorkspace.RData")
   ```

   For details, see Chap. 13, "Managing Your Files and Workspace."
6. Optionally save your command history. R has a history file that saves all of the functions you submit in a given session. This is just like the SPSS journal file. This is similar to the SAS log except that the history contains input and no output or system messages. If you are using a separate text editor, this step is usually unnecessary. You can save or load your history at any time with the `savehistory` and `loadhistory` functions:

   ```
   savehistory(file = "myHistory.Rhistory")
   loadhistory(file = "myHistory.Rhistory")
   ```

   Note that the filename can be anything you like, but the extension should be ".Rhistory." In fact the entire filename will be simply ".Rhistory" if you do not provide one.
7. Quit R by submitting the function `quit()` or just `q()`. R offers to save your workspace automatically on exit. If you are using the `save.image` and `load` functions to tell R where to save/retrieve your workspace as

recommended previously, you can answer *No*. If you answer *Yes*, it will save your work in the file ".RData" in your default working directory. The next time you start R, it will load the contents of the .RData file automatically. Creating an .RData file in this way is a convenient way to work. However, I prefer naming each project myself as described in step 4 above.

3.4 Running Programs That Include Other Programs

When you find yourself using the same block of code repeatedly in different programs, it makes sense to save it to a file and include it into the other programs where it is needed. SAS does this with the form

```
%INCLUDE myprog.sas ;
```

and SPSS does it with

```
INSERT FILE= myprog.sps .
```

or the similar INCLUDE command.

To include a program in R, use the `source` function:

```
source("myprog.R")
```

One catch to keep in mind is that by default R will not display any results that sourced files may have created. Of course, any objects they create – data, functions, and so forth – will be available to the program code that follows. If the program you source creates output that you want to see, you can source the program in the following manner:

```
source("myprog.R", echo = TRUE)
```

This will show you all of the output created by the program. If you prefer to see only some results, you can wrap the `print` function around only those functions whose output you do want displayed. For example, if you sourced the following R program, it would display the standard deviation, but not the mean:

```
x <- c(1, 2, 3, 4, 5)
mean(x)          # This result will not display.
print( sd(x) )   # This one will.
```

An alternative to using the `source` function is to create your own R package and load it with the `library` function. However, that is beyond the scope of this book.

3.5 Running R in Batch Mode

You can write a program to a file and run it all at once, routing its results to another file (or files). This is called batch processing. If you had a program named myprog.sas, you would run it with the following command:

```
SAS myprog
```

SAS would run the program and place the log messages in myprog.log and the listing of the output in myprog.lis. Similarly, SPSS runs batch programs with the statisticsb *b*atch command:

```
statisticsb -f myprog.sps -out myprog.txt
```

If the SPSS program uses the SPSS-R Integration Package, you must add the "-i" parameter. See the next section for details. In its GUI versions, SPSS also offers batch control through its Production Facility.

In R, you can find the details of running batch on your operating system by starting R and entering the following command. Note that the letters of BATCH must be all uppercase:

```
help("BATCH")
```

In Microsoft Windows batch processing is simplified with a set of batch files that are available on CRAN at http://cran.\linebreakr-project.org/other-software.html. Here is an example of using the *Rscript.bat* file to run an R program and display the results on your screen:

```
Rscript myprog.R
```

If you prefer to route your results to a file, you can do so using

```
Rscript myprog.R > myprog.Rout
```

It will route your results to myprog.Rout.

UNIX users can run a batch program with the following command. It will write your output to myprog.Rout:

```
R CMD BATCH myprog.R
```

There are, of course, many options to give you more control over how your batch programs run. See the help file for details.

3.6 Running R in SAS and WPS

Neither SAS nor the similar World Programming System (WPS) keeps its data in your computer's main memory as R does. So you can use either of them to read vast amounts data, manage or transform the data, select the variables and observations you need, and then pass them on to R for analysis. This approach also lets you make the most of your SAS/WPS know-how, calling on R only after the data are cleaned up and ready to analyze.

For example, we can read our practice data set, keep only the variables named q1 through q4, eliminate the missing values using the N function, and select only the males using

```
LIBNAME myLib  C:\myRfolder ;
DATA mySubset;
  SET  myLib.mydata;
  * Keep only the variables you need;
  KEEP q1-q4;
  *Eliminate missing values;
  WHERE N(OF q1-q4) = 4 & gender = "m";
RUN;
```

Now we are ready to send these data on to R. SAS users can run R programs in four ways:

Through SAS/IML Studio;
Through a program called *A Bridge to R*;
Through the SAS X command;
Sequentially, simply using SAS followed by R.

We will discuss these variations in the following sections.

3.6.1 SAS/IML Studio

The most comprehensive approach to running R in SAS is SAS/IML Studio. The aptly named subroutine `ExportDatasetToR` sends your data set to R, and the `submit/R;` statement tells IML Studio that R code follows.

```
proc iml;
run ExportDatasetToR("mySubset");
submit/R;
```

Now we are ready to run any R code we like. For example, to print our data and perform a linear regression analysis we can use:

```
print(mydata)
myModel <- lm(q4 ~ q1 + q2 + q3, data = mydata)
summary(myModel)
```

We will discuss those R statements later. When you are ready to finish your R program and return to SAS programming statements, enter the statement:

```
endsubmit;
```

For details regarding transferring data or results back and forth between SAS and R, see Wicklin's article [75].

3.6.2 A Bridge to R

A similar way to run R programs from within SAS is to use a software package called *A Bridge to R*, available from MineQuest, LLC (http://www.minequest.com). That program adds the ability to run R programs from either Base SAS or WPS. It sends your data from SAS or WPS to R using a SAS transport format data set, which only allows for eight-character variable names. To use it, simply place your R programming statements where our indented example is below and submit your program as usual.

```
%Rstart(dataformat = XPT, data = mydata,
   rGraphicsViewer = NOGRAPHWINDOW);
datalines4;
  print(mydata)
  myModel <- lm(q4 ~ q1 + q2 + q3, data = mydata)
  summary(myModel)
;;;;
%Rstop(import=);
```

While this approach uses SAS transport format files behind the scenes, you do not have to create them yourself nor do you need to import them into R.

3.6.3 The SAS X Command

The third way to run R from within SAS is to use SAS's X command. This is less expensive than the previous two approaches, which require you to purchase additional software. However, it also has a big disadvantage: you must pass your data back and forth between SAS and R by writing data or results to files, and you must run R in batch mode. Here are the steps to follow:

1. Read your data into SAS as usual and do whatever data preparation work you need. Then write your data to a permanent SAS data set.
2. Use SAS's X command to submit a batch program that runs your R program. For example:

   ```
   OPTIONS NOXWAIT;
   X  CD C:\myRfolder  ;
   X  Rscript ReadSAS.R  > ReadSAS.Rout ;
   ```

The NOXWAIT option tells SAS to close the Windows command window automatically. The CD command should change to the directory in which you have your R program. The Rscript.bat file must either be in that directory or must be on your system's path. For details regarding running R in batch mode, see Sec. 3.5.

3. R has now placed its results in a file, in this example, ReadSAS.Rout. You can use any method you like to see its contents. The Windows type command is perhaps the easiest:

   ```
   OPTIONS XWAIT;
   X  type ReadSAS.Rout ;
   ```

   This time I set the XWAIT option so I could read the results before the command window disappeared. The only things that the ReadSAS.R program did was read a SAS data set, convert it to R, and print it, as described in Sec. 6.10, "Reading Data from SAS." When I was done viewing the results, I entered the Windows exit command to continue.
4. If you need to return any results to SAS, you must write them to a file as described in Sec. 6.16. Then read them back into your SAS program and continue working.

3.6.4 Running SAS and R Sequentially

The fourth way to use SAS and R together is to use them sequentially. That is, do your initial work in SAS, write your data to a SAS data set and exit SAS. Then start R, import the data and continue working. For details on reading SAS data sets in R, see Sec. 6.10. This approach is easy to implement, does not require additional software, and allows you to explore your data interactively in R.

3.6.5 Example Program Running R from Within SAS

The program below demonstrates the first three approaches discussed above to read data, pass it on to R, and run analyses there. The last approach is described in Sec. 6.10.

```
* Filename: RunningRinSAS.sas ;

LIBNAME myLib  C:\myRfolder ;
DATA mySubset;
  SET  myLib.mydata;
  * Keep only the variables you need;
  KEEP q1-q4;
  *Eliminate missing values;
  WHERE N(OF q1-q4)=4 & gender="m";
RUN;
```

```
* Using SAS/IML;
proc iml;
run ExportDatasetToR("mySubset");
submit/R;
  print(mydata)
  myModel <- lm(q4 ~ q1 + q2 + q3, data = mydata)
  summary(myModel)
endsubmit;

* Using A Bridge to R;
%Rstart(dataformat = XPT, data = mydata,
   rGraphicsViewer = NOGRAPHWINDOW);
datalines4;
  print(mydata)
  myModel <- lm(q4 ~ q1 + q2 + q3, data = mydata)
  summary(myModel)
;;;;
%Rstop(import=);

* Running R with X Command;
OPTIONS NOXWAIT;
X  CD C:\myRfolder  ;
X  Rscript ReadSAS.R  > ReadSAS.Rout ;

* Displaying the results;
OPTIONS XWAIT;
X  type ReadSAS.Rout ;
* Enter "exit" in the command window ;
* when you finish reading it         ;
```

3.7 Running R in SPSS

SPSS has a very useful interface to R that allows you to transfer data back and forth, run R programs, and get R results back into nicely formatted SPSS pivot tables. You can even add R programs to SPSS menus so that people can use R without knowing how to program.

Since SPSS does not need to keep its data in the computer's main memory as R does, you can read vast amounts of data into SPSS, select the subset of variables and/or cases you need and then pass them on to R for analysis. This approach also lets you make the most of your SPSS know-how, calling on R only after the data are cleaned up and ready to analyze.

This interface is called the *SPSS Statistics-R Integration Package* and it is documented fully in a manual of the same name [52]. The package plug-in and its manual are available at http://www.ibm.com/developerworks/spssdevcentral. Full installation instructions are also at that site, but it is quite easy as long as you follow the steps in order. First install SPSS (version 16 or later), then the latest version of R that it supports, and finally the plug-in. The version of R that SPSS supports at the time you download it may be a version or two behind R's production release. Older versions of R are available at http://cran.r-project.org/.

Understanding how the *SPSS Statistics-R Integration Package* works requires discussing topics that we have not yet covered. If this is your first time reading this book, you might want to skip this section for now and return to it when you have finished the book.

To see how to run an R program within an SPSS program, let us step through an example. First, you must do something to get a data set into SPSS. We will use our practice data set mydata.sav, but any valid SPSS data set will do. Open the data by choosing *File> Open> Data* from the menus or by running the SPSS programming code below. If you use the commands, adjust your path specification to match your computer.

```
CD  C:\myRfolder .
GET FILE =  mydata.sav .
```

Now that you have data in SPSS, you can do any type of modifications you like, perhaps creating new variables or selecting subsets of observations before passing the data to R. For the next step, you must have an SPSS syntax window open. So if you used menus to open the file, you must now choose *File> New> Syntax* to open a program editor window. Enter the program statement below.

```
BEGIN PROGRAM R.
```

From this command on we will enter R programming statements. To get the whole current data set and name it mydata in R we can use the following:

```
mydata <- spssdata.GetDataFromSPSS(missingValueToNA = TRUE)
```

The argument `missingValueToNA = TRUE` converts SPSS's missing values to R's standard representation for missing, which is "NA". Without that argument, SPSS will convert them to them to "NaN", Not a Number, a different kind of missing value in R. Many R procedures treat these two in the same way, but it is best to use NA unless you have a specific reason not to. When I created the data set in SPSS, I set a blank to represent missing data for gender so it would transfer to R as a missing value. I also set the scale of the workshop variable to be nominal, so it would pass to R as R's equivalent, a factor. Getting the data from SPSS to R is a snap, but getting an R data set to SPSS is more complicated. See the manual for details.

The previous example took all of the variables over to R. However, it is often helpful to select variables by adding two arguments to this R function.

The variables argument lets you list variables similar to the way SPSS does except that it encloses the list within the c function. We will discuss that function more later. You can use the form c("workshop gender q1 to q4") or simply c("workshop to q4"). You can also use syntax that is common to R, such as c(1:6). This syntax uses the fact that workshop is the first variable and q4 is sixth variable in the data set.

```
mydata <- spssdata.GetDataFromSPSS(
  variables = c("workshop gender q1 to q4"),
  missingValueToNA = TRUE,
  row.label = "id" )
```

You can include the optional row.label argument to specify an ID variable that R will use automatically in procedures that may identify individual cases. If the data set had SPLIT FILE turned on, this step would have retrieved only data from the first split group. See the manual for details about the aptly named function, GetSplitDataFromSPSS.

Now that we have transferred the data to R, we can write any R statements we like. Below we print all of the data.

```
> mydata

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2   <NA>  3  1 NA  3
5        1      m  3  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

Notice that the variable ID is not labeled. Its values are used on the far left to label the rows. If we had not specified the row.label = "id" argument, we would see the ID variable listed before workshop and labeled “id.” However, the row labels would still appear the same because R always labels them. If you do not provide a variable that contains labels to use, it defaults to simply sequential numbers, 1, 2, 3, etc.

Now let us calculate some descriptive statistics using variables q1 to q4. There are a number of different ways to select variables in R. One way is to use mydata[3:6] since q1 is the third variable in the data set and q4 is the sixth. R can select variables by name, but we will save that topic for later. The summary function in R gets descriptive statistics.

```
summary(mydata[3:6])

       q1               q2              q3               q4
 Min.   :1.000   Min.   :1.00   Min.   :2.000   Min.   :1.00
 1st Qu.:2.000   1st Qu.:1.00   1st Qu.:4.000   1st Qu.:2.50
 Median :3.000   Median :2.50   Median :4.000   Median :3.50
 Mean   :3.125   Mean   :2.75   Mean   :4.143   Mean   :3.25
 3rd Qu.:4.250   3rd Qu.:4.25   3rd Qu.:5.000   3rd Qu.:4.25
 Max.   :5.000   Max.   :5.00   Max.   :5.000   Max.   :5.00
                                NA s   :1.000
```

Next, we will do a linear regression model using standard R commands that we will discuss in detail much later. Our goal here is just to see what the `spsspivottable.Display` function does.

```
myModel <- lm(q4 ~ q1 + q2 + q3, data = mydata)
myAnova <- anova(myModel)

spsspivottable.Display(myAnova,
  title  = "My ANOVA table",
  format = formatSpec.GeneralStat)
```

The function call immediately above created Table 3.1 formatted exactly as you see it. I routinely tell SPSS to put all my output in CompactAcademicTimesRoman style. That style draws only horizontal lines in tables, as most scientific journals prefer. If you copy this table and paste it into a word processor, it should maintain its nice formatting and be a fully editable table.

When I ran the program, this table appeared first in the SPSS output window even though it was the last analysis run. SPSS puts its pivot tables first.

So far we have submitted commands to R and seen the results returned to SPSS. You can, however, open an R console window using this command:

```
browser()
```

From there you can interact with R and see the result in R's console window.

Finally, I ended the program with the statement below and exited SPSS in the usual way:

```
END PROGRAM.
```

If your program contains some R code, then some SPSS code, then more R code, any data sets or variables you created in the earlier R session(s) will still exist. If the program you submit from SPSS to R uses R's `quit` function, it will cause both R and SPSS to terminate. To learn how to add R functions to the SPSS GUI, see the SPSS help file topic, *Custom Dialog Builder* in SPSS Statistics 17 or later.

Table 3.1. An example pivot table created by R and transferred to SPSS.

	My ANOVA table				
	Df	Sum Sq	Mean Sq	F value	Pr(>F)
q1	1.000	12.659	12.659	34.890	.010
q2	1.000	3.468	3.468	9.557	.054
q3	1.000	.213	.213	.587	.499
Residuals	3.000	1.089	.363		

3.7.1 Example Program Running R from Within SPSS

The program below combines all of the steps discussed above to read data, pass it on to R, and run analyses there.

```
* Filename: RunningRinSPSS.sps

CD  C:\myRfolder .
GET FILE =  mydata.sav .

BEGIN PROGRAM R.

mydata <- spssdata.GetDataFromSPSS(
  variables = c("workshop gender q1 to q4"),
  missingValueToNA = TRUE,
  row.label = "id" )

mydata

mydata[3:6]

myModel <- lm(q4 ~ q1 + q2 + q3, data = mydata)
myAnova <- anova(myModel)

spsspivottable.Display(myAnova,
title  = "My Anova Table",
format = formatSpec.GeneralStat)

END PROGRAM.
```

3.8 Running R in Excel

R integrates nicely into Excel, where you can control it using either function calls or the R Commander menus described in Sect. 3.11.2. As of this writing, only Excel on Microsoft Windows can control R, and both must be the 32-bit versions. However, work is underway to support the 64-bit versions and to allow similar control from the spreadsheet in the free OpenOffice.org software.

There are several pieces of software – some R packages and some not – that allow communications to flow between R and Excel. Among these pieces are Neuwirth's `RExcel` [43] and Heiberger and Neuwirth's `R through Excel` [27] and Baier and Neuwirth's `statconnDCOM` software [5]. The latter implements Microsoft's Distributed Component Object Model (DCOM), which allows R to communicate with any software that implements that standard.

The easiest way to get R working with Excel is to use the *R and Friends* installer available at `http://rcom.univie.ac.at/`. That installer gives you R, R Commander, and all the pieces you need to make them work with Excel.

If you already have R installed, you can also install the pieces one at a time. Start by installing and loading the `RExcelInstaller` package:

```
install.packages("RExcelInstaller")
library("RExcelInstaller")
```

The package will then give you detailed instructions on the steps to follow to finish the installation.

After installing the software and starting Excel, you will see a new RExcel menu in the Add-Ins tab (Fig. 3.3, upper left corner). Choosing *RExcel> R Commander> with Excel Menus* will activate menus that we will use in a different way in Sect. 3.11.2. You can see the menus in Fig. 3.3 where I have selected *Statistics> Means* from the R Commander menus.

You can use Excel to open any file and then transfer it to R by selecting its cell range (including a row with variable names) and choosing *Add-Ins> RExcel> Put R Var> Dataframe*. With your data frame transferred to R, you can then analyze it using R commands or R Commander menus. R Commander works on an "active data set", which you can select by choosing *Data> Active data set> Select active data set*. You can save the R data frame by choosing *Data> Active data set> save active data set*.

To transfer a data frame from R to Excel, set the one you want to be the active data set (if you have not already done so) by choosing *Data> Active data set> Select active data set. . . .* Then position the cursor in a spreadsheet cell and choose *Add-Ins> RExcel> Get R Value> Active dataframe*. The data frame will appear below and to the right of the chosen cell.

After you run an analysis, you can bring its results into an Excel spreadsheet by selecting a cell and then choosing *Rexcel> Get R output*.

For a much more comprehensive demonstration of the various ways to use R and Excel together, see Richard M. Heiberger and Erich Neuwirth's book *R Through Excel: A Spreadsheet Interface for Statistics, Data Analysis, and Graphics* [27].

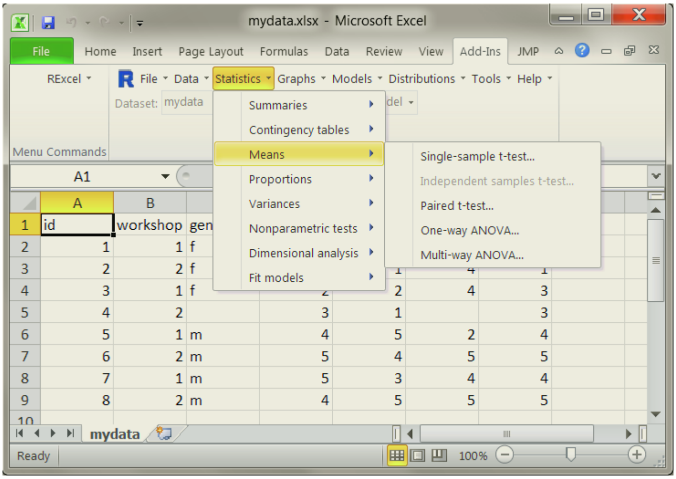


Fig. 3.3. R and R Commander both integrated into Excel

3.9 Running R from Within Text Editors

Although R offers a basic program editor on Windows and Macintosh, people who program a lot prefer to use a powerful editor. While any text editor will do, there are significant advantages in choosing one that is optimized for R. Since the R versions for Linux and UNIX do not include a text editor, this approach is a popular way to run R on those systems. The advantages of running R from a text editor include the following:

Such editors are connected to R itself and can submit code directly and interactively without leaving the editor. This greatly speeds your work.

Such editors understand R syntax and apply color coding to help you avoid trouble. For example, you have probably been vexed with forgetting to put a final quotation mark around a character string. Your following commands become part of that string and you have a mess on your hands! These editors will keep all following code highlighted as they do character strings, making problems like that obvious.

Such editors can deactivate a block of R code by adding the R comment symbol # to the beginning of each line. That is very helpful in debugging.

Such editors can automatically open all the programs you were working on when you exited the software the last time. If you work on several programs at once, copying and pasting sections among them (as I often do), this is *much* faster than remembering what you were doing before and finding each file to open.

The following are some good text editors:

EMACS with ESS – Available on all operating systems, EMACS is the hands-down favorite among hard-core UNIX fans. The Emacs Speaks Statistics (ESS) option is what links it to R and provides its R-specific capabilities. This editor can do it all, but it is not the easiest to learn. You can download it at http://ess.r-project.org/. Emacs with ESS pre-packaged is also available for Windows at http://vgoulet.act.ulaval.ca/en/emacs/.

Komodo Edit with SciViews-K – Komodo Edit from ActiveState is a full-featured program editor that is free. It makes heavy use of menus and dialog boxes, making it particularly easy for Windows and Mac users to learn. Philippe Grosjean's SciViews-K extension provides the link to R and it even includes some rudimentary dialog boxes that generate R code for you. It runs on Windows, Macintosh, Linux, and UNIX and is available at http://sciviews.org/SciViews-K/.

Notepad++ with NppToR – Windows users usually know the rudimentary Notepad editor that comes with the operating system. Notepad++ is a full-featured editor that is modeled after Notepad itself, making it very easy to learn. Andrew Redd wrote the NppToR part that hooks Notepad++ into R. It runs the standard R GUI at the same time, letting you use features of both. This Windows-specific software is available at http://npptor.sourceforge.net.

Tinn-R – This is another Windows editor that works similarly to Notepad [58]. Tinn stands for "Tinn Is Not Notepad." It has frequently used R commands on its toolbar and also includes R reference material to help you program. Tinn-R is available at: http://www.sciviews.org/Tinn-R/

3.10 Integrated Development Environments

An integrated development environment (IDE) is a set of tools that work together to enhance programmer productivity. Since IDEs include text editors that are optimized for the language you are using, they have all the advantages discussed in the previous section. For R, IDEs may also include package managers to help you install, load, and manage add-on packages; object managers to let you view and manage things like data sets; and help or documentation viewers and even graphics display windows.

3.10.1 Eclipse

The Eclipse IDE has a full suite of debugging tools and supports most programming languages. This power comes at a price, however. Its complexity means that it is used mainly by full-time programmers. If you have been using SAS AppDev Studio with the Eclipse plug-in, this is the R equivalent. Stephan Wahlbrink's StatET plug-in provides the link from Eclipse to R. It runs on Windows, Macintosh, Linux, and UNIX and is available at `http://www.walware.de/goto/statet`.

3.10.2 JGR

JGR [28] (pronounced "jaguar") stands for the *J*ava *G*UI for *R*. It is very similar to R's own simple interface, making it very easy to learn. Written by Helbig, et al., JGR provides some very helpful additions to R, like syntax checking in its program editor. It also provides the help R files in a way that lets you execute any part of an example you select. That is very helpful when trying to understand a complicated example.

JGR is installed differently than most R packages. In Microsoft Window or Apple Macintosh, you download two programs: an installer and a launcher. Running the installer installs JGR, and double-clicking the launcher starts it up. The JGR Web site that contains both programs is `http://www.rforge.net/JGR/`. Linux users follow slightly different steps that are described at the site.

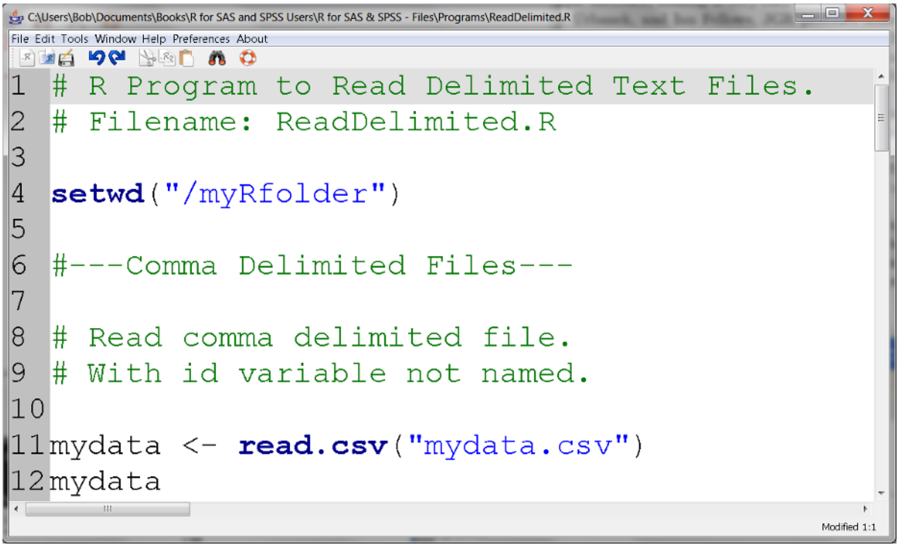
I started JGR by double-clicking on its launcher and opened an R program using *File> Open Document*. You can see the program in Fig. 3.4. Note that the JGR program editor has automatically color-coded my comments, function names, and arguments, making it much easier to spot errors. In the printed version of this book, those colors are displayed as shades of gray. If someone brings you a messy R program, the program editor can format it nicely by choosing *Edit> Format Selection*.

In the next example, I typed "`cor(`" into the bottom of the console area shown in Fig. 3.5. JGR then displayed a box showing the various arguments that control the `cor` function for doing correlations. That is very helpful when you are learning!

JGR's *Package Manager* makes it easier to control which packages you are using (Fig. 3.6). Simply checking the boxes under "loaded" will load those packages from your library. If you also check it under "default," JGR will load them every time you start JGR. Without JGR's help, automatically loading packages would require editing your .Rprofile as described in Appendix C.

JGR's *Object Browser* makes it easy to manage your workspace; see Fig. 3.7. Selecting different tabs across the top enable you to see the different types of objects in your workspace. Double-clicking on a data frame in Object Browser starts the *Data Table* editor, which is much nicer than the one built into R. It lets you rename variables, search for values, sort by clicking on variable names, cut and paste values, and add or delete rows or columns.

If you have created models, they will appear under the *models* tab. There you can do things like review them or sort them by various measures such as their R-squared values. There are many more useful features in JGR that are described on its Web site.


C:\Users\Bob\Documents\Books\R for SAS and SPSS Users\R for SAS & SPSS - Files\Programs\ReadDelimited.R
File Edit Tools Window Help Preferences About
Modified 1:1

```
# R Program to Read Delimited Text Files.
# Filename: ReadDelimited.R

setwd("/myRfolder")

#---Comma Delimited Files---

# Read comma delimited file.
# With id variable not named.

mydata <- read.csv("mydata.csv")
mydata
```

Fig. 3.4. Color-coded editor in JGR helps prevent typing errors

3.10.3 RStudio

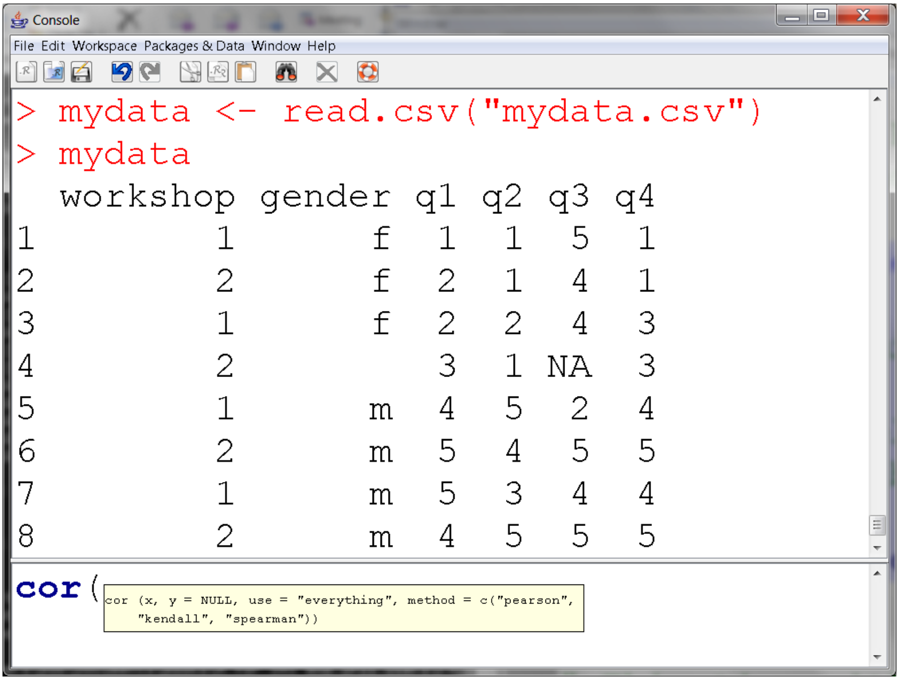
RStudio is a free and open source integrated development environment written by JJ Allaire, Joe Cheng, Josh Paulson, and Paul DiCristina, at RStudio, Inc. It is easy to install, learn, and use. It runs on Windows, Macintosh, Linux, and even runs over the Web in a browser window from Linux servers.

The screenshot in Fig. 3.8 shows the program I was editing in the upper left, the data set I created in the upper right, the program output in the lower left, and a plot of a linear regression in the lower right. Each of the four windows is tabbed, allowing you to do many more things including browsing your files, editing multiple programs, reading documentation, examining your command history, and installing or loading packages.

RStudio is a big improvement over the default user interfaces on any of the operating systems that it supports. You can download it for free at `http://rstudio.org/`.

3.11 Graphical User Interfaces

The main R installation provides an interface to help you enter programs. For Windows and Macintosh users it includes a very minimal GUI. As we have discussed, that interface allows you to use menus and dialog boxes for a few tasks, like opening and saving files. However, it does not include a point-and-click GUI for running analyses. Fortunately, users have written several GUIs to address this need. You can learn about several at the main R Web site,



```
> mydata <- read.csv("mydata.csv")
> mydata
  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2         3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

```
cor (
cor (x, y = NULL, use = "everything", method = c("pearson",
    "kendall", "spearman"))
```

Fig. 3.5. JGR showing arguments that you might choose for the `cor` function

http://www.r-project.org/, under *Related Projects* and then *R GUIs*. We will discuss the most promising ones in this section.

3.11.1 Deducer

Ian Fellows' `Deducer` [18] is a user interface that is similar to SPSS's point-and-click menu system (Fig. 3.9). It is also similar to R Commander covered in Sect. 3.11.2. Having arrived on the scene more recently, `Deducer` does not have as many plug-ins as R Commander, nor it does not yet integrate into Excel.

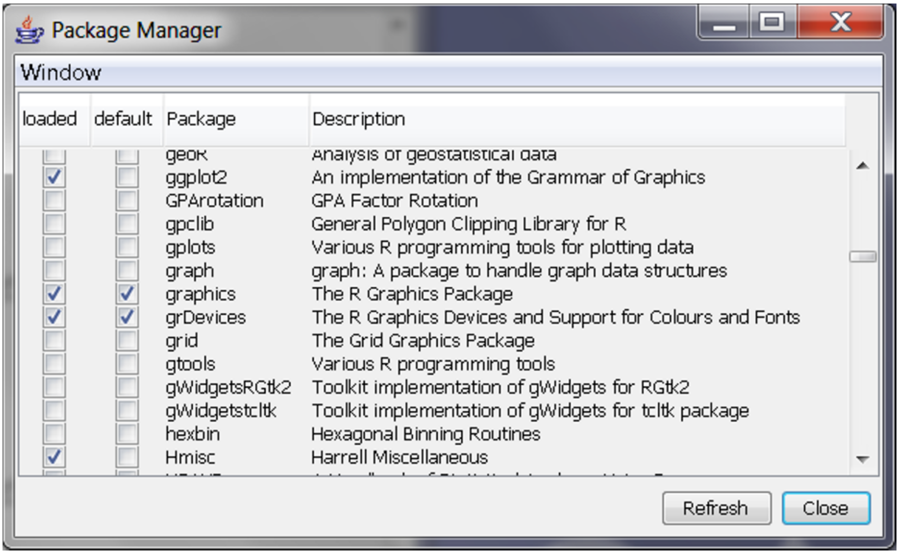
You can install `Deducer` using:

```
install.packages("Deducer")
```

When you load it from the library with:

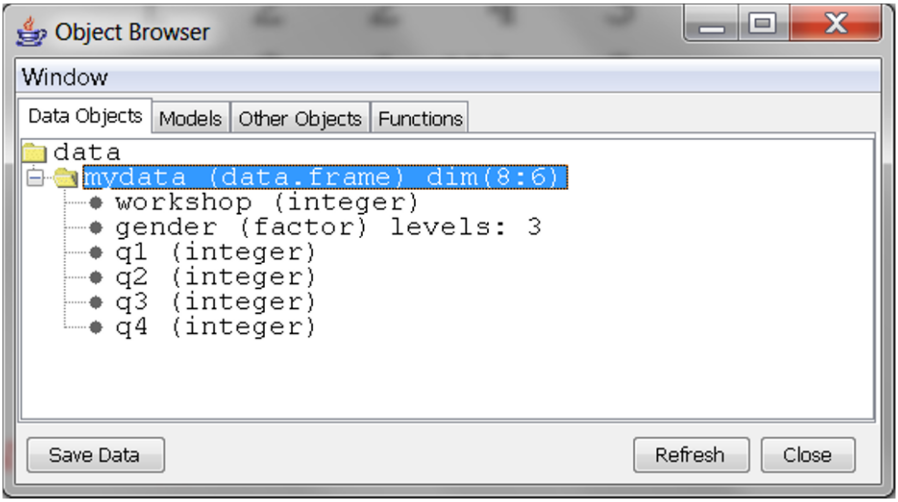
```
library("Deducer")
```

Your R Console window will gain some menu choices, which you can see in Figure 3.9. Here are the steps I followed to perform a simple analysis using `Deducer`.



loaded	default	Package	Description
☐	☐	geoR	Analysis of geostatistical data
☑	☐	ggplot2	An implementation of the Grammar of Graphics
☐	☐	GPArotation	GPA Factor Rotation
☐	☐	gpclib	General Polygon Clipping Library for R
☐	☐	gplots	Various R programming tools for plotting data
☐	☐	graph	graph: A package to handle graph data structures
☑	☑	graphics	The R Graphics Package
☑	☑	grDevices	The R Graphics Devices and Support for Colours and Fonts
☐	☐	grid	The Grid Graphics Package
☐	☐	gtools	Various R programming tools
☐	☐	gWidgetsRGtk2	Toolkit implementation of gWidgets for RGtk2
☐	☐	gWidgetstcltk	Toolkit implementation of gWidgets for tcltk package
☐	☐	hexbin	Hexagonal Binning Routines
☑	☐	Hmisc	Harrell Miscellaneous

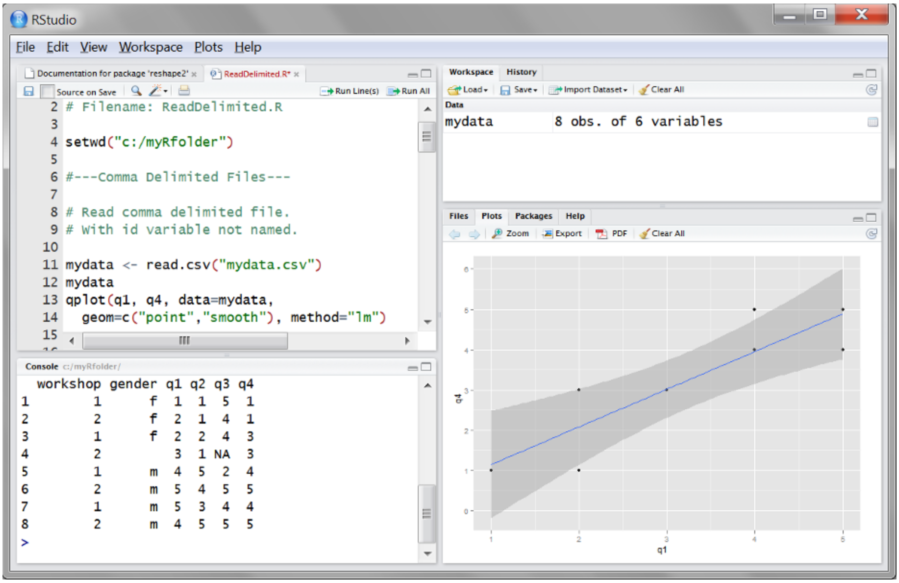
Fig. 3.6. JGR's Package Manager, which allows you to load packages from the library on demand or at startup



```
data
mydata (data.frame) dim(8:6)
  workshop (integer)
  gender (factor) levels: 3
  q1 (integer)
  q2 (integer)
  q3 (integer)
  q4 (integer)
```

Fig. 3.7. JGR's Object Browser shows information about each object in your workspace

1. I opened mydata by choosing *Deducer> Open data*, browsed to myRfolder, and chose mydata.RData to open.
2. To see the data, I chose *Deducer> Data viewer*. The data popped up as shown in Fig. 3.10. Using it, you can see and edit the data. Note that it has a *Data View* tab and a *Variable View* tab, just like the SPSS data editor.

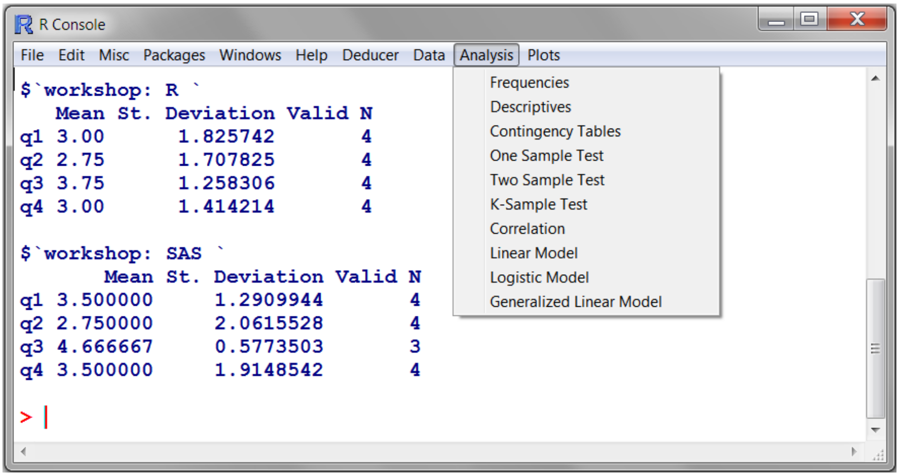


```
# Filename: ReadDelimited.R

setwd("c:/myRfolder")

#---Comma Delimited Files---

# Read comma delimited file.
# With id variable not named.

mydata <- read.csv("mydata.csv")
mydata
qplot(q1, q4, data=mydata,
  geom=c("point","smooth"), method="lm")
```

```
mydata          8 obs. of 6 variables
```

```
  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2         3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
>
```

Fig. 3.8. RStudio lets you edit files (*upper left*) manage objects (*upper right*), view output (*lower left*) and view plots, files or packages (*lower right*)

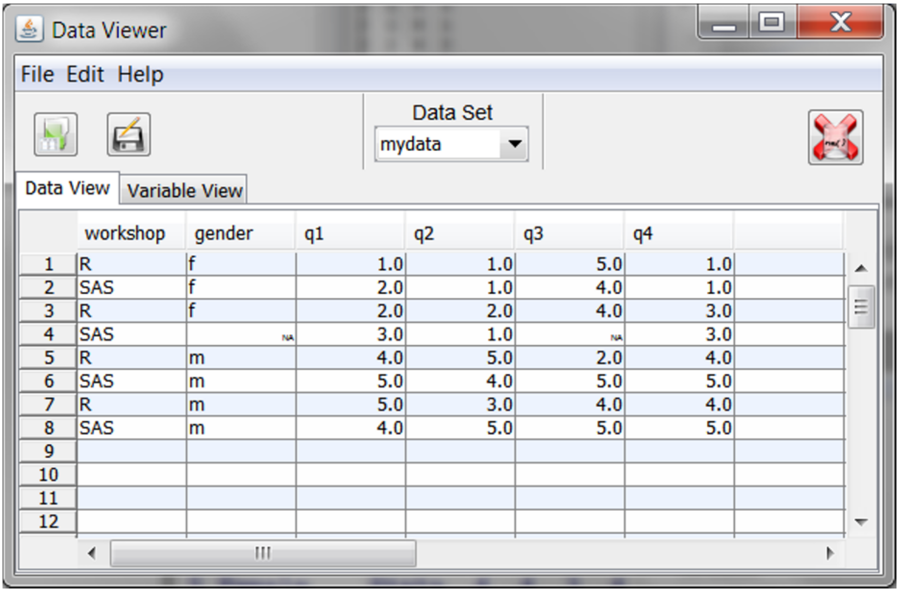


```
$`workshop: R `
   Mean St. Deviation Valid N
q1 3.00      1.825742       4
q2 2.75      1.707825       4
q3 3.75      1.258306       4
q4 3.00      1.414214       4

$`workshop: SAS `
       Mean St. Deviation Valid N
q1 3.500000     1.2909944       4
q2 2.750000     2.0615528       4
q3 4.666667     0.5773503       3
q4 3.500000     1.9148542       4

> |
```

Fig. 3.9. The `Deducer` graphical user interface integrated into the main R console

These allow you to see both the data (shown) and variable information such as variable names and types.

3. To obtain some descriptive statistics, I chose *Analysis> Descriptives*. I selected the q variables and clicked the right-facing arrow icon to move



	workshop	gender	q1	q2	q3	q4
1	R	f	1.0	1.0	5.0	1.0
2	SAS	f	2.0	1.0	4.0	1.0
3	R	f	2.0	2.0	4.0	3.0
4	SAS	NA	3.0	1.0	NA	3.0
5	R	m	4.0	5.0	2.0	4.0
6	SAS	m	5.0	4.0	5.0	5.0
7	R	m	5.0	3.0	4.0	4.0
8	SAS	m	4.0	5.0	5.0	5.0
9						
10						
11						
12						

Fig. 3.10. `Deducer`'s Data viewer/editor

them to the *Descriptives of:* box. You can see this step in Fig. 3.11. I then chose workshop as my *Stratify By:* variable and chose *Continue.*

4. I was offered a list of statistics from which to choose (not shown). I accepted the defaults and chose *Run.* The results are shown back in Fig. 3.9.

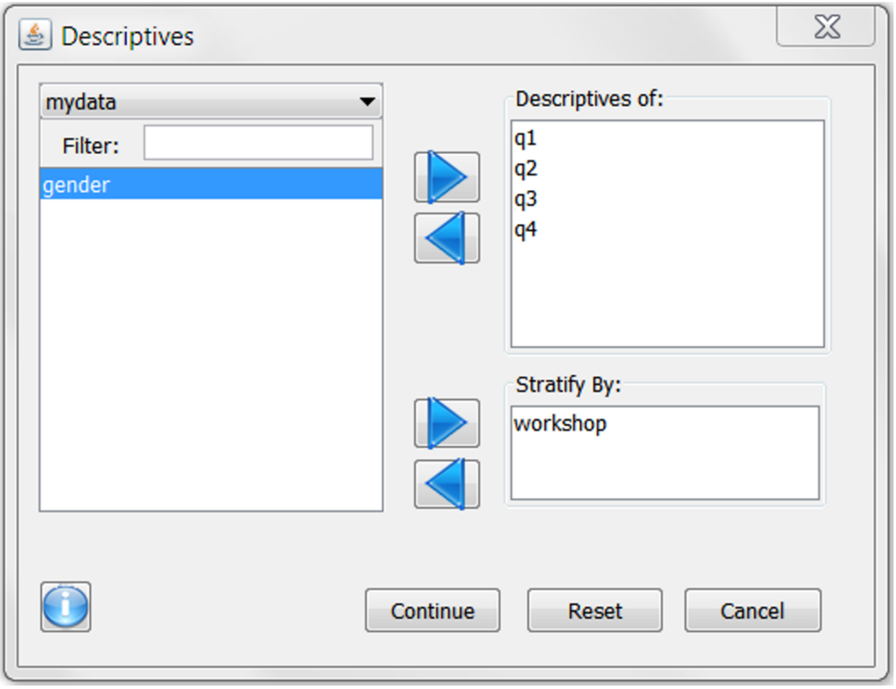
`Deducer` also has a powerful *Plot Builder* (Fig. 3.12) that helps you create graphs using the flexible Grammar of Graphics approach that we will discuss in Chap. 16. Plot Builder is very similar to IBM's SPSS Visualization Designer. This feature alone makes it worth looking into, even if you do most of your work using programming or other user interfaces.

`Deducer` also integrates into the JGR user interface, which is covered in Sect. 3.10.2. The combination of `Deducer` and JGR provides a very comprehensive set of tools for both R beginners and advanced programmers.

3.11.2 R Commander

Fox's R Commander [20] looks and works similarly to the SPSS GUI. It provides menus for many analytic and graphical methods and shows you the R commands that it enters, making it easy to learn the commands as you use it. Since it does not come with the main R installation, you have to install it one time with the `install.packages` function:

```
install.packages("Rcmdr", dependencies = TRUE)
```



mydata

Filter:

gender

Descriptives of:
q1
q2
q3
q4

Stratify By:
workshop

Fig. 3.11. `Deducer`'s descriptive statistics dialog box

R Commander uses *many* other packages, and R will download and install them for you if you use the `dependencies = TRUE` argument. It also has many plug-ins available that add even more methods to its menus. They are easy to find because their names all begin with "RcmdrPlugin." You install them just like any other R package. To use them, you start R Commander and choose *Tools> Load Rcmdr plug-in(s)....* A menu will then appear from which you can choose the plug-in you need.

Let us examine a basic R Commander session. Below are the steps I followed to create the screen image you see in Fig. 3.13.

1. I started R. For details see the section, "Running R Interactively on Windows," or similarly named sections for other operating systems previously covered in this chapter.
2. Then, from within R itself I started R Commander by loading its package from the library using `library("Rcmdr")`. That brought up the window similar to the one shown in Fig. 3.13, but relatively empty.
3. I then chose *Data>Load a data set*, browsed to C:\myRfolder, and selected mydata.RData.
4. Unlike the SPSS GUI, the data did not appear. So I clicked on the *View data set* button. The data appeared, I looked it over, then closed it.

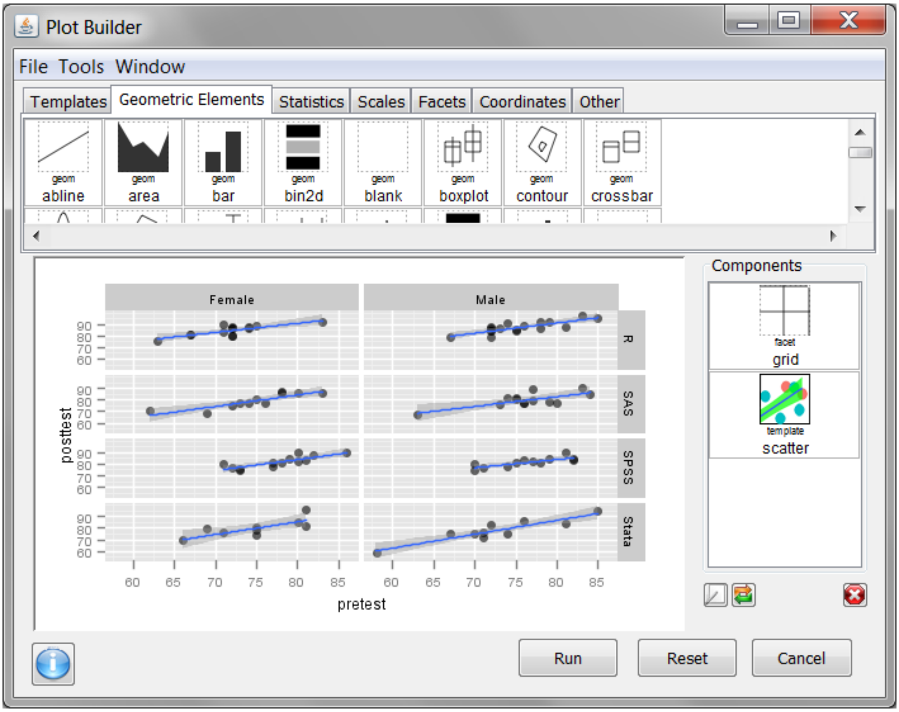


Fig. 3.12. The `Deducer`'s powerful and easy-to-use Plot Builder

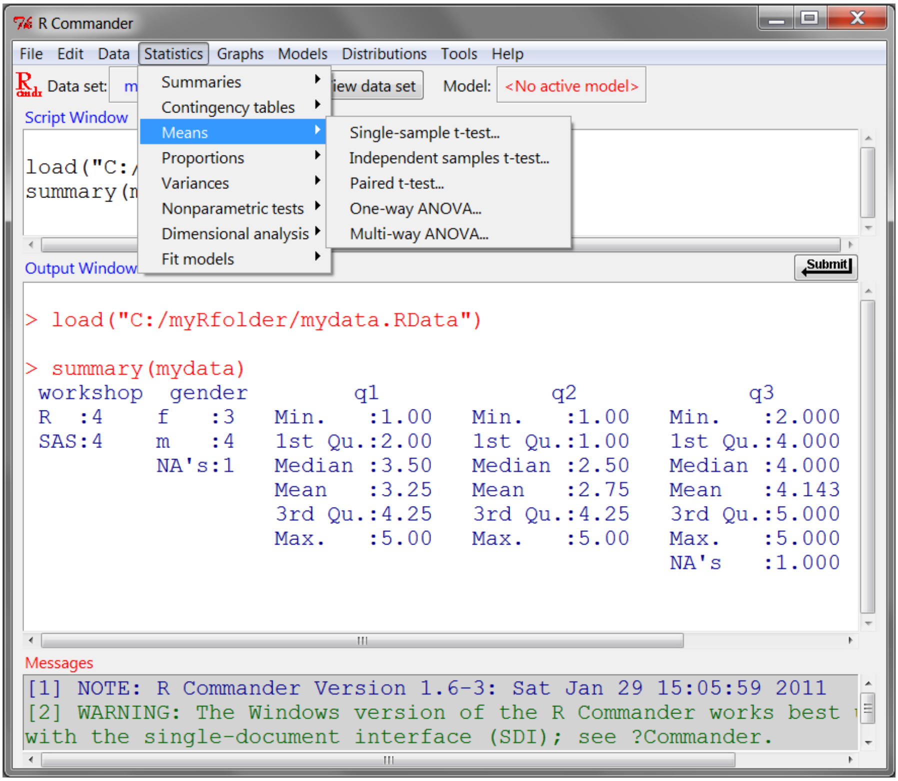
5. I then chose *Statistics> Summaries> Active Data Set*. You can see the output on the bottom of the screen in Fig. 3.13.
6. Finally, I chose *Statistics> Means*. The menu is still open, showing that I can choose various t-tests and analysis of variance (ANOVA) procedures.

You can learn more about R Commander at `http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/`.

3.11.3 `rattle`

Williams' `rattle` package [77] provides a ribbon (tabbed-dialog box) style of user interface that is similar to that used by Microsoft Office. Although its emphasis is on data mining, the interface is useful for standard statistical analyses as well. Its name stands for the *R analytical tool to learn easily*. That name fits it well, as it is very easy to learn. Its point-and-click interface writes and executes R programs for you.

Before you install the `rattle` package, you must install some other tools. See the Web site for directions `http://rattle.togaware.com`. Once it is installed, you load it from your library in the usual way.



```
load("C:/
summary(m
```

```
> load("C:/myRfolder/mydata.RData")

> summary(mydata)
 workshop  gender        q1              q2              q3       
 R  :4     f   :3   Min.   :1.00   Min.   :1.00   Min.   :2.000  
 SAS:4     m   :4   1st Qu.:2.00   1st Qu.:1.00   1st Qu.:4.000  
           NA's:1   Median :3.50   Median :2.50   Median :4.000  
                    Mean   :3.25   Mean   :2.75   Mean   :4.143  
                    3rd Qu.:4.25   3rd Qu.:4.25   3rd Qu.:5.000  
                    Max.   :5.00   Max.   :5.00   Max.   :5.000  
                                                  NA's   :1.000  
```

```
[1] NOTE: R Commander Version 1.6-3: Sat Jan 29 15:05:59 2011
[2] WARNING: The Windows version of the R Commander works best 
with the single-document interface (SDI); see ?Commander.
```

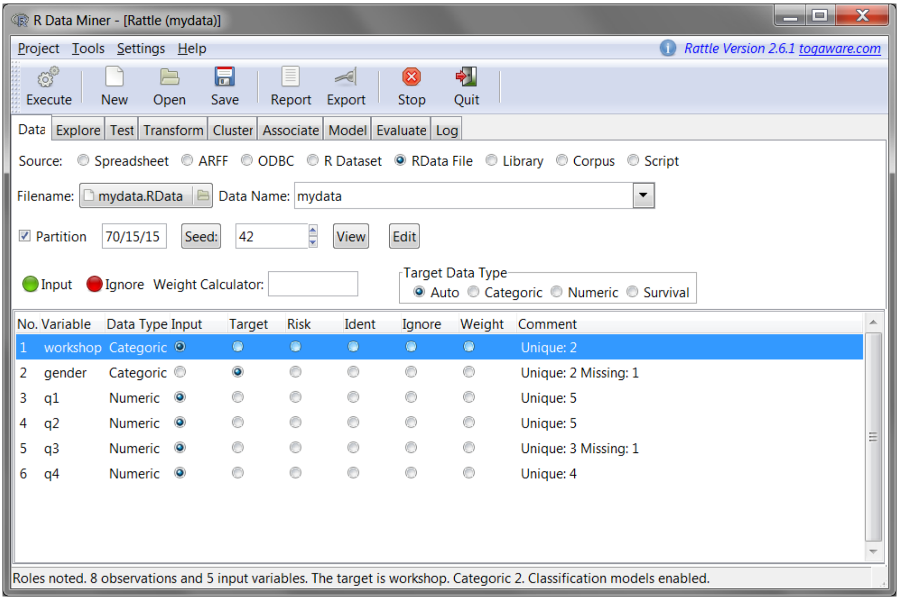
Fig. 3.13. The R Commander user interface with work in progress

```
> library("rattle")

Rattle, Graphical interface for data mining
  using R, Version 2.2.64.

Copyright (C) 2007 Graham.Williams@togaware.com, GPL

Type "rattle()" to shake, rattle, and roll your data.
```

As the instructions tell you, simply enter the call to the `rattle` function to bring up its interface:

```
> rattle()
```

The main Rattle interface shown in Fig. 3.14 will then appear. It shows the steps it uses to do an analysis on the tabs at the top of its window. You move from left to right, clicking on each tab to do the following steps. When



Source: ○ Spreadsheet ○ ARFF ○ ODBC ○ R Dataset ● RData File ○ Library ○ Corpus ○ Script

Filename: mydata.RData Data Name: mydata

☑ Partition 70/15/15 Seed: 42 View Edit

● Input ● Ignore Weight Calculator:

Target Data Type: ● Auto ○ Categoric ○ Numeric ○ Survival

No.	Variable	Data Type	Input	Target	Risk	Ident	Ignore	Weight	Comment
1	workshop	Categoric	●	○	○	○	○	○	Unique: 2
2	gender	Categoric	○	●	○	○	○	○	Unique: 2 Missing: 1
3	q1	Numeric	●	○	○	○	○	○	Unique: 5
4	q2	Numeric	●	○	○	○	○	○	Unique: 5
5	q3	Numeric	●	○	○	○	○	○	Unique: 3 Missing: 1
6	q4	Numeric	●	○	○	○	○	○	Unique: 4

Roles noted. 8 observations and 5 input variables. The target is workshop. Categoric 2. Classification models enabled.

Fig. 3.14. The `rattle` user interface for data mining

you are ready to run a particular step, click on the *Execute* icon in the upper left corner of the screen.

1. Data. Choose your data type from a comma-separated value (CSV) file, attribute-relation file format (ARFF), open database connectivity (ODBC), .RData file, R data object already loaded or created before starting Rattle, or even manual data entry.

   Here you also choose your variables and the roles they play in the analysis. I have chosen gender as the target variable (dependent variable) and the other variables as inputs (independent variables or predictors).
2. Explore. Examine the variables using summary statistics, distributions, interactive visualization via GGobi, correlation, hierarchical cluster analysis of variables, and principal components. A very interesting feature in distribution analysis is the application of Benford's law, an examination of the initial digits of data values that people use to detect fraudulent data (e.g., faked expense account values.)
3. Test. Perform standard analysis such as Kruskal–Wallis, Wilcoxon rank-sum, t-test, F-test, correlation and Wilcoxon signed-rank.
4. Transform. Here you can perform data tasks such as recoding, rescaling, taking logarithms, converting to ranks, replacing missing values with reasonable estimates (imputation).
5. Cluster. Perform various types of cluster analyses.

6. Associate. Perform association rule analysis to find relationships among observations or variables.
7. Model. Apply models from tree, boost, forest, SVM, regression, neural networks, or survival analysis.
8. Evaluate. Assess model quality and compare different models using confusion tables, lift charts, ROC curves, and so forth.
9. Log. See the R program that Rattle wrote for you to do all of the steps.

For more details, see Williams' book *Data Mining with Rattle and R* [78].

3.11.4 Red-R

Flowchart-style GUIs have been steadily growing in popularity. SAS Enterprise Guide and Enterprise Miner both use this approach, as does IBM SPSS Modeler (formerly Clementine). One implementation of this approach[1] for R is called Red-R [57] and is available at `http://www.red-r.org/`. Figure 3.15 shows an example analysis using Red-R.

Red-R comes with a set of icons called *widgets* that represent common steps in data acquisition, management, graphing, analysis, and presentation. The left side of the Red-R screen contains a *widget toolbar*. Clicking on a widget there makes it appear in its flowchart area or *schema* where you can move it to any position. The little "bumps" on either side of the widgets are called *slots*. You use your mouse to click and drag to connect the *output slot* on the right side of one widget to the *input slot* on the left side of another. That causes the data to flow in that direction. Double-clicking on an icon brings up a dialog box very similar to those in the SPSS, R Commander, and Deducer GUIs. The main difference is that these dialog boxes save their settings so you can use the same schema in different ways. Each dialog contains a button to activate it (often labeled *commit*). The graphs are interactive, so selecting points in a graph will cause only those points to be transferred *downstream* to the next widget. When you are finished, the whole flowchart is saved with its settings and the R code that each node used to do its work.

For example, in Fig.3.15 moving from left to right, I read an R data file, split it by gender, and got summary statistics for each group. Then I did a scatter plot on the males followed by a linear regression.

While flowchart-style user interfaces take a little longer to learn than those that focus on menus and dialog boxes, they do offer several important advantages:

You can get the big picture about an analysis with a quick glance.

Flowcharts are a time-honored approach to help simplify the construction of complex programs.

[1] Another is AnalyticFlow, but it does not appear to have as much development support: `http://www.ef-prime.com/products/ranalyticflow_en/`.

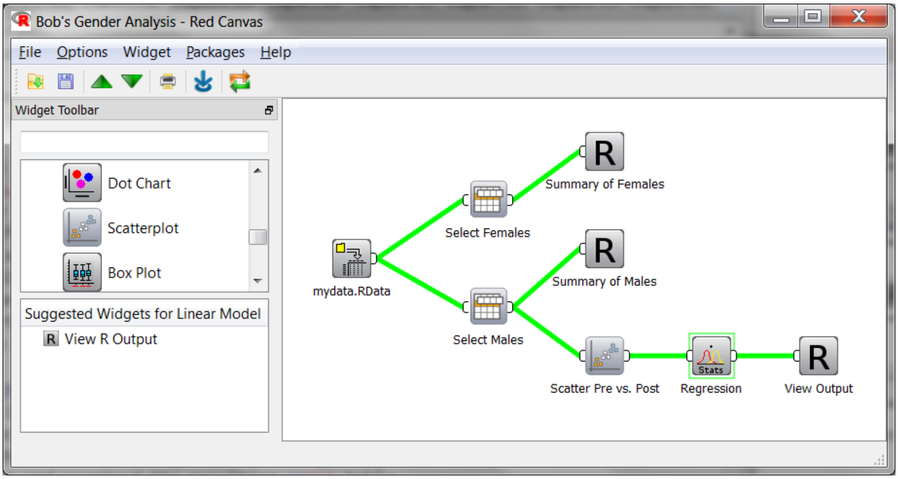


Fig. 3.15. The Red-R flowchart-style graphical user interface

You have an audit trail of what was done. I frequently have clients claim to have done the "same" analysis twice using the menu-and-dialog-box approach common to SPSS, R Commander and Deducer. However, the results do not match and they want to know why! Unless they saved the program generated by that approach, there is no way to know. People using that type of interface often do not save the program because programming does not interest them. With the flowchart approach, the audit trail is maintained and it is in a form its creator understands.

You can reuse an analysis on new data easily without resorting to programming. People use GUIs to avoid programming in the first place; they do not like to switch to a program even to change the first line and point it to a new data set. While GUIs may never offer as much power and flexibility as programming does, at least this approach gives you a considerable level of control.

4

Help and Documentation

R has an extensive array of help files and documentation. However, they can be somewhat intimidating at first, since many of them assume you already know a lot about R.

To see how R's help files differ in style from those of SAS and SPSS, let us examine the help file for the `print` function. The help file in R says you can use the `print` function to "Print Values," which is clear enough. However, it then goes on to say that "`print` prints its argument and returns it invisibly (via `invisible(x)`). It is a generic function which means that new printing methods can be easily added for new classes."

That requires a much higher level of knowledge than does the SPSS description of its similar command: "LIST displays case values for variables in the active dataset." However, when you are done with this book, you should be able to understand most help files well.

4.1 Starting Help

You can start the help system by choosing *Help> HTML Help* in Windows or *Help> R Help* in Mac OS. In any operating system you can submit the `help.start` function in the R console:

```
help.start()
```

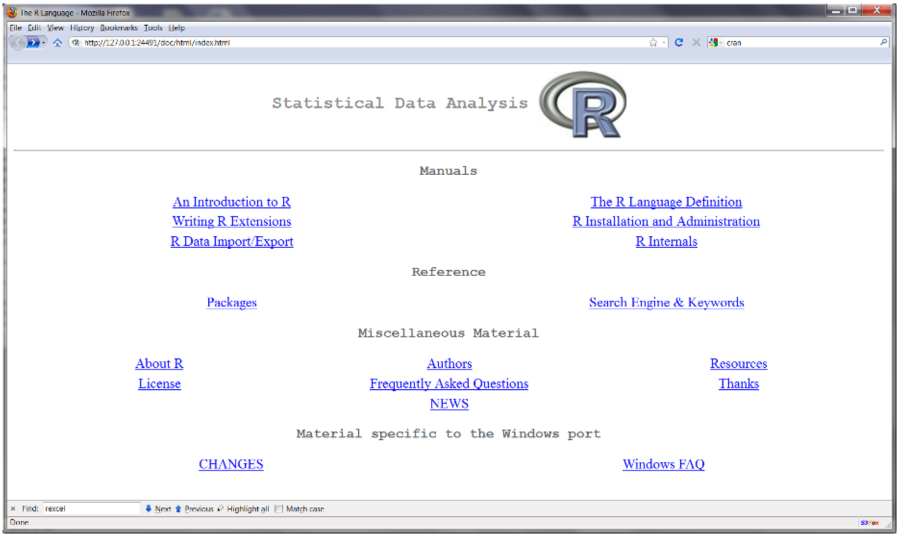
That is how Linux/UNIX users start it since they lack menus. Regardless of how you start it, you will get a help window that looks something like Fig. 4.1. To get help for a certain function such as `summary`, use the form:

```
help("summary")
```

or prefix the topic with a question mark:

```
?"summary"
```

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_4, © Springer Science+Business Media, LLC 2011


The R Language - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:24491/doc/html/index.html
cran
× Find: rexcel
Next Previous Highlight all Match case
Done

Statistical Data Analysis

Manuals

An Introduction to R
Writing R Extensions
R Data Import/Export
The R Language Definition
R Installation and Administration
R Internals

Reference

Packages
Search Engine & Keywords

Miscellaneous Material

About R
License
Authors
Frequently Asked Questions
NEWS
Resources
Thanks

Material specific to the Windows port

CHANGES
Windows FAQ

Fig. 4.1. R's main help window

The quotes around your search string are often optional. However, when requesting help on an operator, you must enclose it in quotes. For example, to get help on the assignment operator (equivalent to the equal sign in SAS or SPSS), enter:

```
help( "<-" )
```

This also applies to flow-control statements such as `if`, `else`, `for`, `in`, `repeat`, `while`, `break`, and `next`. For example, if we try to get help regarding the `while` function without putting it in quotes, we get an error message:

```
> help(while)

Error: unexpected ) in "help(while)"

> help("while")
  [help will appear]
```

Although it is a bit of extra typing, you can always put quotes around the item for which you are searching. If you do not know the name of a command or operator, use the `help.search` function to search the help files:

```
help.search("your search string")
```

A shortcut to the `help.search` function is to prefix the term with two question marks: "??". For a single word search, use this form:

```
??"yourstring"
```

For a string with more than one term in it, you must enclose it in quotes:

```
??"your multi-word string"
```

A particularly useful help file is the one on extracting and replacing parts of an object. That help file is opened with the following function call (the "E" in "Extract" is necessary):

```
help("Extract")
```

It is best to read that file after you have read Chapter 9, "Selecting Variables and Observations."

4.2 Examples in Help Files

Most of R's help files include examples that will execute. You can cut and paste them into a script window to submit in easily understood pieces. You can also have R execute all of the examples at once with the `example` function. Here are the examples for the `mean` function, but do not try to understand them now. We will cover the `mean` function later.

```
> example("mean")

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))

[1] 8.75 5.50

mean> mean(USArrests, trim = 0.2)

  Murder  Assault UrbanPop     Rape
    7.42   167.60    66.20    20.16
```

R changes its prefix of each example command from "`>`" to "`mean>`" to let you know that it is still submitting examples from the `mean` function's help files. Note that when a help file example is labeled "Not run," it means that while it is good to study, it will not run unless you adapt it to your needs.

A very nice feature of the JGR GUI is that you can execute most help file example programs by submitting them directly from the help window. You simply select the part you wish to run, right-click on the selection, and then choose "run line or selection." See Sect. 3.10.2, "JGR Java GUI for R," for details.

In SAS and SPSS, the help files include documentation for add-on packages that you might not have installed. However, in R you must first install a package and then load it from your library before you can get help. So you cannot use help to find things that you do not already know something about.

A popular addition to R is Harrell's Hmisc package [32]. It has many useful functions that add SAS-like capabilities to R. One of these is the contents function. Let us try to get help on it before loading the Hmisc package.

```
> help("contents")

No documentation for  contents  in specified packages
and libraries: you could try  ??contents
```

The help system does not find it, but it does remind you how you might search the help files. However, that search would find the contents function only if the Hmisc package were already installed (but not necessarily loaded). If you did not already know that Hmisc had such a function, you might search the Internet (or read a good book!) to find it. Let us now load the Hmisc package from our library.

```
> library("Hmisc")
```

R responds with a warning. We will discuss what this means later, but it does not cause a problem now.

```
Attaching package:  Hmisc

  The following object(s) are masked from package:base :
       format.pval,
       round.POSIXt,
       trunc.POSIXt,
       units
```

Now that the Hmisc package is loaded, we can get help on the contents function with the command help("contents"). We do not need to look at the actual help file at the moment. We will cover that function much later.

If you want help on a topic and you are not sure of its exact name you can use the help.search function. Let us use it to find things that relate to the string "contents."

```
> help.search("contents")

Help files with alias or concept or title matching  contents ...
fuzzy matching:

  anchors::replace.list   Updating contents of one list using...
  ape::GC.content         Content in GC from DNA Sequences
  DAAG::ironslag          Iron Content Measurements
  geoR::ca20              Calcium content in soil samples...
  ...
  Hmisc::contents         Metadata for a Data Frame
  MASS::abbey             Determinations of Nickel Content
  MEMSS::Milk             Protein content of cows milk
  multcomp::fattyacid     Fatty Acid Content of Bacillus...
  nlme::Milk              Protein content of cows milk
  PASWR::Bac              Blood Alcohol Content...
```

If you knew you wanted to use a function named `contents` but forgot which package(s) had a function of that name, this is a good way to find it.

4.3 Help for Functions That Call Other Functions

R has functions that exist to call other functions. These are called *generic functions*. In many cases, the help file for the generic function will refer you to those other functions, providing all of the help you need. However, in some cases you need to dig for such help in other ways. We will discuss this topic in Chap. 5 "Programming Language Basics", Sect. 5.7.4, "Controlling Functions with an Object's Class." We will also examine an example of this in Chap. 15, "Traditional Graphics," Sect. 15.10.9, "Scatter Plot Matrices."

4.4 Help for Packages

Thus far we have examined ways to get help about a specific function. You can also get help on an entire package. For example, the `foreign` package [14] helps you import data from other software. You can get help on a package itself by using the `package` argument. Here is a partial listing of its output:

```
> help(package = "foreign")

               Information on package  foreign
Description:
Package:       foreign
Priority:      recommended
Version:       0.8-41
Date:          2010-09-23
Title:         Read Data Stored by...SAS, SPSS, Stata,
Depends:       R (>= 2.10.0), stats
Imports:       methods, utils
Maintainer:    R-core <R-core@r-project.org>
Author:        R-core members, Saikat DebRoy, Roger Bivand...
```

```
                file in the sources.
Description:    Functions for reading and writing data stored by
                statistical packages...SAS, SPSS, Stata, ...
Index:
read.dta        Read Stata Binary Files
read.spss       Read an SPSS Data File
read.ssd        Obtain a Data Frame from a SAS Perm. Dataset...
read.xport      Read a SAS XPORT Format Library
write.foreign   Write Text Files and Code to Read Them...
```

To get help on a package, you must first install it, but you need not load it. However, not all packages provide help for the package as a whole. Most do, however, provide help on the functions that the package contains.

4.5 Help for Data Sets

If a data set has a help file associated with it, you can see it with the `help` function. For example,

```
help("esoph")
```

will tell you that this data set is "data from a case-control study of esophageal cancer in Ile-et-Vilaine, France."

4.6 Books and Manuals

Other books on R are available free at http://cran.r-project.org/ under documentation. We will use a number of functions from the `Hmisc` package. Its manual is *An Introduction to S and the Hmisc and Design Libraries* [2] by Alzola and Harrell. It is available at http://biostat.mc.vanderbilt.edu/twiki/pub/Main/RS/sintro.pdf. The most widely recommended advanced statistics book on R is *Modern Applied Statistics with S* (abbreviated MASS) by Venables and Ripley [65]. Note that R is almost identical to the S language and recently published books on S usually point out what the differences are.

An excellent book on managing data in R is Spector's *Data Manipulation with R* [51]. We will discuss books on graphics in the chapters on that topic.

4.7 E-mail Lists

There are different e-mail discussion lists regarding R that you can read about and sign up for at http://www.r-project.org/ under *Mailing Lists*. I recommend signing up for the one named *R-help*. There you can learn a lot by reading answers to the myriad of questions people post there.

If you post your own questions on the list, you are likely to get an answer in an hour or two. However, please read the posting guide, http://www.R-project.org/posting-guide.html, before sending your first question. Taking the time to write a clear and concise question and providing a descriptive subject line will encourage others to take the time to respond. Sending a small example that demonstrates your problem clearly is particularly helpful. See Chap. 12, "Generating Data," for ways to make up a small data set for that purpose. Also, include the version of R you are using and your operating system. You can generate all of the relevant details using the sessionInfo function:

```
> sessionInfo()

R version 2.12.1 (2010-12-16)
Platform: i386-pc-mingw32/i386 (32-bit)

locale:
[1] LC_COLLATE=English_United States.1252...

attached base packages:
[1] splines stats graphics grDevices utils datasets methods
[8] base

other attached packages:
[1] Hmisc_3.8-3  survival_2.36-2 prettyR_1.8-6

loaded via a namespace (and not attached):
[1] cluster_1.13.2  grid_2.12.1  lattice_0.19-17 tools_2.12.1
```

4.8 Searching the Web

Searching the Web for information on R using generic search engines such as Google can be frustrating, since the letter R refers to many different things. However, if you add the letter R to other keywords, it is surprisingly effective. Adding the word "package" to your search will also narrow it down. For example, to find packages on cluster analysis, you could search for "R cluster package" (without the quotes!).

An excellent site that searches just for R topics is Jonathon Barron's *R Site Search* at http://finzi.psych.upenn.edu/search.html. You can search just the R site while in R itself by entering the RSiteSearch function

```
RSiteSearch("your search string")
```

or by going to http://www.r-project.org/ and clicking *Search*.

4.9 Vignettes

Another kind of help is a *vignette*, a short description. People who write packages can put anything into its vignette. The command

```
vignette(all = TRUE)
```

will show you vignettes for all of the packages you have installed. To see the vignette for a particular package, enter it in the `vignette` function with its name in quotes:

```
vignette("mypackage")
```

Unfortunately, many packages do not have vignettes.

4.10 Demonstrations

Some packages include demonstrations, or "demos," that can help you learn how to use them by showing you actual running examples. You can see a list of them by entering the `demo()` function. Not many packages include demos, but when they do they are usually worth running. See Sect. 14.8 for some examples.

5

Programming Language Basics

5.1 Introduction

In this chapter we will go through the fundamental features in R. It will be helpful if you can download the book's files from the Web site `http://r4stats.com` and run each line as we discuss it. Many of our examples will use our practice data set described in Sect. 1.7.

R is an object-oriented language. Everything that exists in it – variables, data sets, functions (procedures) – are all objects.

Object names in R can be any length consisting of letters, numbers, underscores "_," or periods "." and should begin with a letter. However, in R if you always put quotes around a variable or data set name (actually any object name), it can then contain any characters, including spaces.

Unlike SAS, the period has no meaning in the name of a data set. However, given that my readers will often be SAS users, I avoid using the period.

Case matters in R, so you can have two variables – one named *myvar* and another named *MyVar* – in the same data set, although that is not a good idea! Some add-on packages tweak function names like the capitalized "`Save`" to represent a compatible, but enhanced, version of a built-in function like the lowercased "`save`." As in any statistics package, it is best to avoid names that match function names like "`mean`" or that match logical conditions like "TRUE."

While in SAS you perform analyses using procedures and in SPSS you use commands, in R you perform analyses using *functions*. When you execute a function, you are said to *call* it. The resulting output is what the function call *returns*. A few functions do not return output but have side effects such as writing an external file.

Function calls can begin and end anywhere on a line and R will ignore any additional spaces. R will try to execute a function call when it reaches the end of a line. Therefore, to continue a function call on a new line, you must ensure that the fragment you leave behind is not already a complete function call by itself. Continuing a function call on a new line after a comma

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_5, © Springer Science+Business Media, LLC 2011

is usually a safe bet. As you will see, R functions separate their parameters or *arguments* using commas, making them a convenient stopping point. The R console will tell you that it is continuing a line when it changes the prompt from “>” to “+”. If you see “+” unexpectedly, you may have simply forgotten to add the final close parenthesis, “)”. Submitting only that character will then finish your function call. If you are getting the “+” and cannot figure out why, you can cancel the pending function call with the Escape key on Windows or CTRL-C on Macintosh or Linux/UNIX. For CTRL-C, hold the CTRL key down (Linux/UNIX) or the control key (Macintosh) while pressing the letter C. You may end any R function call with a semicolon. This is not required, though, except when entering multiple function calls on a single line.

5.2 Simple Calculations

Although few people would bother to use R just as a simple calculator, you can do so with commands like

```
> 2+3

 [1] 5
```

The “[1]” tells you the resulting value is the first result. It is only useful when your results run across several lines. We can tell R to generate some data for us to see how the numbering depends on the width of the output. The form 1:50 will generate the integers from 1 to 50.

```
> 1:50

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
[20] 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
[39] 39 40 41 42 43 44 45 46 47 48 49 50
```

Now, it is obvious that the numbers in square brackets are counting or indexing the values. I have set the line width to 63 characters to help things fit in this book. You can use the options function to change the width to 40 and see how the bracketed numbers change.

```
> options(width = 40)

> 1:50

 [1]  1  2  3  4  5  6  7  8  9 10 11 12
 [13] 13 14 15 16 17 18 19 20 21 22 23 24
 [25] 25 26 27 28 29 30 31 32 33 34 35 36
 [37] 37 38 39 40 41 42 43 44 45 46 47 48
 [49] 49 50
```

```
> options(width = 63)  # Set it wider again.
```

In SAS, that setting is done with OPTIONS LINESIZE=63. SPSS uses SET WIDTH 63.

An important thing to keep in mind is that if you use your computer's mouse to shrink the R console window to have fewer columns, it will override the width setting.

You can assign the values to symbolic variables like x and y using the assignment operator, a two-character sequence "`<-`". You can use the equal sign as SAS and SPSS do, but there are some rather esoteric advantages[1] to using "`<-`" instead. Here we use it to assign values to x and y and then do some simple math.

```
> x <- 2

> y <- 3

> x + y
 [1] 5

> x * y
 [1] 6
```

We have added extra spaces in the above commands and extra lines in the output for legibility. Additional spaces do not affect the commands.

5.3 Data Structures

SAS and SPSS both use one main data structure, the *data set*. Instead, R has several different data structures including *vectors*, *factors*, *data frames*, *matrices*, *arrays*, and *lists*. The *data frame* is most like a data set in SAS or SPSS. R is flexible enough to allow you to create your own data structures, and some add-on packages do just that.

5.3.1 Vectors

A *vector* is an object that contains a set of values called *elements*. You can think of it as a SAS or SPSS variable, but that would imply that it is a column in a data set. It is not. It exists by itself and is neither a column nor a row. For R, it is usually one of two things: a variable or a set of parameter settings called *arguments* that you use to control functions. One of the more

[1] These are beyond our scope.

intriguing aspects of R is that its arguments are often more than single static character strings as they are in SAS and SPSS. The values of arguments are often vectors that happen to have single values. To SAS and SPSS users that is a radical idea. It will become clear as you read through this book.

Creating Vectors

Let us create a vector by entering the responses to the first question, "Which workshop did you take?" without any value labels:

```
workshop <- c( 1, 2, 1, 2, 1, 2, 1, 2 )
```

All of the workshop values are numeric, so the vector's *mode* is *numeric.* SAS and SPSS both refer to that as a variable's *type.* As in SAS and SPSS, if even one value were alphabetic (character or string), then the mode would be *coerced*, or forced, to be *character.* R does all its work with *functions*, which are similar to SAS *statements* and *procedures*, or SPSS *commands* and *procedures*. Functions have a name followed by its parameters (or *keywords* in SPSS jargon), called *arguments*, in parentheses. The `c` function's job is to *c*ombine multiple values into a single vector. Its arguments are just the values to combine, in this case 1,2,1,2. . . .

To print our vector, we can use the `print` function. This is R's equivalent to the SAS PRINT procedure or SPSS's LIST or PRINT statements. However, this function is used so often, it is the default function used when you *type the name* of any object! So when working interactively, these two commands do exactly the same thing:

```
> print(workshop)

 [1] 1 2 1 2 1 2 1 2

> workshop

 [1] 1 2 1 2 1 2 1 2
```

We run all of the examples in this book *interactively*; that is, we submit function calls and see the results immediately. You can also run R in *batch mode*, where you would put all your function calls into a file and tell R to run them all at once, routing the results to a file. In batch mode you must write out the `print` function. I will point out a few other instances when you must write out the `print` function name in later chapters. Although typing out the `print` function for most of our examples is not necessary, I will do it occasionally when showing how the R code looks in a typical analysis.

Let us create a character variable. Using R jargon, we would say we are going to create a *character vector*, or a vector whose *mode* is *character.* These are the genders of our hypothetical students:

```
> gender <- c("f", "f", "f", NA, "m", "m", "m", "m")

> gender

 [1] "f" "f" "f" NA "m" "m" "m" "m"
```

NA stands for Not Available, which R uses to represent missing values. Later I will read data from files whose values are separated by commas. In that case, R would recognize two commas in a row as having a missing value in between them. However, the values in the `c` function are its arguments. R does not allow its functions to have missing arguments. Entering NA gets around that limitation.

Even when entering character values for gender, never enclose NA in quotes. If you did, it would be just those letters rather than a missing value.

Now let us enter the rest of our data:

```
q1 <- c(1, 2, 2, 3, 4, 5, 5, 4)
q2 <- c(1, 1, 2, 1, 5, 4, 3, 5)
q3 <- c(5, 4, 4,NA, 2, 5, 4, 5)
q4 <- c(1, 1, 3, 3, 4, 5, 4, 5)
```

Using Vectors

Just as with variables in SAS or SPSS, you can do all kinds of things with vectors, like add them:

```
> mySum <- q1 + q2 + q3 + q4

> mySum

[1]  8  8 11 NA 15 19 16 19
```

That approach works just like SAS or SPSS since R added all the elements in order simply by using the "+" sign. In many other languages that process would have required a DO or FOR loop by creating the sums for the first element, then the second, and so on. You could do it that way in R too, but it is not necessary. The fact that R functions work on every element of vectors automatically is called *vectorization*. R's functions are *vectorized*, helping you to avoid needless and often inefficient DO or FOR loops.

While vectorization typically works just like SAS or SPSS, sometimes it will surprise you. For example, if you add two variables in SAS or SPSS and one is shorter than the other, those packages will force the two to match lengths by filling in missing values. Let us see what happens in R:

```
> myShortVector <- c(10, 100)

> q1
```

```
[1] 1 2 2 3 4 5 5 4

> mySum <- q1 + myShortVector

> mySum

[1]  11 102  12 103  14 105  15 104
```

What happened? Rather than set all but the first two values to missing, R used the 10 and 100 values over and over again. This process is called *recycling.* To SAS or SPSS users, recycling appears disastrous at first. Is it not adding values from the wrong observations? Do not fear, though, as in most cases vectors on the same set of observations will be padded with missing values. That will ensure the behavior you expected will indeed happen:

```
> myVector <- c(10, 100, NA, NA, NA, NA, NA, NA)

> mySum <- q1 + myVector

> mySum

[1]  11 102  NA  NA  NA  NA  NA  NA
```

Once you recover from the initial shock of these differing results, you will find that when you expect SAS- or SPSS-like results in addition or subtraction, you will have them. Furthermore, recycling offers you a way to simplify programs that would otherwise require the use of DO or FOR loops.

Most mathematical functions in R are vectorized. For example, the `sqrt` function will take the square root of each element in the vector, just as in SAS or SPSS:

```
> sqrt(q1)

[1] 1.000 1.414 1.414 1.732 2.000 2.236 2.236 2.000
```

Statistical functions work on a whole vector at once. To get a simple table of frequencies, we can use the `table` function:

```
> table(workshop)

workshop
1 2
4 4

> table(gender)
```

```
gender
f m
3 4
```

The first thing you will notice about the output is how plain it is. No percents are calculated and no lines drawn to form a table. When you first see a table like the one for workshop, its complete lack of labels may leave you wondering what it means. There are four people who took workshop 1 and four people who took workshop 2. It is not hard to understand – just a shock when you come from a package that labels its output better.

This is a difference in perspective between R and SAS or SPSS. R creates or *returns* output that other functions can use immediately. Other functions exist that provide more output, like percents. Still others format output into publication-quality form.

Let us get the mean of the responses to question 3:

```
> mean(q3)

 [1] NA
```

The result is NA, or Not Available! Many R functions handle missing values in an opposite manner from SAS or SPSS. R will usually provide output that is NA when performing an operation on data that contains *any* missing values. It will typically provide the answer you seek only when you tell it to override that perspective. There are several ways to do this in R. For the `mean` function, you set the *NA remove argument*, `na.rm`, equal to TRUE.

```
> mean(q3, na.rm = TRUE)

[1] 4.142857
```

R has most of the same mathematical (Table 10.2) and statistical (see Table 10.1) functions that SAS and SPSS do.

Selecting Vector Elements

So far we have performed a few simple analyses on entire vectors. You can easily select subsets using a method called *subscripting* or *indexing*.[2] You specify which of the vector's elements you want in square brackets following the vector's name. For example, to see the fifth element of q1, you enter

```
> q1[5]

[1] 4
```

[2] To be more precise, subscripting is done by using index values, logic, or names. However, people use subscripting and indexing interchangeably.

When you want to specify multiple elements, you must first combine them into a vector using the `c` function. Therefore, to see elements 5 through 8, you can use

```
> q1[c(5, 6, 7, 8)]

[1] 4 5 5 4
```

The colon operator, ":", can generate vectors directly, so an alternate way of selecting elements 5 through 8 is

```
> q1[5:8]

[1] 4 5 5 4
```

You can also insert logical selections. They generate logical vectors to perform your selection. R uses "==" for logical equivalence, not the equal sign or "EQ":

```
> q1[gender == "m"]

[1] NA  4  5  5  4
```

The spaces on either side of the "==" or any other logical operators improve program legibility. Usually the goal of any of these selection methods is to perform some analysis on a subset. For example, to get the mean response to item q1 for the males, we can use

```
> mean(q1[gender == "m"], na.rm = TRUE)

[1] 4.5
```

R's ability to select vector elements is very flexible. I will demonstrate how to apply these techniques toward selecting parts of other data structures in the sections that immediately follow. Later I will devote three entire chapters to showing how to apply these techniques to data sets in Chap. 7, "Selecting Variables," through Chap. 9, "Selecting Variables and Observations."

5.3.2 Factors

Two of the variables we entered above, workshop and gender, are clearly categorical. R has a special data structure called a *factor* for such variables. Regardless of whether a variable's original values are numeric or character, when a variable becomes a factor, its mode becomes *numeric*.

Creating Factors from Numeric Vectors

Before we create a factor, let us enter workshop again as a numeric vector and display its values.

```
> workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)

> workshop

[1] 1 2 1 2 1 2 1 2
```

Now let us perform two simple analyses.

```
> table(workshop)

workshop

1 2
4 4

> mean(workshop)

[1] 1.5
```

We see that four people took each workshop. We also see that the `mean` function happily returned the mean of the workshops, which is a fairly nonsensical measure for a categorical variable. R usually tries to do correct things statistically, but we have not yet told it that workshop is categorical.

Recall that to select elements of a vector you can use subscripting and place an index value in square brackets. For example, to choose the third element of gender, you can use

```
> gender[3]

[1] f

Levels: f m
```

To see the first two and the last two elements, you can subscript using those index values in a vector using the `c` function like this:

```
> gender[c(1, 2, 7, 8)]

[1] f f m m

Levels: f m
```

Let us now see the genders of the people who took the SAS workshop, which has a value of 2.

```
> gender[workshop == 2]

[1] "f" NA  "m" "m"
```

Now let us enter the variable again, convert it to a factor using the `factor` function, and display its values.

```
> workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)

> workshop <- factor(workshop)

> workshop

[1] 1 2 1 2 1 2 1 2

Levels: 1 2
```

I could have assigned the resulting factor to a new variable name (on the left side of the `<-`), of course. However, the name "workshop" that appears within the parentheses on the `factor` function call (the right side) must exist already.

After using the `factor` function, we see that the display of workshop values has an additional feature, the levels. Let us repeat our two analytic functions:

```
> table(workshop)

workshop

1 2
4 4

> mean(workshop)

[1] NA

Warning message:
In argument is not numeric or logical: returning NA
```

The output from the `table` function is identical, but now the `mean` function warns us that this is not a reasonable request and it returns a missing value of NA.

Now that workshop is a factor, we can check the genders of the people who took the SAS workshop (workshop 2) in two ways:

```
> gender[workshop == 2]

[1] "f" NA  "m" "m"
```

```
> gender[workshop == "2"]

[1] "f" NA  "m" "m"
```

The second example uses quotes around the 2 and it still works. This is due to the fact that the original numeric values are now also stored as value labels.

Now I will enter workshop again, this time using additional arguments in the factor function call to assign more useful value labels.

```
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)
workshop <- factor(
  workshop,
  levels = c( 1,   2,     3,      4),
  labels = c("R", "SAS", "SPSS", "Stata")
)
```

The factor function call above has three arguments:

1. The name of a vector to convert to a factor.
2. The levels or values that the data can have. This allows you to specify values that are not yet in the data. In our case, workshop is limited to the values 1 and 2, but we can include the values 3 and 4 for future expansion. I spaced the values out just to match the spacing of the labels below, but that is not required. Notice that these values are contained in the c function; therefore, they are actually a vector!
   The order you list the values in will determine their appearance order in output like frequency tables and graphs. The first one you list will determine the comparison level if you use the factor in modeling (see Sect. 10.8 for details).
   If the values have a numeric order like low, medium, or high, then you can use the ordered function rather than the factor function. It works almost identically but registers the variable as ordinal rather than simply categorical.
3. Optionally, the labels for the levels. The factor function will match the labels to the levels in the order in which they are listed in the function call. The order of the values in the data set is irrelevant. If you do not provide the labels argument, R will use the values themselves as the labels. If you supply them, the values must be nested within a call to the c function, making them a character vector.

Now when we print the data, they show us that the people in our practice data set have only taken workshops in R and SAS. It also lists the levels so you can see what labels are possible:

```
> workshop
```

```
 [1] R SAS R SAS R SAS R SAS

Levels: R SAS SPSS Stata
```

The `table` function now displays the workshop labels and how many people took each:

```
> table(workshop)

workshop

    R   SAS  SPSS Stata
    4     4     0     0
```

The labels have now replaced the original values. So to check the genders of the people who took the SAS workshop, we can no longer use the value 2.

```
> gender[workshop == 2]

factor(0)

Levels: Male Female
```

When we select based on the value label, it works.

```
> gender[workshop == "SAS"]

[1] Female <NA>   Male   Male

Levels: Male Female
```

Creating Factors from Character Vectors

You can convert character vectors to factors in a similar manner. Let us again enter gender as a character vector and print its values.

```
> gender <- c("f", "f", "f", NA, "m", "m", "m", "m")

> gender

 [1] "f" "f" "f" NA "m" "m" "m" "m"
```

Notice that the missing value, NA, does not have quotes around it. R leaves out the quotes to let you know that it is not a valid character string that might stand for something like North America.

If we are happy with those labels, we can convert gender to a factor by using the simplest call to the `factor` function:

```
> gender <- factor(gender)

> gender

 [1] f f f NA m m m m

Levels: f m
```

If, instead, we want nicer labels, we can use the longer form. It uses the same approach we used for workshop, but the values on the levels argument need to be in quotes:

```
> gender <- factor(
+   gender,
+   levels = c("m",    "f"),
+   labels = c("Male", "Female")
+ )

> gender

 [1] Female Female Female NA Male   Male   Male   Male

Levels: Male Female

> table(gender)

gender

  Male Female
     4      3
```

You now need to use the new labels when performing selections on gender. For example, to see which workshops the males took, this no longer works:

```
> workshop[gender == "m"]

[1] <NA>

Levels: R SAS SPSS STATA
```

Instead, specifying the new label of “Male” finds the workshops they took:

```
> workshop[gender == "Male"]

[1] <NA> R SAS R SAS

Levels: R SAS SPSS Stata
```

Note that the last line of output conveniently tells you all of the levels of the factor even though the males did not take all of the workshops.

We will examine factors and compare them to SPSS value labels and SAS formats in Sect. 11.1, *Value Labels or Formats (and Measurement Level)*.

For the remainder of the book we will use the shorter labels, "m" and "f."

5.3.3 Data Frames

The data structure in R that is most like a SAS or SPSS data set is the *data frame*. SAS and SPSS data sets are always rectangular, with *variables* in the columns and records in the rows. SAS calls these records *observations* and SPSS calls them *cases*. A data frame is also rectangular. In R terminology, the columns are called *vectors*, *variables*, or just *columns*. The rows are called *observations*, *cases*, or just *rows*.

A data frame is a generalized *matrix*, one that can contain both character and numeric columns. A data frame is also a special type of *list*, one that requires each *component* to have the same *length*. We will discuss matrices and lists in the next two sections.

We have already seen that R can store variables in vectors and factors. Why does it need another data structure? R can generate almost any type of analysis or graph from data stored in vectors or factors. For example, getting a scatter plot of the responses to q1 versus q4 is easy. R will pair the first number from each vector as the first (x,y) pair to plot and so on down the line. However, it is up to you to make sure that this pairing makes sense. If you sort one vector independently of the others, or remove the missing values from vectors independently, the critical information of how the pairs should form is lost. A plot will still appear, but it will contain a completely misleading view of the data. Sorting almost any two variables in ascending order independently will create the appearance of a very strong relationship. The data frame helps maintain this critical pairing information.

Creating a Data Frame

The most common way to create a data frame is to read it from another source such as a text file, spreadsheet, or database. You can usually do that with a single function call. We will do that later in Chap. 6, "Data Acquisition." For the moment, I will create one by combining the vectors and factors. The following is my program so far:

```
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)

workshop <- factor(workshop,
   levels = c( 1,   2,     3,      4),
   labels = c("R", "SAS", "SPSS", "Stata") )
```

```
gender <- c("f", "f", "f", NA, "m", "m", "m", "m")

gender <- factor(gender)

q1 <- c(1, 2, 2, 3, 4, 5, 5, 4)
q2 <- c(1, 1, 2, 1, 5, 4, 3, 5)
q3 <- c(5, 4, 4,NA, 2, 5, 4, 5)
q4 <- c(1, 1, 3, 3, 4, 5, 4, 5)
```

Now we will use the `data.frame` function to combine our variables (vectors and factors) into a data frame. Its arguments are simply the names of the objects we wish to combine.

```
> mydata <- data.frame(workshop, gender, q1, q2, q3, q4)

> mydata

  workshop gender q1 q2 q3 q4
1        R      f  1  1  5  1
2      SAS      f  2  1  4  1
3        R      f  2  2  4  3
4      SAS   <NA>  3  1 NA  3
5        R      m  4  5  2  4
6      SAS      m  5  4  5  5
7        R      m  5  3  4  4
8      SAS      m  4  5  5  5
```

Notice that the missing value for gender is now shown as "`<NA>`." When R prints data frames, it drops the quotes around character values and so must differentiate missing value NAs from valid character strings that happen to be the letters "NA."

If I wanted to rename the vectors as I created the data frame, I could do so with the following form. Here the vector "gender" will be stored in mydata with the name "sex" and the others will keep their original names. Of course, I could have renamed every variable using this approach.

```
mydata <- data.frame(workshop, sex = gender, q1, q2, q3, q4)
```

For the remainder of the book I will leave the variable name as "gender."

Although I had already made gender into a factor, the `data.frame` function will coerce all character variables to become factors when the data frame is created. You do not always want that to happen (for example, when you have vectors that store people's names and addresses.) To prevent that from occurring, you can add the `stringsAsFactors = FALSE` argument in the call to the `data.frame` function.

In SAS and SPSS, you do not know where variable names are stored or how. You just know they are in the data set somewhere. In R however, variable names are stored in the *names attribute* – essentially character vectors – within data objects. In essence, they are just another form of data that you can manipulate. We can display the names of a data frame using the `names` function:

```
> names(mydata)

[1] "workshop" "gender" "q1" "q2" "q3" "q4"
```

R data frames also have a formal place for an ID variable it calls the *row names attribute*. These names can be informative text labels like subject names, but, by default, they are sequential numbers stored as character values. The `row.names` function will display them:

```
> row.names(mydata)

 [1] "1" "2" "3" "4" "5" "6" "7" "8"
```

SAS and SPSS display sequential numbers like this in their data editors. However, those numbers are reassigned to new observations when you sort your data. Row names in R are more useful since sorting never changes their values. You can always use them to return your data to their original state by sorting on the row names. See Sec. 10.18, "Sorting Data Frames," for details.

SAS and SPSS users typically enter an ID variable containing an observation/case number or perhaps a subject's name. However, this variable is like any other unless you manually supply it to a procedure that identifies observations. In R, procedures that identify observations will do so automatically using row names. If you set an ID variable to be the row names *while reading a text file*, then variable's original name (id, subject, SSN, etc.) vanishes. Since functions that do things like identify outliers will use the information automatically, you usually do not need the name. We will discuss row names further when we read text files and in Sect. 10.6, "Renaming Variables (. . . and Observations)."

Selecting Components of Data Frames

There are several ways to select the components of a data frame. For now, we will focus on just two: selecting by subscripting and by a method called $ notation. We will save the other methods for later chapters.

Selecting Data Frame Components by Subscripting

While vectors and factors have only one-dimensional subscripts with which to select their elements, data frames have two-dimensional ones. These are in the form

```
mydataframe[rows, columns]
```

For example, you can choose the eighth observation's value of the sixth variable, q4, using

```
> mydata[8, 6]

[1] 5
```

If you leave out a row or column subscript, R will assume you want them all. So to select all of the observations for the sixth variable, I can use

```
> mydata[ ,6]

[1] 1 1 3 3 4 5 4 5
```

It so happens that the above example is selecting a vector. We saw ealier that we could add subscripts to the end of a vector to select a subset of it. So for variable q4, I can choose its fifth through eighth elements using

```
> q4[5:8]

[1] 4 5 4 5
```

In our data frame, `mydata[ , 6]` is the same vector as variable q4. Therefore, we can make this same selection by appending `[5:8]` to it:

```
> mydata[ , 6][5:8]

[1] 4 5 4 5
```

Selecting Data Frame Components Using $ Notation

Since the components of our data frame have names, I can also select them by name using the form

```
myDataFrameName$myComponentName
```

Therefore, to select q1 from mydata, I can use

```
> mydata$q1

[1] 1 2 2 3 4 5 5 4
```

The variable q1 is still a vector, so I can append index values to it to make further selections. To select the fifth through eighth values (the males), we can use

```
> mydata$q1[ 5:8 ]

[1] 4 5 5 4
```

As we will soon see, there are many other ways to select subsets of data frames. We will save the other methods for Chap. 7, "Selecting Variables," through Chap. 9, "Selecting Variables and Observations."

5.3.4 Matrices

A *matrix* is a two-dimensional data object that looks like a SAS or SPSS data set, but it is actually one long vector wrapped into rows and columns. Because of this, its values must be of the same mode, (e.g., all numeric or all character). This constraint makes matrices more efficient than data frames for some types of analyses, but their main advantage is that they lend themselves to the use of matrix algebra.

To use matrices in SAS, you could run PROC IML, transfer a data set from SAS into an IML matrix, and then begin working in a whole new syntax. When finished, you would transfer the results back into the main SAS environment. Alternatively, you could run a whole different program: SAS/IML Studio.

SPSS has its similar MATRIX environment, with its separate syntax.

Unlike with SAS and SPSS, matrices are an integral part of R. There is no special matrix procedure to activate. This tight level of integration is one of the things that attracts developers to R.

Creating a Matrix

The cbind function takes *c*olumns and *bind*s them together into a matrix:

```
> mymatrix <- cbind(q1, q2, q3, q4)

> mymatrix

     q1 q2 q3 q4
[1,]  1  1  5  1
[2,]  2  1  4  1
[3,]  2  2  4  3
[4,]  3  1 NA  3
[5,]  4  5  2  4
[6,]  5  4  5  5
[7,]  5  3  4  4
[8,]  4  5  5  5
```

As you can see, a matrix is a two-dimensional array of values. The numbers on the left side in brackets are the row numbers. The form [1,] means that it is row number one and the *lack of a number* following the comma means that R has displayed all of the columns.

We can get the dimensions of the matrix with the `dim` function:

```
> dim(mymatrix)

 [1] 8 4
```

The first dimension is the number of rows, 8, and the second is the number of columns, 4.

To create a matrix, you do not need to start with separate vectors as I did; you can create one directly with the `matrix` function. The `matrix` function call below has four arguments. The first argument is data, which you must enclose in a call to the `c` function. The next three specify the number of rows, columns, and whether or not you are entering the data by rows. If you leave the `byrow = TRUE` argument off, you would enter the data turned on its side. I prefer to enter it by rows since it looks more like the layout of SAS and SPSS data sets.

```
> mymatrix <- matrix(
+   c(1, 1, 5, 1,
+     2, 1, 4, 1,
+     2, 2, 4, 3,
+     3, 1, NA,3,
+     4, 5, 2, 4,
+     5, 4, 5, 5,
+     5, 3, 4, 4,
+     4, 5, 5, 5),
+   nrow = 8, ncol = 4, byrow = TRUE)

> mymatrix

     [,1] [,2] [,3] [,4]
[1,]    1    1    5    1
[2,]    2    1    4    1
[3,]    2    2    4    3
[4,]    3    1   NA    3
[5,]    4    5    2    4
[6,]    5    4    5    5
[7,]    5    3    4    4
[8,]    4    5    5    5
```

You can see that the result is the same as before, except that the columns are no longer named q1, q2, q3, q4. Now let us see what the `table`, `mean`, and `cor` functions do with matrices. I will use the earlier version of the matrix, so you will see the variable names.

```
> table(mymatrix)

mymatrix

1 2 3 4 5
6 4 4 8 9

> mean( mymatrix, na.rm = TRUE )

 [1] 3.266667

> cor( mymatrix, use = "pairwise" )

           q1         q2         q3         q4
q1  1.0000000  0.7395179 -0.1250000  0.9013878
q2  0.7395179  1.0000000 -0.2700309  0.8090398
q3 -0.1250000 -0.2700309  1.0000000 -0.2182179
q4  0.9013878  0.8090398 -0.2182179  1.0000000
```

The `table` function counts the responses *across all survey questions at once*! That is not something SAS or SPSS would usually do. It is odd, but not useless. We can see that nine times people strongly agreed (a value of 5) with any of the questions on our survey.

The `mean` function gets the mean response of them all. Again, it is not of much interest in our situation, but you might find cases where it would be of value.[3]

The `cor` function correlates each item with the others, which is a very common statistical procedure. The fact that the names q1, q1, etc. appear shows that we are using the version of the matrix we created by combining the vectors with those names.

If you put a matrix into a data frame, its columns will become individual vectors. For example, now that we have mymatrix, we can create our practice data frame in two ways. Both have an identical result:

```
mydata <- data.frame( workshop, gender, q1, q2, q3, q4 )
```

or

```
mydata <- data.frame( workshop, gender, mymatrix )
```

In our case, there is not much difference between the two approaches. However, if you had 100 variables already in a matrix, the latter would be much easier to do.

[3] For example, sensor arrays commonly measure a single variable, such as CO2 levels, across an latitude-longitude grid. It is convenient to store such data in a matrix and the mean CO2 level across that area would be a useful measure.

Selecting Subsets of Matrices

Like data frames, matrices have two dimensions. You can select a subset of a matrix by specifying the two index values in the form

```
mymatrix[rows, columns]
```

For example, I can choose the eighth row and the fourth column using

```
> mymatrix[8, 4]

q4
 5
```

I can choose the males, rows five through eight, and variables q3 and q4 using:

```
> mymatrix[5:8, 3:4]

     [,1] [,2]
[1,]    2    4
[2,]    5    5
[3,]    4    4
[4,]    5    5
```

In our discussion of vectors, you learned that you could select parts of a vector using only one-dimensional indices. For example, `q4[1:4]` selects the first four elements of vector q4. When you leave out one of the two index values for a matrix, you are selecting a vector. Therefore, I can do this very same example by appending `[1:4]` to `mymatrix[ ,4]` as in

```
> mymatrix[ ,4][1:4]

[1] 1 1 3 3
```

Most of the other methods we have used for selecting elements of vectors or factors work in a similar manner with matrices. An important one that does *not* work with matrices is the dollar format that we used with data frames. Even using the form of mymatrix that contained the column names this does not work:

```
> mymatrix$q4  # No good!

Error in mymatrix$q4 : $ operator is invalid for atomic vectors
```

A similar form places one name, or vectors of names, in the subscripts:

```
> mymatrix[ ,"q4"]

[1] 1 1 3 3 4 5 4 5
```

Since this latter form works with both matrices and data frames, people who frequently work with both tend to prefer it over the dollar format.

Table 5.1. Matrix functions

	R	SAS	SPSS
Add	`a + b`	`a + b`	`a + b`
Determinant	`det(a)$values`	`DET(a)`	`DET(a)`
Diagonal	`diag(a)`	`DIAG(a)`	`DIAG(a)`
Eigenvalues	`eigen(x)$values`	`EIGENVAL(x)`	`EVAL(a)`
Eigenvectors	`eigen(a)$vectors`	`EIGENVEC(a)`	Not a function
Inverse	`solve(a)`	`INV(a)`	`INV(a)`
Multiply	`a %*% b`	`a * b`	`a * b`
Transpose	`t(a)`	`a` or `T(a)`	`T(a)`

Matrix Algebra

Matrix algebra is a powerful tool for data analysis. It is used inside most of the functions that come with R. R users most often use matrix algebra when writing their own functions. Even if you do not plan to write your own complex functions, the fact that R has matrix algebra capabilities tightly integrated into it draws developers to R, resulting in a vast array of add-on packages that everyone can use.

You can use matrix algebra at any time using the same syntax as you would for any other part of your program. The function names are different of course, but that is all. For example, if I wish to swap the row and column positions in mymatrix, I can use the `t` function to transpose it:

```
> mymatrixT <- t(mymatrix)

> mymatrixT

   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
q1    1    2    2    3    4    5    5    4
q2    1    1    2    1    5    4    3    5
q3    5    4    4   NA    2    5    4    5
q4    1    1    3    3    4    5    4    5
```

There are many packages available from CRAN that extend R's substantial built-in capabilities. Bates and Maechler's `Matrix` package [6] is particularly useful when dealing with sparse or dense matrices. Although further use of matrix algebra is beyond the scope of this book, Table 5.1 contains a list of commonly used matrix algebra functions.

5.3.5 Arrays

Just as a matrix is a two-dimensional extension of a vector, an array is a multidimensional extension of a matrix. A three-dimensional array is a set of matrices layered like pages in a book. Each matrix layer has the same number of rows and columns. A four dimensional array would be like a set of books on a shelf, each containing the same number of pages, and so on.

Our practice data set is artificially balanced. Four people took the R workshop and four took the SAS workshop. Therefore, we could take the answers to their survey questions – the data must all be the same type, in this case numeric – and store them in an array. There would be two layers, one for each workshop. The first layer would be a matrix of the four people who took the R workshop, and the second layer would contain the four who took the SAS workshop.

However, in observational studies, people choose their own workshops, and so each set of data would have a different number of people. Therefore, we could not store the data in an array. The data would instead be stored in a typical data set (an R data frame) and could be viewed as a "ragged" array.

The use of arrays is beyond the scope of this book, except for the "ragged" kind, data frames.

5.3.6 Lists

A *list* is a very flexible data structure. You can use it to store combinations of any other objects, even other lists. The objects stored in a list are called its *components*. That is a broader term than *variables*, or *elements* of a vector, reflecting the wider range of objects possible.

You can use a list to store related sets of data stored in different formats like vectors and matrices (example below). R often uses lists to store different bits of output from the same analysis. For example, results from a linear regression would have equation parameters, residuals, and so on. See Chap. 17, "Statistics," for details.

You can also use lists to store sets of arguments to control functions. We will do that later when reading multiple lines of data per case from a text file. Since each record we read will contain a different set of variables – each with a different set of column widths – a list is a perfect way to store them. For an example, see Sect. 6.7, "Reading Fixed-Width Text Files, Two or More Records Per Case."

We will also store arguments when aggregating data by workshop and gender in Sect. 10.12, "Creating Summarized or Aggregated Data Sets."

Creating a List

Now let us store some data in a list. We can combine our variables (vectors) *and our matrix* into a list using the `list` function.

```
> mylist <- list(workshop, gender, q1, q2, q3, q4, mymatrix)
```

Now let us print it.

```
> mylist

 [[1]]
 [1] R   SAS R SAS R SAS R SAS
Levels: R SAS SPSS Stata

 [[2]]
 [1] f    f    f    <NA> m    m    m    m
Levels: f m

 [[3]]
 [1] 1 2 2 3 4 5 5 4

 [[4]]
 [1] 1 1 2 1 5 4 3 5

 [[5]]
 [1]  5  4  4 NA  2  5  4  5

 [[6]]
 [1] 1 1 3 3 4 5 4 5

 [[7]]
     q1 q2 q3 q4
 [1,]  1  1  5  1
 [2,]  2  1  4  1
 [3,]  2  2  4  3
 [4,]  3  1 NA  3
 [5,]  4  5  2  4
 [6,]  5  4  5  5
 [7,]  5  3  4  4
 [8,]  4  5  5  5
```

Notice how the vector components of the list print sideways now. That allows each component to have a different length, or even to have a totally different structure, like a matrix. Also notice that it counts the components of the list with an additional index value in double brackets [[1]], [[2]], etc. Then each component has its usual index values in single brackets.

Previously, when I added mymatrix to a data frame, the structure of the matrix vanished and the matrix columns became variables in the data frame. Here, though, the matrix is able to maintain its separate identity within the list.

Let us create the list again, this time naming each component. Another term for these optional component names are *tags*.

```
> mylist <- list(
+   workshop = workshop,
+   gender   = gender,
+   q1 = q1,
+   q2 = q2,
+   q3 = q3,
+   q4 = q4,
+   mymatrix = mymatrix)
```

Now when I print it, the names [[1]], [[2]], etc. are replaced by the names (or tags) I supplied.

```
> mylist

$workshop
[1] R SAS R SAS R SAS R SAS
Levels: R SAS

$gender
[1] f    f    f    <NA> m    m    m    m
Levels: f m

$q1
[1] 1 2 2 3 4 5 5 4

$q2
[1] 1 1 2 1 5 4 3 5

$q3
[1]  5  4  4 NA  2  5  4  5

$q4
[1] 1 1 3 3 4 5 4 5

$mymatrix
     q1 q2 q3 q4
[1,]  1  1  5  1
[2,]  2  1  4  1
[3,]  2  2  4  3
[4,]  3  1 NA  3
[5,]  4  5  2  4
[6,]  5  4  5  5
[7,]  5  3  4  4
[8,]  4  5  5  5
```

Selecting Components of a List

A data frame is a specific type of list, one whose components must all be of the same length. Therefore any method of selecting components that we discussed for data frames will also apply here. However, since the types of objects lists can store are broader, so too are the techniques for selecting their components.

Selecting Components of a List by Subscripting

To select the components from a list, you can always use the double-bracketed subscripts. For example, to select the vector containing gender, you can use:

```
> mylist[[2]]

[1] f    f    f    <NA> m    m    m    m

Levels: f m
```

Back when we first learned about vectors and factors, we selected parts of them by adding index values or logical selections in bracketed subscripts. Since we have selected a factor from our list, we can add those, too. Here we select observations 5 through 8:

```
> mylist[[2]][5:8]

[1] m m m m

Levels: f m
```

We can also select parts of a list using single brackets, but when we do so, the result will be *a list with a single component, not just a factor*!

```
> mylist[2]

$gender

[1] f    f    f    <NA> m    m    m    m

Levels: f m
```

Note that it lists the name “$gender,” which does look exactly like the way lists name their components. So what would happen now if we tried to select observations 5 through 8?

```
> mylist[2][5:8]  # Bad!
```

```
$<NA>
NULL

$<NA>
NULL

$<NA>
NULL

$<NA>
NULL
```

We got NULL results because `mylist[2]` is a list containing a *single* component – the factor gender – and we are asking for components 5 through 8. They do not exist!

If you wish to select multiple components from a list, you use the single brackets. Here we select the first three:

```
> mylist[1:3]

$workshop
[1] R   SAS R   SAS R   SAS R   SAS

Levels: R SAS

$gender
[1] f    f    f    <NA> m    m    m    m

Levels: f m

$q1
[1] 1 2 2 3 4 5 5 4
```

R's subscripting approach starts looking pretty confusing at this point, but do not worry. In future chapters you will see that selections usually look far more natural with variable names used to select columns and with logical selections choosing rows.

Selecting Components of a List Using $ Notation

Since I have named our list's components, I can make the same selections by using the form

```
myListName$myComponentName
```

Therefore, to select the component named q1, I can use:

```
> mylist$q1

[1] 1 2 2 3 4 5 5 4
```

You can also append index values in square brackets to our selections to choose subsets. Here I select mymatrix, then choose the fifth through eighth rows and third and fourth columns:

```
> mylist$mymatrix[ 5:8, 3:4 ]

     q3 q4
[1,]  2  4
[2,]  5  5
[3,]  4  4
[4,]  5  5
```

5.4 Saving Your Work

When learning any new computer program, always do a small amount of work, save it, and get completely out of the software. Then go back in and verify that you really did know how to save your work.

This is a good point at which to stop, clean things up, and save your work. Until you save your work, everything resides in the computer's main random access memory. You never know when a power outage might erase it. R calls this temporary work area its *workspace*. You want to transfer everything we have created from this temporary workspace to a permanent file on your computer's hard drive.

R's workspace is analogous to the SAS work library, except that it is in memory rather than in a temp space on your hard drive. In SPSS jargon it would simply be your unsaved data sets.

You can use the `ls` function to see all of the data objects you have created. If you put no arguments between the `ls` function's parentheses, you will get a list of all your objects. Another, more descriptive name for this function is `objects`. I use `ls` below instead of `objects` because it is more popular. That may be due to the fact that Linux and UNIX have a "ls" command that performs a similar function by listing your files.

If you have done the examples from the beginning of this chapter, here are the objects you will see in your workspace.

```
> ls()

 [1] "gender"   "mydata"   "mylist"   "mymatrix" "q1"
 [6] "q2"       "q3"       "q4"       "workshop" "x"
 [11] "y"
```

You want to save some of these objects to your computer's hard drive, but where will they go? The directory or folder that R will store files in is called its *working directory.* Unless you tell it otherwise, R will put any file you save into that directory. On Windows XP or earlier, this is: C:\Documents and Settings\username\My Documents. On Windows Vista or later, this is: C:\Users\Username\Documents. On Macintosh, the default working directory is /Users/username.

The `setwd` function *set*s your *w*orking *d*irectory, telling R where you would like your files to go. This is the equivalent to "X CD C:\myRfolder" in SAS and "CD C:\myRfolder" in SPSS. The `getwd` function *get*s your *w*orking *d*irectory for you to see.

```
> getwd()

[1] "C:/Users/Bob/Documents"

> setwd("C:/myRfolder")

> getwd()

[1] "C:/myRfolder"
```

Notice that R uses a *forward* slash in "C:/myRfolder." R can use forward slashes in filenames *even on computers running Windows*! The usual backslashes used in Windows file specifications have a different meaning in R, and in this context will generate an error message:

```
> setwd("C:\myRfolder")  # backslashes are bad in filenames!

Error in setwd("myRfolder") : cannot change working directory

In addition: Warning messages:
1:  \m  is an unrecognized escape in a character string
2: unrecognized escape removed from "\myRfolder"
```

The message warns you that R is trying to figure out what "\m" means. We will discuss why later.

So now you know what is in your workspace and where your working directory resides. You are ready to save your work. However, which objects should you save? Once you have combined the vectors into a data frame, you no longer need the individual vectors. I will save just our data frame, mydata, and the matrix of survey questions, mymatrix.

The `save` function writes the objects you specify, to the file you list as its last argument.

```
save(mydata, mymatrix, file = "mydata.RData")
```

While SAS and SPSS users typically only save one data set to a file (SAS has exceptions that are used by experts), R users often save multiple objects to a single file using this approach.

Rather than tell R what you *do* want to save, you could remove the objects that you do *not* want to save and then save everything that remains. We can remove the ones we do not want by listing them as arguments separated by commas on the `remove` function. It also has a more popular shorter name, `rm`.

```
> rm(x, y, workshop, gender, q1, q2, q3, q4, mylist)

> ls()

 [1] "mydata" "mymatrix"
```

The `save.image` function will save all objects in your workspace to the file you specify:

```
save.image(file = "myWorkspace.RData")
```

When you exit R, it will ask if you want to save your workspace. Since you saved it to a file you yourself have named, you can tell it no. The next time you start R, you can load your work with the `load` function:

```
> load("mydata.RData")
```

If you want to see what you have loaded, use the `ls` function:

```
> ls()

 [1] "mydata" "mymatrix"
```

For more details, see Chap. 13, "Managing Your Files and Workspace."

5.5 Comments to Document Your Programs

No matter how simple you may think a program is when you write it, it is good to sprinkle in comments liberally to remind yourself later what you did and why you did it.

SAS and SPSS both use the COMMENT command or the * operator to begin a comment. SAS ends them with semicolons and SPSS ends with periods. The /*...*/ style comments in SAS and SPSS allow you to place comments in the middle of a line between keywords or to block off many lines of programming that you want to "turn off" for debugging purposes.

As we have discussed briefly, R uses the # operator to begin a comment. R comments continue until the end of the line. No special character is required to end a comment. You can make your comments easier to read if you skip one space after each # character. If you add a comment to the end of a programming line, it also improves legibility if you skip two spaces before each # character. You can put comments in the middle of statements, but only if they continue to the end of the line:

```
# This comment is on its own line, between functions.

workshop <- c(1, 2, 1, 2,  # This comment is within arguments.
              1, 2, 1, 2)  # And this is at the end.
```

Unlike the SAS and SPSS /*. . . */ style comments, there is no way to comment out a whole block of code that you want to ignore. However, any text editor that works well with R can easily add the # character to (and later remove from) the front of each line in a selected block of code. For some examples, see Sect. 3.9, "Running R From Within Text Editors."

An alternative is to turn off a block of code by pretending to define a function. For example:

```
BigComment <- function(x)
{
   # Here is code I do not want to run,
   # but I might need to run it later.
mean(x, na.rm = TRUE)
sd(x,   na.rm = TRUE)
}
```

This is not a very good approach, though, since R is actually creating the function, so the code within it must be correct. Since the need to turn off blocks of code often arises from the need to debug the code, this is usually not very helpful!

5.6 Comments to Document Your Objects

Another helpful way to document your work is to store comments in the R objects you create. This is analogous to the SAS `LABEL` option on the `DATA` statement. That option provides a single location for all comments regarding the data set.

R's comment capability is more like SPSS's. In SPSS, to comment the whole data set, you would use the `DOCUMENT` or `ADD DOCUMENT` commands, or even the older `FILE LABEL` command. To comment an individual variable, you would create a custom variable attribute.

To store a comment in an object, you use the `comment` function:

```
comment(mydata) <- "Example data from R for SAS and SPSS Users"
```

Later you can view it using the `comment` function:

```
> comment(mydata)

[1] "Example data for R for SAS and SPSS Users"
```

Other functions that display object attributes will also display this comment. You can assign comments to vectors as well, but they do not display automatically in the output like variable labels, so I would not normally do so. We will discuss variable labels later in Sect. 11.2.

5.7 Controlling Functions (Procedures)

SAS and SPSS both control the output of their procedures through statements like GLM and related substatements such as CLASS to specify which variables are factors (categorical). Those statements have options that control exactly what appears in the output. Modeling statements have a formula syntax.

R has analogs to these options to control the output of its functions, plus a few unique ones. The output itself is called what the function *returns*.

5.7.1 Controlling Functions with Arguments

SAS and SPSS use options to control what procedures do. R does, too, using slightly different terminology. R uses *arguments* to control functions. Let us look at the help file for the `mean` function. The following command will call up its help file:

```
> help("mean")

mean                package:base                R Documentation

Arithmetic Mean

Description: Generic function for the (trimmed)
             arithmetic mean.
Usage:
     mean(x, ...)
     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:

x: An R object. Currently there are methods for numeric/logical
   vectors and date, date-time and time interval objects, and
   for data frames all of whose columns have a method...
```

```
trim: the fraction (0 to 0.5) of observations to be trimmed
  from each end of  x  before the mean is computed...

na.rm: a logical value indicating whether  NA  values should
       be stripped before the computation proceeds.

...: further arguments passed to or from other methods.

Value:

  For a data frame, a named vector with the appropriate method
  being applied column by column. If  trim  is non-zero,...

References: Becker, R. A., Chambers, J. M. and Wilks,...

See Also:  weighted.mean ,  mean.POSIXct ,  colMeans  for row...

Examples:
     x <- c(0:10, 50)
     xm <- mean(x)
     c(xm, mean(x, trim = 0.10))
     mean(USArrests, trim = 0.2)
```

In the section labeled *Usage*, the help file tells us that the overall form of the function is `mean(x, ...)`. That means you have to provide an R object represented by `x`, followed by arguments represented by "`...`". The *Default S3 Method* section tells us the arguments used by the `mean` function itself as well as their initial, or default, settings. So if you do not tell it otherwise, it will not trim any data (`trim = 0`) and will not remove missing values (`na.rm = FALSE`). Therefore, the presence of any missing values will result in the mean being missing or NA, too. The "`...`" is called the *triple dot* argument. It means that more arguments are possible, but the `mean` function will pass those along to other functions that it calls. We will see examples of that later.

The *Arguments* section gets into the details. It tells you that `x` can be a numeric data frame, numeric vector, or date vector. The `trim` argument tells R the percent of the extreme values to exclude before calculating the mean. It goes on to define what `na.rm` and "`...`" do.

We can run the `mean` function on our q3 variable by naming each argument. We deleted it previously with the `rm` function, but imagine that we had not done that. Here we call the function, naming its arguments in the order they appear in the help file and setting their values:

```
mean(x = q3, trim = .25, na.rm = TRUE)
```

The spaces around the equal signs and after each comma are optional, but they make the program easier to read. If you name all of the arguments, you can use them in any order:

```
mean(na.rm = TRUE, x = q3, trim = .25)
```

You can also run it by listing every argument in their proper positions but without the argument names:

```
mean(q3, .25, TRUE)
```

All of these approaches work equally well. However, people usually run R functions by listing the object to analyze first without saying `x =`, followed by the names and values of only those arguments they want to change:

```
mean(q3, trim = .25, na.rm = TRUE)
```

You can also abbreviate some argument names, but I strongly recommend against doing so. Some functions pass on arguments they do not recognize to other functions they control. As mentioned earlier in this section, this is indicated by "`...`" as a function's last argument in the help file. Once a function has started passing arguments on to other functions, it will pass them *all* on unless it sees the *full name* of an argument it uses!

People sometimes abbreviate the values TRUE or FALSE as T or F. This is a bad idea, as you can define T or F to be anything you like, leading to undesired results. You may avoid that trap yourself, but if you write a function that others will use, they may use those variable names. R will not allow you to redefine what TRUE and FALSE mean, so using those is safe.

The following is an example function call that uses several abbreviations. It will run, but I do not recommend using abbreviations.

```
mean(q3, t = .25, na = T)
```

A common error for R beginners is to try to call a function using just a set of comma-separated values, as in

```
> mean(1, 2, 3)

[1] 1
```

Clearly the means of those values is not 1! What is happening is those numbers are being supplied as values to the arguments in order. Therefore, it is the same as

```
mean(x = 1, trim = 2, na.rm = 3)
```

The values for `trim` and `na.rm` are invalid and ignored, so we have asked R to get the mean of the single value "1"! The solution is to combine the values into a vector before calling the `mean` function or by nesting one call within the other:

```
> mean( c(1, 2, 3) )

[1] 2
```

Since the colon operator creates a vector directly, the use of the c function would be redundant, and this works fine:

```
> mean( 1:3 )

[1] 2
```

5.7.2 Controlling Functions with Objects

In the previous section we viewed arguments as single values. That is what SAS and SPSS users are accustomed to. However, in R, arguments can be much more than that: they can be entire objects. The data are usually one of the arguments, and they are obviously more than a single value. However, objects can be used as other arguments, too. Let us return to our previous example where we first created workshop as a factor:

```
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)

workshop <- factor(
  workshop,
  levels = c( 1,   2,     3,      4),
  labels = c("R", "SAS", "SPSS", "Stata")
)
```

Notice how the levels and labels use the c function. The c function creates vectors, of course, so we are in fact providing vectors for those arguments. Therefore, we could do this just as easily in two steps:

```
  myLevels = c( 1,   2,     3,      4)
  myLabels = c("R", "SAS", "SPSS", "Stata")

workshop <- factor(
  workshop,
  levels = myLevels,
  labels = myLabels
)
```

The last part could be shortened to,

```
workshop <- factor(workshop, myLevels, myLabels)
```

In SAS and SPSS this type of control is called *macro substitution* and it involves a different syntax with different rules. In R, though, you can see that there is no different syntax. We were using a vector to supply those arguments before and we are doing so now, just in two steps.

5.7.3 Controlling Functions with Formulas

An important type of argument is the *formula*. It is the first parameter in functions that do modeling. For example, we can do linear regression, predicting q4 from the others with the following call to the `lm` function for *l*inear *m*odels:

```
lm( q4 ~ q1 + q2 + q3, data = mydata )
```

Some modeling functions accept arguments in the form of both formulas and vectors. For example, both of these function calls will compare the genders on the mean of the variable q1:

```
t.test( q1 ~ gender, data = mydata )

t.test( q1[ which(gender == "Female") ],
        q1[ which(gender == "Male")   ],
        data = mydata)  # Data ignored!
```

However, there is one very important difference. When using a formula, the `data` argument can supply the name of a data frame that R will search before looking elsewhere for variables. When not using a formula, as in the second example, the `data` argument is ignored! If q1 does not exist as a vector in your workspace or if you have not attached mydata, R will not find it. This approach maintains R's extreme flexibility while helping to keep formulas short.

The symbols that R uses for formulas are somewhat different from those used by SAS or SPSS. Table 17.1 shows some common examples.

5.7.4 Controlling Functions with an Object's Class

As we have seen, R has various kinds of data structures: vectors, factors, data frames, etc. The kind of structure an object is, is known as its *class*. Each data structure stores its class as an *attribute*, or stored setting, that functions use to determine how to process the object. In other words, R sees what you are giving it and it tries to do the right thing with it.

For objects whose *mode* is numeric, character, or logical, an object's class is the same as its mode. However, for matrices, arrays, factors, lists, or data frames, other values are possible (Table 5.2).

You can display an object's class with the `class` function:

```
> workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)

> class(workshop)

 [1] "numeric"
```

Table 5.2. Modes and classes of various R objects

Object	Mode	Class
Numeric vector	numeric	numeric
Character vector	character	character
Factor	numeric	factor
Data frame	list	data.frame
List	list	list
Numeric matrix	numeric	matrix
Character matrix	character	matrix
Model	list	lm...
Table	numeric	table

```
> summary(workshop)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     1.0     1.5     1.5     2.0     2.0
```

The class *numeric* indicates that this version of workshop is a numeric vector, not yet a factor. The summary function provided us with inappropriate information (i.e., the median workshop is nonsensical) because we have not yet told it it that workshop is a factor. Note that when we convert workshop into a factor, we are changing its class to factor, and then summary gives us the more appropriate counts instead:

```
> workshop <- factor(workshop,
+    levels = c( 1,  2,    3,     4),
+    labels = c("R","SAS","SPSS","Stata") )

> class(workshop)
 [1] "factor"

> summary(workshop)

    R   SAS   SPSS Stata
    4     4      0     0
```

When we first created gender, it was a character vector, so its class was character. Later we made its class factor. Numeric vectors like q1 have a class of numeric. The names of some other classes are obvious: factor, data.frame, matrix, list, and array. Objects created by functions have many other classes. For example, the linear model function, lm, stores its output in lists with a class of *lm*.

R has some special functions called *generic functions.* They accept multiple classes of objects and change their processing accordingly. These functions are tiny. Their task is simply to determine the class of the object and then pass it off to another that will do the actual work. The methods function will tell you what other functions a generic function will call. Let us look at the methods that the `summary` function uses.

```
> methods(summary)

 [1] summary.aov            summary.aovlist
 [3] summary.connection     summary.data.frame
 [5] summary.Date           summary.default
 [7] summary.ecdf*          summary.factor
 [9] summary.glm            summary.infl
 [11] summary.lm             summary.loess*
 [13] summary.manova         summary.matrix
 [15] summary.mlm            summary.nls*
 [17] summary.packageStatus* summary.POSIXct
 [19] summary.POSIXlt        summary.ppr*
 [21] summary.prcomp*        summary.princomp*
 [23] summary.stepfun        summary.stl*
 [25] summary.table          summary.tukeysmooth*
   Non-visible functions are asterisked
```

So when we enter `summary(mydata)`, the `summary` function sees that mydata is a data frame and then passes it on to the function named `summary.data.frame`. The functions marked with asterisks above are "nonvisible." They are meant to be used by a package's developer, not its end users. Visible functions can be seen by typing their name (without any parentheses). That makes it easy to copy and change them.

When we discussed the help files, we saw that the `mean` function ended with an argument of "`...`". That indicates that the function will pass arguments on to other functions. While it is very helpful that generic functions automatically do the "right thing" when you give it various objects to analyze, this flexibility complicates the process of using help files.

When written well, the help file for a generic function will refer you to other functions, providing a clear path to all you need to know. However, it does not always go so smoothly. We will see a good example of this in Chap. 15, "Traditional Graphics." The `plot` function is generic. When we call it with our data frame, it will give us a scatter plot matrix. However, to find out all of the arguments we might use to improve the plot, we have to use `methods(plot)` to find that `plot.data.frame` exists. We could then use `help("plot.data.frame")` to find that `plot.data.frame` calls the `pairs` function, then finally `help("pairs")` to find the arguments we seek. This is a worst-case scenario, but it is important to realize that this situation does occasionally arise.

As you work with R, you may occasionally forget the mode or class of an object you created. This can result in unexpected output. You can always use the `mode` or `class` functions to remind yourself.

5.7.5 Controlling Functions with Extractor Functions

Procedures in SAS and SPSS typically display all their output at once.[4] R has simple functions, like the `mean` function, that display all their results all at once. However, R functions that model relationships among variables (e.g., regression, ANOVA, etc.) tend to show you very little output initially. You save the output to a *model object* and then use *extractor functions* to get more information when you need it.

This section is poorly named from an R expert's perspective. Extractor functions do not actually control other functions the way parameters control SAS or SPSS output. Instead they show us what the other functions have already done. In essence, most modeling in R is done through its equivalent to the SAS Output Delivery System (ODS) or the SPSS Output Management System (OMS). R's output is in the form of data that can be readily manipulated and analyzed by other functions.

Let us look at an example of predicting q4 from q1 with linear regression using the `lm` function:

```
> lm( q4 ~ q1 + q2 + q3, data = mydata)

Call:
lm(formula = q4 ~ q1 + q2 + q3, data = mydata)

Coefficients:
(Intercept)           q1           q2           q3
    -1.3243       0.4297       0.6310       0.3150
```

The output is extremely sparse, lacking the usual tests of significance. Now, instead, I will store the results in a model object called myModel and check its class:

```
> myModel <- lm( q4 ~ q1 + q2 + q3, data = mydata )

> class(myModel)

 [1] "lm"
```

The `class` function tells us that myModel has a class of "*lm*" for *l*inear *m*odel. We have seen that R functions offer different results (methods) for different types (classes) of objects. So let us see what the `summary` function does with this class of object:

```
> summary(myModel)
```

[4] SAS has some interactive procedures that let you request additional output once you have seen the initial output, but SAS users rarely use it that way.

```
Call:
lm(formula = q4 ~ q1 + q2 + q3, data = mydata)

Residuals:
       1        2        3        5        6        7
-0.31139 -0.42616  0.94283 -0.17975  0.07658  0.02257
       8
-0.12468

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.3243     1.2877  -1.028    0.379
q1            0.4297     0.2623   1.638    0.200
q2            0.6310     0.2503   2.521    0.086
q3            0.3150     0.2557   1.232    0.306
---
Signif. codes:  0  ***  0.001  **  0.01  *  0.05  .  0.1

Residual standard error: 0.6382 on 3 degrees of freedom
  (1 observation deleted due to missingness)

Multiple R-squared: 0.9299,     Adjusted R-squared: 0.8598

F-statistic: 13.27 on 3 and 3 DF,  p-value: 0.03084
```

This is the type of output that SAS and SPSS prints immediately. There are many other extractor functions that we might use, including `anova` to extract an analysis of variance table, `plot` for diagnostic plots, `predict` to get predicted values, `resid` to get residuals, and so on. We will discuss those in Chap. 17, “Statistics.”

What are the advantages of the extractor approach?

You get only what you need, when you need it.

The output is in a form that is very easy to use in further analysis. Essentially the output itself is data!

You use methods that are consistent across functions. Rather than having to learning different ways of saving residuals or predicted values in every procedure SAS and SPSS do, you learn one approach that works with all modeling functions.

5.8 How Much Output There?

In the previous section we discussed saving output and using extractor functions to get more results. However, how do we know what an output object contains? Previously, the `print` function showed us what was in our objects, so let us give that a try. We can do that by simply typing an object's name or by explicitly using the `print` function. To make it perfectly clear that we are using the `print` function, let us actually type out its name.

```
> print(myModel)

Call:

lm(formula = q4 ~ q1 + q2 + q3, data = mydata)

Coefficients:

(Intercept)           q1           q2           q3
    -1.3243       0.4297       0.6310       0.3150
```

We see that the object contains the original function call complete with its arguments and the linear model coefficients. Now let us check the mode, class, and names of myModel.

```
> mode(myModel)
 [1] "list"

> class(myModel)
 [1] "lm"

> names(myModel)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"        "df.residual"
 [9] "na.action"     "xlevels"       "call"          "terms"
 [13] "model"
```

So we see that myModel is a list, or collection, of objects. More specifically, it is a list with a class of "lm." The `names` function shows us the names of all of the objects in it. Why did the `print` function not show them to us? Because the `print` function has a predetermined method for displaying lm class objects. That method says, basically, "If an object's class is lm, then print only the original formula that created the model and its coefficients."

When we put our own variables together into a list, it had a class of simply "list" (its mode was list also). The `print` function's method for that class tells it to print all of the list's components. We can strip away the class attribute of any object with the `unclass` function. In this case, it resets its class to "list." If we do that, then the `print` function will indeed print all of the list's components.

```
> print( unclass(mymodel) )

$coefficients
```

```
(Intercept)          q1          q2          q3
 -1.3242616   0.4297468   0.6310127   0.3149789

$residuals
          1           2           3           5           6
-0.31139241 -0.42616034  0.94282700 -0.17974684  0.07658228
          7           8
 0.02257384 -0.12468354

$effects
(Intercept)          q1          q2          q3
 -8.6931829   3.6733345  -1.4475844   0.7861009   0.2801541

  0.7929917  -0.7172223

$rank
[1] 4

$fitted.values
       1        2        3        5        6        7
1.311392 1.426160 2.057173 4.179747 4.923418 3.977426
       8
5.124684

$assign
[1] 0 1 2 3

$qr
$qr
  (Intercept)         q1         q2          q3
1  -2.6457513 -8.6931829 -7.9372539 -10.9609697
2   0.3779645  3.9279220  3.3096380  -0.3273268
3   0.3779645  0.1677124 -2.6544861   0.7220481
5   0.3779645 -0.3414626  0.4356232   2.4957256
6   0.3779645 -0.5960502 -0.3321400  -0.1051645
7   0.3779645 -0.5960502 -0.7088608   0.4471879
8   0.3779645 -0.3414626  0.4356232  -0.4186885
attr(,"assign")
[1] 0 1 2 3

$qraux
[1] 1.377964 1.167712 1.087546 1.783367

$pivot
[1] 1 2 3 4
```

```
$tol
[1] 1e-07

$rank
[1] 4

attr(,"class")
[1] "qr"

$df.residual
[1] 3

$na.action
4
4
attr(,"class")
[1] "omit"

$xlevels
list()

$call
lm(formula = q4 ~ q1 + q2 + q3, data = mydata)

$terms
q4 ~ q1 + q2 + q3
attr(,"variables")
list(q4, q1, q2, q3)
attr(,"factors")
   q1 q2 q3
q4  0  0  0
q1  1  0  0
q2  0  1  0
q3  0  0  1
attr(,"term.labels")
[1] "q1" "q2" "q3"
attr(,"order")
[1] 1 1 1
attr(,"intercept")
[1] 1
attr(,"response")
[1] 1
attr(,".Environment")
<environment: R_GlobalEnv>
```

```
attr(,"predvars")
list(q4, q1, q2, q3)
attr(,"dataClasses")
       q4        q1        q2        q3
"numeric" "numeric" "numeric" "numeric"

$model
  q4 q1 q2 q3
1  1  1  1  5
2  1  2  1  4
3  3  2  2  4
5  4  4  5  2
6  5  5  4  5
7  4  5  3  4
8  5  4  5  5
```

It looks like the `print` function was doing us a big favor by not printing everything! When you explore the contents of any object, you can take this approach or, given just the names, explore things one at a time. For example, we saw that myModel contained the object named “$coefficients.” One way to print one component of a list is to refer to it as mylist$mycomponent. So in this case we can see just the component that contains the model coefficients by entering

```
> myModel$coefficients

(Intercept)          q1          q2          q3
 -1.3242616   0.4297468   0.6310127   0.3149789
```

That looks like a vector. Let us use the `class` function to check:

```
> class( myModel$coefficients )

 [1] "numeric"
```

Yes, it is a numeric vector. So we can use it with anything that accepts such data. For example, we might get a bar plot of the coefficients with the following (plot not shown). We will discuss bar plots more in Chap. 15, “Traditional Graphics.”

```
> barplot( myModel$coefficients )
```

For many modeling functions, it is very informative to perform a similar exploration on the objects created by them.

5.9 Writing Your Own Functions (Macros)

In SAS or SPSS, if you wanted to use the same set of commands repeatedly, you would write a macro. In SPSS, you might instead write a macro-like Python program. Those approaches entail using languages that are separate from their main programming statements, and the resulting macros operate quite differently from the procedures that come with SAS or SPSS. In R, you write functions using the same language you use for anything else. The resulting function is used in exactly the same way as a function that is built into R.

I will show you some variations of a simple function, one that calculates the mean and standard deviation at the same time. For this example, I will apply it to just the numbers 1, 2, 3, 4, and 5.

```
> myvar <- c(1, 2, 3, 4, 5)
```

I will begin the function called *mystats* and tell it that it is a function of x. What follows in curly brackets is the function itself. I will create this with an error to see what happens.

```
# A bad function.
mystats <- function(x) {
  mean(x, na.rm = TRUE)
    sd(x, na.rm = TRUE)
}
```

Now let us apply it like any other function.

```
> mystats(myvar)

 [1] 1.5811
```

We got the standard deviation, but what happened to the mean? When I introduced the `print` function, I mentioned that usually you can type an object's name rather than, say, `print(myobject)`. Well, this is one of those cases where we need to explicitly tell R to print the result. I will add that to the function.

```
# A good function that just prints.
mystats <- function(x) {
  print( mean(x, na.rm = TRUE) )
  print(   sd(x, na.rm = TRUE) )
}
```

Now let us run it.

```
> mystats(myvar)
 [1] 3
 [1] 1.5811
```

That looks better. Next I will do it in a slightly different way, so that it will write our results to a vector for further use. I will use the c function to combine the results into a vector.

```
# A function with vector output.
mystats <- function(x) {
  mymean <- mean(x, na.rm = TRUE)
  mysd   <-   sd(x, na.rm = TRUE)
  c( mean = mymean, sd = mysd )
}
```

Now when I run it, we get the results in vector form.

```
> mystats(myvar)

  mean     sd
3.0000 1.5811
```

As with any R function that creates a vector, you can assign the result to a variable to use in any way you like.

```
> myVector <- mystats(myvar)

> myVector

  mean     sd
3.0000 1.5811
```

This simple result is far more interesting than it first appears. The vector has a name, “myVector,” but what are the strings “mean” and “sd”? At first they seem like value labels, but if we had another value 3.0 appear, it would not automatically get the label of “mean.” In addition, this is a numeric vector, not a factor, so they cannot be value labels.

These are *names*, stored in the names attribute, just like variable names in a data frame. But then what is “myVector”? That is just the vector’s name. To reduce confusion about names, these “value names” are called *tags* and this type of vector is called a *tagged vector*. I hardly ever create such names unless, as in this example, I am storing output.

Many R functions return their results in the form of a list. Recall that each member of a list can be any data structure. I will use a list to save the original data, as well as the mean and standard deviation:

```
# A function with list output.
mystats <- function(x) {
  myinput <- x
  mymean  <- mean(x, na.rm = TRUE)
  mysd    <-   sd(x, na.rm = TRUE)
  list(data = myinput, mean = mymean, sd = mysd)
}
```

Now I will run it to see how the results look.

```
mystats(myvar)

$data
 [1] 1 2 3 4 5

$mean
 [1] 3

$sd
 [1] 1.5811
```

You can save the result to mylist and then print just the data.

```
> myStatlist <- mystats(myvar)

> myStatlist$data
 [1] 1 2 3 4 5
```

If you want to see the function itself, simply type the name of the function without any parentheses following.

```
> mystats

function(x) {
  myinput <- x
  mymean  <- mean(x, na.rm = TRUE)
  mysd    <-   sd(x, na.rm = TRUE)
  list(data = myinput, mean = mymean, sd = mysd)
}
```

You could easily copy this function into a script editor window and change it. You can see and change many R functions in this way.

Coming from SAS or SPSS, `function` is perhaps the most unusual of all R functions. Its input is several functions and its output is one function. It is as if it is using functions as its data.

R has an ability to use a function without naming it. I show how to use these *anonymous functions* in Section10.2.4, “Applying Your Own Functions.”

5.10 Controlling Program Flow

R is a complete programming language with all the usual commands to control the flow of a program. These include the functions `if`, `else`, `for`, `in`, `repeat`, `while`, `break`, and `next`. So your programs and functions you write can get as complex as necessary. However, controlling the flow of commands is needed far less in R than in SAS or SPSS. That is because R has a way to *apply* functions automatically across variables or observations. For the purposes of this book, that type of flow control is all we need, and we get a *lot* done within those constraints! For details on applying functions repeatedly to variables or observations, see Sect. 10.2

5.11 R Program Demonstrating Programming Basics

Most of the chapters in this book end with equivalent example programs in SAS, SPSS, and R. However, this chapter focuses so much on R that I will end it only with the program for R.

```
# Filename: ProgrammingBasics.R

# ---Simple Calculations---
2 + 3

x <- 2
y <- 3
x + y
x * y

# ---Data Structures---

# Vectors
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)
print(workshop)
workshop

gender <- c("f", "f", "f", NA, "m", "m", "m", "m")
q1 <- c(1, 2, 2, 3, 4, 5, 5, 4)
q2 <- c(1, 1, 2, 1, 5, 4, 3, 5)
q3 <- c(5, 4, 4,NA, 2, 5, 4, 5)
q4 <- c(1, 1, 3, 3, 4, 5, 4, 5)

# Selecting Elements of Vectors
q1[5]
q1[ c(5, 6, 7, 8) ]
q1[5:8]
q1[gender == "m"]
mean( q1[ gender == "m" ], na.rm = TRUE)
```

```
# ---Factors---

# Numeric Factors

# First, as a vector
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)
workshop
table(workshop)
mean(workshop)
gender[workshop == 2]

# Now as a factor
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)
workshop <- factor(workshop)
workshop
table(workshop)
mean(workshop) #generates error now.
gender[workshop == 2]
gender[workshop == "2"]

# Recreate workshop, making it a factor
# including levels that don t yet exist.
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)
workshop <- factor(
  workshop,
  levels = c( 1,   2,     3,      4),
  labels = c("R", "SAS", "SPSS", "Stata")
)

# Recreate it with just the levels it
# curently has.
workshop <- c(1, 2, 1, 2, 1, 2, 1, 2)
workshop <- factor(
  workshop,
  levels = c( 1,  2),
  labels = c("R","SAS")
)

workshop
table(workshop)
gender[workshop == 2]
gender[workshop == "2"]
gender[workshop == "SAS"]

# Character factors
```

```
gender <- c("f", "f", "f", NA, "m", "m", "m", "m")
gender <- factor(
  gender,
  levels = c("m",    "f"),
  labels = c("Male", "Female")
)

gender
table(gender)
workshop[gender == "m"]
workshop[gender == "Male"]

# Recreate gender and make it a factor,
# keeping simpler m and f as labels.
gender <- c("f", "f", "f", NA, "m", "m", "m", "m")
gender <- factor(gender)
gender

# Data Frames
mydata <- data.frame(workshop, gender, q1, q2, q3, q4)
mydata

names(mydata)
row.names(mydata)

# Selecting components by index number
mydata[8, 6] #8th obs, 6th var
mydata[ , 6] #All obs, 6th var
mydata[ , 6][5:8] #6th var, obs 5:8

# Selecting components by name
mydata$q1
mydata$q1[5:8]


# Example renaming gender to sex while
# creating a data frame (left as a comment)
#
# mydata <- data.frame(workshop, sex = gender,
#   q1, q2, q3, q4)

# Matrices

# Creating from vectors
```

```
mymatrix <- cbind(q1, q2, q3, q4)
mymatrix
dim(mymatrix)

# Creating from matrix function
# left as a comment so we keep
# version with names q1, q2...
#
# mymatrix <- matrix(
#   c(1, 1, 5, 1,
#     2, 1, 4, 1,
#     2, 2, 4, 3,
#     3, 1, NA,3,
#     4, 5, 2, 4,
#     5, 4, 5, 5,
#     5, 3, 4, 4,
#     4, 5, 5, 5),
#  nrow = 8, ncol = 4, byrow = TRUE)
# mymatrix

table(mymatrix)
mean(mymatrix, na.rm = TRUE)
cor(mymatrix, use = "pairwise")

# Selecting Subsets of Matrices

mymatrix[8, 4]
mymatrix[5:8, 3:4]
mymatrix[ ,4][1:4]
mymatrix$q4  # No good!
mymatrix[ ,"q4"]

# Matrix Algebra

mymatrixT <- t(mymatrix)
mymatrixT

# Lists
mylist <- list(workshop, gender,
  q1, q2, q3, q4, mymatrix)
mylist

# List, this time adding names
mylist <- list(
  workshop = workshop,
  gender   = gender,
  q1 = q1,
  q2 = q2,
  q3 = q3,
  q4 = q4,
  mymatrix = mymatrix)
mylist

# Selecting components by index number.
mylist[[2]]
mylist[[2]][5:8]

mylist[2]
mylist[2][5:8]  # Bad!

# Selecting components by name.
mylist$q1
mylist$mymatrix[5:8, 3:4]


# ---Saving Your Work---

ls()
objects() #same as ls()

save.image("myall.RData")
save(mydata, file = "mydata.RData")


# The 2nd approach is commented to keep
# the q variables for following examples.
# rm(x, y, workshop, gender, q1, q2, q3, q4, mylist)
# ls()
# save.image(file = "mydata.RData")

# ---Comments to Document Your Programs---

# This comment is on its own line, between functions.

workshop <- c(1, 2, 1, 2, #This comment is within the arguments.
              1, 2, 1, 2) #And this is at the end.

# ---Comments to Document Your Objects---

comment(mydata) <- "Example data from R for SAS and SPSS Users"
comment(mydata)

# ---Controlling Functions---

# Controlling Functions with Arguments

help("mean")
mean(x = q3, trim = .25, na.rm = TRUE)
mean(na.rm = TRUE, x = q3, trim = .25)
mean(q3, .25, TRUE)
mean(q3, t = .25, na.rm = TRUE)
mean(1, 2, 3)
mean( c(1, 2, 3) )
mean( 1:3 )

# Controlling Functions With Formulas

lm( q4 ~ q1 + q2 + q3, data = mydata )

t.test(q1 ~ gender, data = mydata)

t.test( q1[ which(gender == "Female") ],
        q1[ which(gender == "Male")   ],
        data = mydata)  # Data ignored!

# Controlling Functions with Extractor Functions


lm( q4 ~ q1 + q2 + q3, data = mydata )

myModel <- lm( q4 ~ q1 + q2 + q3, data = mydata )
class(myModel)
summary(myModel)

# How Much Output Is There?

print(mymodel)

mode(myModel)
class(myModel)
names(myModel)
print( unclass(myModel) )

myModel$coefficients
class( myModel$coefficients )
```

```
barplot( myModel$coefficients )

# ---Writing Your Own Functions (Macros)---

myvar <- c(1, 2, 3, 4, 5)

# A bad function.
mystats <- function(x) {
  mean(x, na.rm = TRUE)
    sd(x, na.rm = TRUE)
}

mystats(myvar)

# A good function that just prints.
mystats <- function(x) {
  print( mean(x, na.rm = TRUE) )
  print(   sd(x, na.rm = TRUE) )
}
mystats(myvar)

# A function with vector output.
mystats  <- function(x) {
  mymean <- mean(x, na.rm = TRUE)
  mysd   <-   sd(x, na.rm = TRUE)
  c(mean = mymean, sd = mysd )
}
mystats(myvar)
myVector <- mystats(myvar)
myVector

# A function with list output.
mystats   <- function(x) {
  myinput <- x
  mymean  <- mean(x, na.rm = TRUE)
  mysd    <-   sd(x, na.rm = TRUE)
  list(data = myinput, mean = mymean, sd = mysd)
}
mystats(myvar)
myStatlist <- mystats(myvar)
myStatlist
mystats

save(mydata, mymatrix, mylist, mystats,
  file = "myWorkspace.RData")
```

6

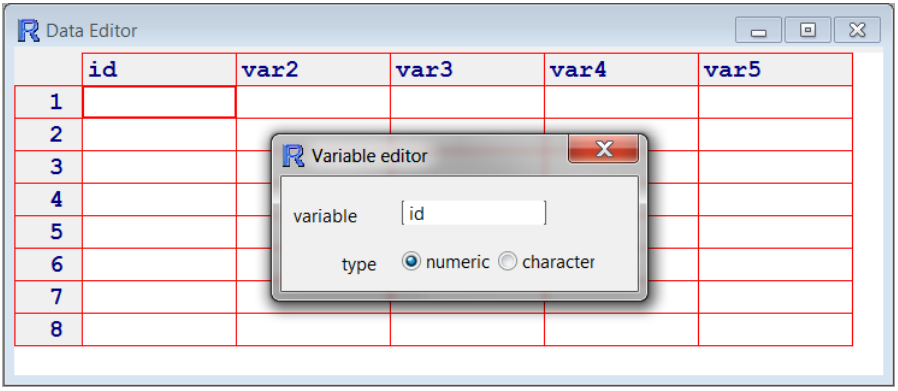
Data Acquisition

You can enter data directly into R, and you can read data from a wide range of sources. In this chapter I will demonstrate R's data editor as well as reading and writing data in text, Excel, SAS, SPSS and ODBC formats. For other topics, especially regarding relational databases, see the R Data Import/Export manual [46]. If you are reading data that contain dates or times, see Sect. 10.21.

6.1 Manual Data Entry Using the R Data Editor

R has a simple spreadsheet-style data editor. Unlike SAS and SPSS, you cannot use it to create a new data frame. You can only edit an existing one. However, it is easy to create an empty data frame, which you can then fill in using the editor. Simply submit the following command:

```
mydata <- edit( data.frame() )
```



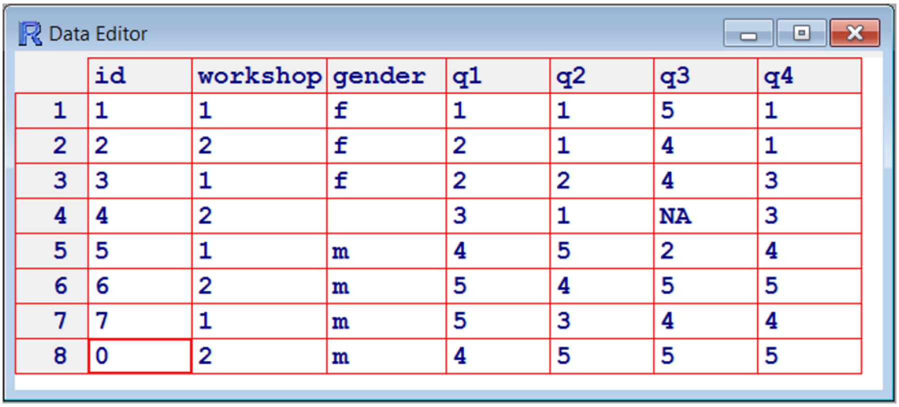
	id	var2	var3	var4	var5
1					
2					
3					
4					
5					
6					
7					
8					

variable id

type ● numeric ○ character

Fig. 6.1. Adding a new variable in the R data editor

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_6, © Springer Science+Business Media, LLC 2011



	id	workshop	gender	q1	q2	q3	q4
1	1	1	f	1	1	5	1
2	2	2	f	2	1	4	1
3	3	1	f	2	2	4	3
4	4	2		3	1	NA	3
5	5	1	m	4	5	2	4
6	6	2	m	5	4	5	5
7	7	1	m	5	3	4	4
8	0	2	m	4	5	5	5

Fig. 6.2. The R data editor with practice data entered

The window in Fig. 6.1 will appear.[1] Initially the variables are named *var1*, *var2*, and so on. You can easily change these names by clicking on them. I clicked on the variable name *var1*, which brought up the *Variable editor* window shown in the center of Fig. 6.1. I then changed it to “id” and left the “numeric” button selected so that it would remain a numeric variable. I then closed the variable editor window by clicking the usual X in the upper right corner.

Follow the steps above until you have created the data frame shown in Fig. 6.2. Make sure to click “character” when defining a character variable. When you come to the NA values for observation 4, leave them blank. You could enter the two-character string “NA” for numeric variables, but R will not recognize that as a missing value for character variables here. Exit the editor and save changes by choosing *File> Close* or by clicking the Windows X button. There is no *File> Save* option, which feels quite scary the first time you use it, but R does indeed save the data.

Notice that the variable in our ID variable matches the row names on the leftmost edge of Fig. 6.2. R went ahead and created row names of “1,” “2,” etc. so why did I bother to enter them into the variable id? Because while the data editor allows us to easily change *variable* names, it does not allow us to change *row* names. If you are happy with its default names, you do not need to create your own id variable. However, if you wanted to enter your own row names using the data editor, you can enter them instead into a variable like id and then later set that variable to be row names with the following command:

```
row.names(mydata) <- mydata$id
```

[1] These steps are for the Windows version. The Macintosh version is different but easy to figure out. The Linux version does not include even this basic GUI.

This command selects id from mydata using the form dataframe$variable, which we will discuss further in Sect. 7.7, "Selecting Variables Using $ Notation."

Before using these data, you would also want to use the `factor` function to make workshop and gender into factors.

```
mydata$workshop <- factor(mydata$workshop)
mydata$gender   <- factor(mydata$gender)
```

To see how to do this with value labels, see our discussion in Sect. 11.1, "Value Labels or Formats (and Measurement Level)."

We now have a data frame that we can analyze, save as a permanent R data file, or write out in text, Excel, SAS, or SPSS format.

When we were initially creating the empty data frame, we could have entered the variable names with the following function call:

```
mydata <- data.frame(id = 0., workshop = 0.,
  gender = " ", q1 = 0., q2 = 0., q3 = 0., q4 = 0.)
```

Since this approach allows you to name all the variables, it is a major time saver when you have to create more than one copy of the data or if you create a similar data set in the future.

R has a `fix` function that actually calls the more aptly named `edit` function and then writes the data back to your original data frame. So

```
fix(mydata)
```

does the same thing as

```
mydata <- edit(mydata)
```

I recommend not using the `edit` function on existing data frames as I find it all too easy to begin editing with just:

```
edit(mydata)  # Do NOT do this!
```

It will look identical on the screen, but this does not tell `edit` where to save your work. When you exit, your work will appear to be lost. However, R stores the last value you gave it in an object named `.Last.value`, so you can retrieve the data with this command.

```
mydata <- .Last.value
```

We will use the `edit` function later when renaming variables.

6.2 Reading Delimited Text Files

Delimited text files use special characters, such as commas, spaces, or tabs to separate each data value. R can read a wide range of such files. In this section I will show you how to read the most popular types.

6.2.1 Reading Comma-Delimited Text Files

Let us begin by reading a *c*omma-*s*eparated *v*alue (CSV) file like:

```
workshop,gender,q1,q2,q3,q4
1,1,f,1,1,5,1
2,2,f,2,1,4,1
3,1,f,2,2,4,3
4,2, ,3,1, ,3
5,1,m,4,5,2,4
6,2,m,5,4,5,5
7,1,m,5,3,4,4
8,2,m,4,5,5,5
```

There are several important things to notice about these data.

1. The top row contains variable names. This is called the file's *header* line.
2. ID numbers are in the leftmost column, but the header line does not contain a name like "ID" for it.
3. Values are separated by commas.
4. Spaces (blanks) represent missing values.
5. There are no blanks before or after the character values of "m" and "f."
6. Each line in the file ends with a single stroke of the *Enter* key, not with a final tab. Your operating system stores either a line feed character or a carriage return and a line feed. R will treat them the same.

You can read this file using the `read.csv` function call below. If you have already set your working directory in your current R session, you do not need to set it again.

```
> setwd("c:/myRfolder")

> mydata <- read.csv("mydata.csv")

> mydata

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2         3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

Notice that it read the id variable and stored it automatically in the row names position on the left side of the data frame. It did that because R found seven columns of data but only six names. Whenever R finds one fewer names than columns, it assumes the first column must be an id variable.

If your CSV file does not have a header line *and* all your data values are numeric, R will automatically name the variables "V1," "V2," and so on, similar to SAS's and SPSS's "VAR1," "VAR2," etc. default names. [2] However, our file has both numeric data and character data. R will have trouble figuring out if the first line in the file contains names or data. You can tell it to not try and interpret the first line as variable names by adding the argument `header = FALSE` to the `read.csv` function call.

Let us see what happens when the header line *does* contain a name for the first column, like the following, which is the beginning of the file named mydataID.csv:

```
id,workshop,gender,q1,q2,q3,q4
1,1,f,1,1,5,1
2,2,f,2,1,4,1
...
```

If we read the file exactly as before, we would have an additional variable named "id." R would also create row names of "1," "2," etc., but our ID variable might have contained more useful information. Not getting your identifying variable into the row names attribute does not cause any major problems, but R will automatically identify observations by their row names, so if you have an ID variable, it makes sense to get it into the row names attribute.

To tell R which variable contains the row names, you simply add the `row.names` argument.

```
> mydata <- read.csv("mydataID.csv",
+   row.names = "id")

> mydata

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
...
```

When we let R figure out that there was an ID variable, it *had to be the first column.* That is usually where ID variables reside, but if you ever have one in another location, then you will have to use the `row.names` argument to store it in the row names attribute.

[2] Note the inconsistency with R's own data editor, which uses the default names, "var1," "var2," etc.

6.2.2 Reading Tab-Delimited Text Files

Reading tab-delimited files in R is done very similarly to reading comma-delimited files, but using the `read.delim` function.

The following is the tab-delimited text file we will read:

```
workshop   gender   q1   q2   q3   q4
1    1      f      1  1  5  1
2    2      f      2  1  4  1
3    1      f      2  2  4  3
4    2             3  1     3
5    1      m      4  5  2  4
6    2      m      5  4  5  5
7    1      m      5  3  4  4
8    2      m      4  5  5  5
```

There are several important things to notice about these data.

1. The top row contains variable names. This is called the file's *header* line. Note that your header line should *never* begin nor end with tab characters! That would cause R to think you have more variables than you actually do. If you leave the names out, you should specify the argument `header = FALSE`. In that case R will name the variables "V1," "V2," etc.
2. ID numbers are in the leftmost column, but the header line does not contain a name like "ID" for it.
3. Values are separated by single tab characters.
4. Two consecutive tab characters represent missing values, although a blank space would work, too.
5. There are no blanks before or after the character values "m" and "f."
6. Each line of data ends with a single stroke of the *Enter* key, *not* with a final tab. Your operating system ends lines with either a line feed character, or a carriage return and a line feed. R will treat them the same.

We can use the `read.delim` function to read this file:

```
> setwd("c:/myRfolder")

> mydata <- read.delim("mydata.tab")

> mydata

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2         3  1 NA  3
```

```
5         1       m  4  5  2  4
6         2       m  5  4  5  5
7         1       m  5  3  4  4
8         2       m  4  5  5  5
```

We see that two consecutive tabs for variable q3 was correctly identified as a missing value (NA).

To read a file whose header *does* name the variable in the first column, add the row.names argument:

```
> mydata <- read.delim("mydataID.tab",
+   row.names = "id")

> mydata

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
...
```

If you need to read a file that has multiple tabs or spaces between values, use the read.table function. The read.csv and read.delim functions both call read.table with some arguments preset to helpful values. It is a good idea to read the help files of any of these functions to see how they relate to each other and to see the many additional parameters you can control by using read.table.

The read.table function actually does its work by calling the powerful and complex scan function.

6.2.3 Reading Text from a Web Site

You can use any of the methods described above to read data stored on a Web site. To do so, simply enter the site's URL in place of the filename. I have uploaded the file mydata.csv to this book's Web site, so you can try it. While you may access the book's Web site at the URL http://r4stats.com, that URL is actually an easy-to-remember pointer to the real site used in the code below:[3]

```
myURL <- "http://sites.google.com/site/r4statistics/mydata.csv"
mydata <- read.csv(myURL)
mydata
```

[3] As of this writing, I am planning a change in Web servers. If the code does not work try http://r4stats.com/mydata.csv or check the book's Corrections & Clarifications file on the download site.

6.2.4 Reading Text from the Clipboard

In virtually all software that has a spreadsheet data editor, you can copy data from another program and paste it in. But not in R! However, the Windows version of R does have a way to read from the clipboard using functions. You do begin the process in your other software as usual, by selecting the data you wish to copy and then pressing CTRL-c.

If it is a single column of data, you can then read it with

```
myvector <- readClipboard()
```

If it is a single row of data, you can paste it into a text editor, press Enter at the end of each line, and copy it again from there. Otherwise, a row of values will end up as single-character vector.

If you have copied a whole set of columns (hopefully with column names), you can read it with:

```
mydata <- read.delim("clipboard", header = TRUE)
```

If you have problems using this approach, try pasting the data into a text editor so you can check to see if there are extraneous spaces or tabs in the data. All the rules that apply to reading data from a file apply here as well. For example, if you copied comma-separated values to the clipboard, `read.csv` would be the function to use. I frequently see data discussed on R-help or Web pages that are separated by multiple spaces, so I use `read.table` to read them from the clipboard.

Unfortunately, this approach copies only the number of decimal places that you had displayed at the time. Therefore, almost any other method is better for reading numbers with many digits after the decimal point.

6.2.5 Missing Values for Character Variables

In the previous two subsections, we ignored a potential problem. The missing value for variable q3 was always displayed as NA, Not Available. However, the missing value for gender was displayed as a blank.

If we had entered R's standard missing value, "NA," where we had missing values, then even the character data would have shown up as missing. However, few other programs write out NA as missing.

Just as in SAS or SPSS, you can read blanks as character values, and R will not set them to missing unless you specifically tell it to do so. Often, it is not very important to set those values to missing. A person's mailing address is a good example. You would never use it in an analysis, so there is little need to set it to missing.

However, when you need to use a character variable in an analysis, setting it to missing is, of course, very important. Later in the book we *will* use gender in analyses, so we must make sure that blank values are set to missing.

In our comma-delimited file, the missing value for gender was entered as a single space. Therefore, the argument `na.char = " "` added to any of the comma-delimited examples will set the value to missing. Note there is a single space between the quotes in that argument.

In our tab-delimited file, the missing value for gender was entered as nothing between two tabs (i.e., just two consecutive tabs). Therefore, the argument `na.char = ""` added to any of the tab-delimited examples will set the value to missing. Note that there is now *no* space between the quotes in that argument.

However, in both comma- and tab-delimited files, it is very easy to accidentally have blanks where you think there are none or to enter more than you meant to. Then your `na.char` setting will be wrong for some cases.

It is best to use a solution that will get rid of all trailing blanks. That is what the argument `strip.white = TRUE` does. When you use that argument, `na.char = ""` will work regardless of how many blanks may have been there before.

Let us try it with our comma-delimited file, since it contains a blank we can get rid of:

```
> mydata <- read.csv("mydataID.csv",
+   row.names   = "id",
+   strip.white = TRUE,
+   na.strings  = "" )

> mydata

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2   <NA>  3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

The only difference between this output and the last one we read for mydataID.csv is that gender is shown as `<NA>` now instead of blank. R adds angle brackets, "<>", around the value so you can tell NA stands for Not Available (missing) rather than something meaningful, such as North America. The NA value in the q3 variable is not in angle brackets because it cannot possibly be a valid numeric value.

The `strip.white = TRUE` argument also provides the benefit of getting rid of trailing blanks that would set some genders equal to `"m"` and others to `"m "` or `"m    "`. We do not want trailing blanks to accidentally split the males into different groups!

Finally, getting rid of trailing blanks saves space. Since R stores its data in your computer's main memory, saving space is very important.

6.2.6 Trouble with Tabs

In many text editors, including R's, tabs are invisible. That makes it easy to enter an additional tab or two, throwing R off track. A helpful tool to count the number of fields is the `count.fields` function. This function looks by default for items separated by any amount of spaces or tabs. Therefore, you must specify `sep = "\t"` because that is what R uses to represent the tab character:

```
> count.fields("mydata.tab", sep = "\t")

[1] 6 7 7 7 7 7 7 7 7
```

We see that all the lines in the file contain seven items except for the header row. The header row has 6. That is the clue that R needs to determine that the first column of the data contains an ID variable that it will then store as row names.

If R complains of too many names in the header line, or not enough values on data lines, or if it creates more variables than you expected, often you have an inconsistent number of tabs in your file.

Check the header line that contains your variable names and the first few lines of data for extra tabs, especially at the beginning or end of a line. If you have an ID variable in the first column and it is not named in your header line, it is very tempting to put a tab before the first variable name. That will get it to line up over the first column, but it will also tell R that your first variable name is missing!

If you have a data file that has some short values and some very long values in the same column, the person who entered it may have put two tabs after the short values to get the following column to line up again. In that case, you can read it with the `read.table` function. That function has greater flexibility for reading delimited files.

When a file has varying numbers of tabs between values, `read.table` can read it because its default delimiter is *any number of tabs or spaces*! However, this also means that you cannot represent missing values by entering two consecutive tabs, or even by putting a space between two tabs. With our practice tabbed data set, `read.table` would generate the error message "*line 4 did not have 7 elements*." In that case, you must enter some code to represent "missing." The value "NA" is the one that R understands automatically, for both numeric and character values. If you use any other codes, such as "." or "999," specify the character as missing by using the `na.char` argument. See also Sect. 10.5 to learn a wider range of approaches to handling missing values.

With `read.table`, if you specify the argument, `delim = "\t"`, then it uses one single tab as a delimiter. That is one thing `read.delim` does for you automatically.

6.2.7 Skipping Variables in Delimited Text Files

R must hold all its data in your computer's main memory. This makes skipping columns while reading data particularly important. The following is the R function call to read data while skipping the fourth and fifth columns. If you have already set your working directory in your current R session, you do not need to set it again.

```
> setwd("c:/myRfolder")

> myCols <- read.delim("mydata.tab",
+   strip.white = TRUE,
+   na.strings  = "",
+   colClasses  = c("integer", "integer", "character",
+   "NULL", "NULL", "integer", "integer") )

> myCols
  workshop gender q3 q4
1        1      f  5  1
2        2      f  4  1
3        1      f  4  3
4        2   <NA> NA  3
5        1      m  2  4
6        2      m  5  5
7        1      m  4  4
8        2      m  5  5
>
> # Clean up and save workspace.
> rm(myCols)
```

We used the name myCols to avoid overwriting mydata. You use the `colClasses` argument to specify the *class* of each column. The classes include logical (TRUE/FALSE), integer (whole numbers), numeric (can include decimals), character (alphanumeric string values), and factor (categorical values like gender). See the help file for other classes like dates. The class we need for this example is NULL. We use it to drop variables.

However, `colClasses` requires you to specify the classes of all columns, including any initial ID or row names variable. The classes must be included within quotes since they are character strings.

6.2.8 Reading Character Strings

Many of R's functions for reading text are related. For example, read.csv and read.delim both call the read.table function with some arguments set to useful defaults. The read.table function, in turn, calls the scan function, again with reasonable arguments set to save you the work of understanding all of scan's flexibility. So if you have problems reading a text file, check the help file of the function you are using first. If you do not find a solution, read the help file of the function it calls, often read.table. Save the help file for the complex scan function for last.

For example, when you read string variables, R will usually convert them to factors. However, you do not always want that to happen. Mailing addresses are a common type of data that will never be used as a factor. To prevent the conversion of *all* strings to factors, you can set stringsAsFactors = FALSE. You could instead use as.is = "x" or as.is = c("x","y") to prevent just the variables x or x and y from becoming factors.

6.2.9 Example Programs for Reading Delimited Text Files

SAS Program for Reading Delimited Text Files

The parts of this program to read CSV and tab-delimited files was written by SAS itself using *File> Import Data*. I only had to write the last one, which reads data from a Web site.

```
* Filename: ReadDelimited.sas ;

LIBNAME myLib  C:\myRfolder ;

* ---Comma Delimited Files---;
PROC IMPORT OUT=myLib.mydata
            DATAFILE="C:\myRfolder\mydataID.csv"
            DBMS=CSV REPLACE;
     GETNAMES=YES;
     DATAROW=2;
RUN;
PROC PRINT; RUN;

* ---Tab Delimited Files---;
PROC IMPORT OUT= myLib.mydata
            DATAFILE= "C:\myRworkshop\mydataID.tab"
            DBMS=TAB REPLACE;
     GETNAMES=YES;
     DATAROW=2;
RUN;
PROC PRINT; RUN;
```

```
* ---Reading from a Web Site---;
FILENAME myURL URL
  "http://sites.google.com/site/r4statistics/mydataID.csv";
PROC IMPORT DATAFILE= myURL
            DBMS=CSV REPLACE
            OUT= myLib.mydata;
     GETNAMES=YES;
     DATAROW=2;
RUN;
PROC PRINT; RUN;
```

SPSS Program for Reading Delimited Text Files

Notice that SPSS does not actually use the variable names that are embedded within the data file. We must skip these and begin reading the data on the second line. The VARIABLES keyword provides the names. This program was written by SPSS itself using *File> Open> Data*. SPSS cannot read text files from a URL directly, so that part of our steps is not replicated in this program. (It can read SPSS, SAS, Excel, and Stata files from a URL through the SPSSINC GETURI DATA extension.)

```
* Filename: ReadDelimited.SPS

CD  C:\myRfolder .

* ---Comma Delimited Files---.
GET DATA  /TYPE=TXT
 /FILE= mydataID.csv
 /DELCASE=LINE
 /DELIMITERS=","
 /ARRANGEMENT=DELIMITED
 /FIRSTCASE=2
 /IMPORTCASE=ALL
 /VARIABLES=id F1.0  workshop F1.0 gender A1.0
 q1 F1.0  q2 F1.0  q3 F1.0  q4 F1.0 .
LIST.
SAVE OUTFILE= C:\myRfolder\mydata.sav .

* ---Tab Delimited Files---.
GET DATA
  /TYPE=TXT
  /FILE="C:\myRfolder\mydataID.tab"
  /DELCASE=LINE
  /DELIMITERS="\t"
  /ARRANGEMENT=DELIMITED
  /FIRSTCASE=2
  /IMPORTCASE=ALL
 /VARIABLES = id F1.0  workshop F1.0 gender A1.0
 q1 F1.0  q2 F1.0  q3 F1.0  q4 F1.0 .
LIST.
EXECUTE.
DATASET NAME DataSet1 WINDOW=FRONT.
```

R Program for Reading Delimited Text Files

```
# Filename: ReadDelimited.R

setwd("c:/myRfolder")

#---Comma Delimited Files---

# Read comma delimited file.
# With id variable not named.

mydata <- read.csv("mydata.csv")
mydata

# This time with id named in the header

mydata <- read.csv("mydataID.csv",
  row.names = "id")
mydata

#---Tab Delimited Files---

# Read a tab delimited file with named ID column.
mydata <- read.delim("mydata.tab")
mydata

count.fields("mydata.tab", sep = "\t")

# Again with ID named in the header
mydata <- read.delim("mydataID.tab",
  row.names = "id")
mydata

# ---Reading Text from a Web Site---

myURL <- "http://sites.google.com/site/r4statistics/mydata.csv"
```

```
mydata <- read.csv(myURL)
mydata

# ---Reading Text from the Clipboard---

# Copy a column of numbers or words, then:
myvector <- readClipboard()
myvector

# Open mydata.csv, select & copy contents, then:
mydata <- read.delim("clipboard", header = TRUE)
mydata

#---Missing Values for Character Variables---

mydata <- read.csv("mydataID.csv",
  row.names   = "id",
  strip.white = TRUE,
  na.strings  = "" )
mydata

#---Skipping Variables in Delimited Text Files---

myCols <- read.delim("mydata.tab",
  strip.white = TRUE,
  na.strings  = "",
  colClasses  = c("integer", "integer", "character",
  "NULL", "NULL", "integer", "integer") )
myCols

# Clean up and save workspace.
rm(myCols)

save.image(file = "mydata.RData")
```

6.3 Reading Text Data Within a Program

It is often useful to have a small data set entered inside a program. SAS does this using the DATALINES or CARDS statements. SPSS uses the BEGIN DATA and END DATA commands to accomplish this task.

This approach is popular when teaching or for an example when you post a question on Internet discussion lists. You only have one file, and anyone can copy it and run it without changing it to locate a data file.

Although beginners are often drawn to this approach due to its simplicity, it is not a good idea to use this for more than a few dozen observations. To see the top and bottom of your program requires scrolling past all of the data, which is needlessly time consuming. As we will soon see, R also displays data in the console, scrolling potential error messages offscreen if there is more than a screen's worth of data.

We will discuss two ways to read data within an R program: one that is easy and one that is more generally applicable.

6.3.1 The Easy Approach

The easy approach is to nest the `stdin` function within any other R function that reads data. It tells R that the data are coming from the same place the program is, which is called the *st*andar*d* *in*put.

In our next example, we will use CSV format, so we will nest a call to the `stdin` function within a call to the `read.csv` function.

```
mydata <- read.csv( stdin() )
workshop,gender,q1,q2,q3,q4
1,1,f,1,1,5,1
2,2,f,2,1,4,1
3,1,f,2,2,4,3
4,2,NA,3,1,NA,3
5,1,m,4,5,2,4
6,2,m,5,4,5,5
7,1,m,5,3,4,4
8,2,m,4,5,5,5

# Blank line above ends input.
```

Note that I actually typed "NA" in for missing values, and I was careful to never add any spaces before or after the gender values of "m" or "f." That let us dispense with any additional arguments for the `read.csv` function. I could instead have used spaces as delimiters and used the `read.table` function in place of `read.csv`.

With this approach, it is important to avoid tabs as delimiters. They are not recognized by `stdin`, and your values would be read as if they were not delimited at all. You could avoid this by using R's "`\t`" character to represent tab characters, but that makes your data quite a mess!

Let us run our comma-delimited example and see what the output looks like.

```
> mydata <- read.csv( stdin() )
0: workshop,gender,q1,q2,q3,q4
1: 1,1,f,1,1,5,1
```

```
2: 2,2,f,2,1,4,1
3: 3,1,f,2,2,4,3
4: 4,2,NA,3,1,NA,3
5: 5,1,m,4,5,2,4
6: 6,2,m,5,4,5,5
7: 7,1,m,5,3,4,4
8: 8,2,m,4,5,5,5
9:
> # Blank line above ends input.
> mydata
  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2   <NA>  3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

I often add blank lines between sections of output to make it easier to read, but given that a blank line is actually used to end the data, I did not do so with this output.

You can see that R displays the data itself, and it prefixes each line with “0:”, “1:”, “2:”, etc. With all of the data displayed, this is obviously not something you would want to do with hundreds of observations! When we read data from files, we saw that R did not display them in the console.

The ninth line shows that it is blank and the numeric prefixing stops as R returns to its usual “>” prompt. It is the blank line that tells R that there are no more data. If you forget this, R will read your next program lines as data, continuing until it finds a blank line!

Printing the data by entering `mydata` shows us that the row names were correctly assigned and the two missing values are also correct.

6.3.2 The More General Approach

The previous subsection showed how to read data in the middle of an R program, and it required only a minor change. It had one important limitation however: you cannot use `stdin` to read data in programs that are sourced (included) from files.

Since putting data in the middle of a file is often done for interactive demonstrations, that is not often a serious limitation. However, there are times when you want to put the whole program, including data, in a separate file like “myprog.R” and bring it into R with the command

```
source("myprog.R")
```

To do this, we can place the whole data set into a character vector with a *single value* named "mystring":

```
mystring <-
"workshop,gender,q1,q2,q3,q4
1,1,f,1,1,5,1
2,2,f,2,1,4,1
3,1,f,2,2,4,3
4,2,NA,3,1,NA,3
5,1,m,4,5,2,4
6,2,m,5,4,5,5
7,1,m,5,3,4,4
8,2,m,4,5,5,5"

mydata <- read.csv( textConnection(mystring) )

> mydata
  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2   <NA>  3  1 NA  3...
```

Note that the `c` function is *not* used to combine all of those values into a vector. At the moment, the whole data set is one single character value! The `textConnection` function converts mystring into the equivalent of a file, which R then processes as it would a file.

This approach still has problems with tab-delimited data since strings do not hold tab characters unless you enter them using R's "`\t`" character. Therefore, it is best to use commas or spaces as delimiters.

6.3.3 Example Programs for Reading Text Data Within a Program

SAS Program for Reading Text Data Within a Program

```
* Filename: ReadWithin.sas ;

LIBNAME myLib  C:\myRfolder ;
DATA myLib.mydata;
INFILE DATALINES DELIMITER =  ,
   MISSOVER DSD firstobs=2 ;
INPUT id workshop gender $ q1 q2 q3 q4;
DATALINES;
id,workshop,gender,q1,q2,q3,q4
 1,1,f,1,1,5,1
 2,2,f,2,1,4,1
 3,1,f,2,2,4,3
 4,2, ,3,1, ,3
 5,1,m,4,5,2,4
 6,2,m,5,4,5,5
 7,1,m,5,3,4,4
 8,2,m,4,5,5,5
PROC PRINT; RUN;
```

SPSS Program for Reading Text Data Within a Program

```
* Filename: ReadWithin.sps .

DATA LIST / id 2 workshop 4 gender 6 (A)
 q1 8  q2 10  q3 12  q4 14.
BEGIN DATA.
 1,1,f,1,1,5,1
 2,2,f,2,1,4,1
 3,1,f,2,2,4,3
 4,2, ,3,1, ,3
 5,1,m,4,5,2,4
 6,2,m,5,4,5,5
 7,1,m,5,3,4,4
 8,2,m,4,5,5,5
END DATA.
LIST.
SAVE OUTFILE= C:\myRfolder\mydata.sav .
```

R Program for Reading Text Data Within a Program

```
# Filename: ReadWithin.R

# The stdin approach.
mydata <- read.csv( stdin() )
workshop,gender,q1,q2,q3,q4
1,1,f,1,1,5,1
2,2,f,2,1,4,1
3,1,f,2,2,4,3
4,2,NA,3,1,NA,3
5,1,m,4,5,2,4
6,2,m,5,4,5,5
7,1,m,5,3,4,4
8,2,m,4,5,5,5

# Blank line above ends input.
mydata
```

```
# The textConnection approach
# that works when sourcing files.

mystring <-
"workshop,gender,q1,q2,q3,q4
1,1,f,1,1,5,1
2,2,f,2,1,4,1
3,1,f,2,2,4,3
4,2,NA,3,1,NA,3
5,1,m,4,5,2,4
6,2,m,5,4,5,5
7,1,m,5,3,4,4
8,2,m,4,5,5,5"
mydata <- read.csv( textConnection(mystring) )
mydata

# Set working directory & save workspace.
setwd("c:/myRfolder")

save.image(file = "mydata.RData")
```

6.4 Reading Multiple Observations per Line

With small data sets it can be convenient to enter the data with more than one observation per line. This is most often done with examples for teaching or for demonstrating problems when asking for help on the R-help e-mail list. I will be extending the technique covered in the previous section, so if you did not read it just now, please go back and review it.

To read multiple observations per line in SAS, you would use the trailing `@@` symbol to read it as in

```
INPUT ID GENDER $ Q1-Q4 @@;
```

SPSS would simply use the FREE format.

In the following example, I am reading our practice data set with two observations per line. However, the example works with any number of observations per line.

```
mylist <- scan( stdin(),
  what = list(id = 0, workshop = 0, gender = "",
            q1 = 0, q2 = 0, q3 = 0, q4 = 0) )
 1 1 f 1 1 5 1    2 2 f 2 1 4 1
 3 1 f 2 2 4 3    4 2 NA 3 1 NA 3
 5 1 m 4 5 2 4   6 2 m 5 4 5 5
 7 1 m 5 3 4 4   8 2 m 4 5 5 5

# Blank line above ends input.
```

I am reading the data into mylist by calling the `scan` function, with two arguments:

1. The first argument is the "file" to scan. If the data were in a file, we would list its name here in quotes. We are using the standard input (i.e., the same source as the programming statements themselves), so I put the `stdin()` call there. The `textConnection` approach would work here as well. The example program at the end of this section includes that approach.
2. What to scan. This is the type of data to scan such as numeric or character. Since we have various types to scan, I am giving it a list of variables that are all initialized with zero for numeric variables and an empty character string, `""`, for character variables.

Let us see what it has read:

```
> mylist

$id
[1] 1 2 3 4 5 6 7 8

$workshop
[1] 1 2 1 2 1 2 1 2

$gender
[1] "f" "f" "f" NA  "m" "m" "m" "m"

$q1
[1] 1 2 2 3 4 5 5 4...
```

We see that it read the data just fine, but it is in a list. We can convert that to a data frame using

```
> mydata <- data.frame(mylist)

> mydata

  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
3  3        1      f  2  2  4  3
...
```

I did the above example in two steps to make it easy to understand. However, it would be more efficient to do both steps at once:

```
mydata <- data.frame(
  scan( stdin(),
  what = list(id = 0, workshop = 0, gender = "",
              q1 = 0, q2 = 0, q3 = 0, q4 = 0) )
 1 1 f 1 1 5 1    2 2 f 2 1 4 1
 3 1 f 2 2 4 3    4 2 NA 3 1 NA 3
 5 1 m 4 5 2 4    6 2 m 5 4 5 5
 7 1 m 5 3 4 4    8 2 m 4 5 5 5

# Blank line above ends input.
```

6.4.1 Example Programs for Reading Multiple Observations per Line

Example SAS Program for Reading Multiple Observations per Line

```
* Filename: ReadMultipleObs.sas ;

DATA mydata;
INPUT id workshop gender $ q1-q4 @@;
DATALINES;
1 1 f 1 1 5 1    2 2 f 2 1 4 1
3 1 f 2 2 4 3    4 2 . 3 1 . 3
5 1 m 4 5 2 4    6 2 m 5 4 5 5
7 1 m 5 3 4 4    8 2 m 4 5 5 5
;
PROC PRINT; RUN;
```

Example SPSS Program for Reading Multiple Observations per Line

SPSS must use the FREE format to read multiple observations per line. With that format, it cannot read missing values without a nonblank delimiter. Therefore, I use commas in the example below, so that two consecutive commas will tell SPSS that the value is missing.

```
* Filename: ReadMultipleObs.SPS.

DATA LIST FREE/ id (f1.0)  workshop (f1.0) gender (A)
q1 (f1.0)  q2 (f1.0)  q3 (f1.0) q4 (f1.0).
```

```
BEGIN DATA.
1,1,f,1,1,5,1,    2,2,f,2,1,4,1
3,1,f,2,2,4,3,    4,2, ,3,1, ,3
5,1,m,4,5,2,4,    6,2,m,5,4,5,5
7,1,m,5,3,4,4,    8,2,m,4,5,5,5
END DATA.

LIST.
```

Example R Program for Reading Multiple Observations per Line

```
# Filename: ReadMultipleObs.R

mylist <- scan( stdin(),
  what = list(id = 0, workshop = 0, gender = "",
              q1 = 0, q2 = 0, q3 = 0, q4 = 0))
1 1 f 1 1 5 1    2 2 f 2 1 4 1
3 1 f 2 2 4 3    4 2 NA 3 1 NA 3
5 1 m 4 5 2 4    6 2 m 5 4 5 5
7 1 m 5 3 4 4    8 2 m 4 5 5 5

# Blank line above ends input.
mylist

mydata <- data.frame(mylist)
head(mydata)

# The textConnection approach

mystring <-
"1 1 f 1 1 5 1    2 2 f 2 1 4 1
3 1 f 2 2 4 3    4 2 NA 3 1 NA 3
5 1 m 4 5 2 4    6 2 m 5 4 5 5
7 1 m 5 3 4 4    8 2 m 4 5 5 5
"
mystring
mylist <- scan( textConnection(mystring),
  what = list(id = 0, workshop = 0, gender = "",
              q1 = 0, q2 = 0, q3 = 0, q4 = 0) )

mydata <- data.frame(mylist)

head(mydata)
```

6.5 Reading Data from the Keyboard

If you want to enter data from the keyboard line by line using SAS or SPSS, you would do so as we did in the previous two sections. They do not have a special data entry mode outside of their data editors. You can put R's `scan` function into a special data entry mode by not providing it with any arguments. It then prompts you for data one line at a time, but once you hit the Enter key, you cannot go back and change it in that mode.

Although you can do this on any operating system, its main use may be on Linux or UNIX computers, which lack the R GUI. Since this approach requires only the console command prompt, you can use it even without the R GUI. The following is an example.

```
> id <- scan()
1: 1 2 3 4 5 6 7 8
9:
Read 8 items
```

R prompts with "1:" indicating that you can type the first observation. When I entered the first line (just the digits 1 through 8), it prompted with "9:" indicating that I had already entered 8 values. When I entered a blank line, `scan` stopped reading and saved the vector named id.

To enter character data, we have to add the `what` argument. Since spaces separate the values, to enter a value that includes a space, you would enclose it in quotes like "R.A. Fisher."

```
> gender <- scan(what = "character")
1: f f f f m m m m
9:
Read 8 items
```

When finished with this approach, we could use the `data.frame` function to combine the vectors into a data frame:

```
mydata <- data.frame(id, workshop, gender, q1, q2, q3, q4)
```

6.6 Reading Fixed-Width Text Files, One Record per Case

Files that separate data values with delimiters such as spaces or commas are convenient for people to work with, but they make a file larger. So many text files dispense with such conveniences and instead keep variable values locked into the exact same column(s) of every record.

If you have a nondelimited text file with one record per case, you can read it using the following approach. R has nowhere near the flexibility in reading fixed-width text files that SAS and SPSS have. As you will soon see, making an error specifying the width of one variable will result in reading the wrong columns for all those that follow. While SAS and SPSS offer approaches that would do that, too, I do not recommend their use. In R, though, it is your only option which is fine so long as you carefully check your results.

Other languages such as Perl or Python are extremely good at reading text files and converting them to a form that R can easily read.

Below are the same data that we used in other examples but now it is in fixed-width format (Table 9.5).

```
011f1151
022f2141
031f2243
042 31 3
051m4524
062m5455
071m5344
082m4555
```

Important things to notice about this file.

1. No names appear on first line.
2. Nothing separates values.
3. The first value of each record is two columns wide; the remainder take only one column each. I made ID wider just to demonstrate how to read a variable that is more than one column wide.
4. Blanks represent missing values, but we could use any other character that would fit into the fixed number of columns allocated to each variable.
5. The last line of the file contains data. That is what SAS and SPSS expect, but R generates a warning that there is an "*incomplete final line found.*" It works fine though. If the warning in R bothers you, simply edit the file and press Enter once at the end of the last line.

The R function that reads fixed-width files is `read.fwf`. The following is an example of it reading the file above:

```
> setwd("c:/myRfolder")

> mydata <- read.fwf(
+    file        = "mydataFWF.txt",
+    width       = c(2, -1, 1, 1, 1, 1, 1),
+    col.names   = c("id", "gender", "q1", "q2", "q3", "q4"),
+    row.names   = "id",
+    na.strings  = "",
+    fill        = TRUE,
+    strip.white = TRUE)
```

```
Warning message:

In readLines(file, n = thisblock) :
  incomplete final line found on  mydataFWF.txt

> mydata

  gender q1 q2 q3 q4
1      f  1  1  5  1
2      f  2  1  4  1
3      f  2  2  4  3
4   <NA>  3  1 NA  3
5      m  3  5  2  4
6      m  5  4  5  5
7      m  5  3  4  4
8      m  4  5  5  5
```

The `read.fwf` function call above uses seven arguments:

1. The `file` argument lists the name of the file. It will read it from your current working directory. You can set the working directory with `setwd("path")` or you can specify a path as part of the file specification.
2. The `width` argument provides the width, or number of columns, required by each variable in order. The widths we supplied as a numeric vector are created using the `c` function. The first number, 2, tells R to read ID from columns 1 and 2. The next number, −1, tells R to skip one column. In our next example, we will not need to read the workshop variable, so I have put in a −1 to skip it now. The remaining pattern of 1, 1, 1, 1, tells R that each of the remaining four variables will require one column each. Be very careful at this step! If you made an error and told R that ID was one column wide, then `read.fwf` would read all of the other variables from the wrong columns.

   When you are reading many variables, specifying their length by listing them all like this is tedious. You can make this task much easier by using R's ability to generate vectors of repetitive patterns. For an example, see the Chap. 12.4, "Generating Values for Reading Fixed Width Files."
3. The `col.names` argument provides the column or variable names. Those, too, we provide in a character vector. We create it using the `c` function, `c("id","gender","q1","q2","q3","q4")`. Since the names are character (string) data, we must enclose them in quotes.

   Names can also be tedious to enter. R's ability to generate vectors of repetitive patterns, combined with the `paste` function, can generate long sets of variable names. For details, see Chap. 12, "Generating Data."
4. The `row.names` argument tells R that we have a variable that stores a name or identifier for each row. It also tells it which of the variable names from the `col.names` argument that is: "id."

5. The `na.strings =""` argument tells R that an empty field is a missing value. It already is for numeric data, but, as in SAS or SPSS, a blank is a valid character value. Note that there is no blank between the quotes! That is because we set the `strip.white` option to strip out extra blanks from the end of strings (below). As you see, R displays missing data for character data within angle brackets as `<NA>`.
6. The `fill` argument tells R to fill in blank spaces if the file contains lines that are not of full length (like the SAS MISSOVER option). Now is a good time to stop and enter `help("read.fwf")`. Note that there is no `fill` argument offered. It does, however, list its last argument as "...". This is called the *triple dot* argument. It means that it accepts additional unnamed arguments and will pass them on to another function that `read.fwf` might call. In this case, it is the `read.table` function. Clicking the link in the help file to that function will reveal the `fill` argument and what it does.
7. The `strip.white` argument tells R to remove any additional blanks it finds in character data values. Therefore, if we were reading a long text string like `"Bob      "`, it would delete the additional spaces and store just `"Bob"`. That saves space and makes logical comparisons easier. It is all too easy to count the number of blanks incorrectly when making a comparison like, `name == "Bob      "`.

The file was read just fine. The warning message about an "incomplete final line" is caused by an additional line feed character at the end of the last line of the file. Neither SAS nor SPSS would print a warning about such a condition.

The `read.fwf` function calls the `read.table` function to do its work, so you can use any of those arguments here as well.

6.6.1 Reading Data Using Macro Substitution

In Sect. 5.7.2 we first discussed how R can store the values of its arguments in vectors. That is essentially what SAS and SPSS call *macro substitution*. Let us now use that idea to simplify our program, making it easier to write and maintain.

Since file paths often get quite long, we will store the file name in a character vector named *myfile*. This approach also lets you put all of the file references you use at the top of your programs, so you can change them easily. We do this with the command:

```
myfile <- "mydataFWF.txt"
```

Next, we will store our variable names in another character vector, *myVariableNames*. This makes it much easier to manage when you have a more realistic data set that may contain hundreds of variables:

```
myVariableNames <- c("id", "gender", "q1", "q2", "q3", "q4")
```

Now we will do the same with our variable widths. This makes our next example, which reads multiple records per case, much easier:

```
myVariableWidths <- c(2, -1, 1, 1, 1, 1, 1)
```

Now we will put it all together in a call to the `read.fwf` function:

```
mydata <- read.fwf(
  file        = myfile,
  width       = myVariableWidths,
  col.names   = myVariableNames,
  row.names   = "id",
  na.strings  = "",
  fill        = TRUE,
  strip.white = TRUE)
```

Running this code will read the file in exactly the same was as in the previous example where we filled in all the values directly into the argument fields.

6.6.2 Example Programs for Reading Fixed-Width Text Files, One Record per Case

These programs do not save the data as they skip the workshop variable for demonstration purposes.

SAS Program for Fixed-Width Text Files, One Record per Case

```
* Filename: ReadFWF1.sas ;

LIBNAME myLib  C:\myRfolder ;
DATA myLib.mydata;
INFILE  \myRfolder\mydataFWF.txt  MISSOVER;
INPUT id 1-2 workshop 3 gender $ 4
  q1 5  q2 6  q3 7  q4 8;
RUN;
```

SPSS Program for Fixed-Width Text Files, One Record per Case

```
* Filename: ReadFWF1.sps .

CD  C:\myRfolder .

DATA LIST FILE= mydataFWF.txt  RECORDS=1
 /1 id 1-2 workshop 3 gender 4 (A)  q1 5  q2 6  q3 7  q4 8.
LIST.
```

R Program for Fixed-Width Text Files, One Record per Case

```
# Filename: ReadFWF1.R

setwd("c:/myRfolder")
mydata <- read.fwf(
   file        = "mydataFWF.txt",
   width       = c(2, -1, 1, 1, 1, 1, 1),
   col.names   = c("id", "gender", "q1", "q2", "q3", "q4"),
   row.names   = "id",
   na.strings  = "",
   fill        = TRUE,
   strip.white = TRUE)
mydata

# Again using "macro substitution".

myfile <- "mydataFWF.txt"
myVariableNames  <- c("id", "gender", "q1", "q2", "q3", "q4")
myVariableWidths <- c(2, -1, 1, 1, 1, 1, 1)

mydata <- read.fwf(
   file        = myfile,
   width       = myVariableWidths,
   col.names   = myVariableNames,
   row.names   = "id",
   na.strings  = "",
   fill        = TRUE,
   strip.white = TRUE)
mydata
```

6.7 Reading Fixed-Width Text Files, Two or More Records per Case

It is common to have to read several records per case. In this section we will read two records per case, but it is easy to generalize from here to any number of records. This section builds on the section above, so if you have not just finished reading it, you will want to now. We will only use the macro substitution form in this example.

First, we will store the filename in the character vector named myfile:

```
myfile <- "/mydataFWF.txt"
```

Next, we will store the variable names in another character vector. We will pretend that our same file now has two records per case with q1 to q4 on the first record and q5 to q8 in the same columns on the second. Even though id, workshop, and gender appear on every line, we will not read them again from the second line. Here are our variable names:

```
myVariableNames <- c("id", "workshop", "gender",
  "q1", "q2", "q3", "q4",
  "q5", "q6", "q7", "q8" )
```

Now we need to specify the columns to read. We must store the column widths for each line of data (per case) in their own vectors. Note that on record 2 we begin with −2, −1, −1 to skip the values for id, workshop, and gender.

```
myRecord1Widths <- c( 2, 1, 1, 1, 1, 1, 1)
myRecord2Widths <- c(-2,-1,-1, 1, 1, 1, 1)
```

Next, we need to store both of the above variables in a list. The `list` function below combines the two record width vectors into one list named myVariableWidths:

```
myVariableWidths <- list(myRecord1Widths, myRecord2Widths)
```

Let us look at the new list:

```
> myVariableWidths

 [[1]]
 [1] 2 1 1 1 1 1 1

 [[2]]
 [1] -2 -1 -1  1  1  1  1
```

You can see that the component labeled [[1]] is the first numeric vector and the one labeled [[2]] is the second. In SAS you would tell it that there are two records per case by using "`#2`" to move to the second record. Similarly, SPSS uses "`/2`". R uses a *very* different approach to change records! It is the fact that the list of record lengths contains two components that tells R we have two records per case. When it finishes using the record widths stored in the first component of the list, it will automatically move to the second record, and so on.

Now we are ready to use the `read.fwf` function to read the data file:

```
> mydata <- read.fwf(
+    file        = myfile,
+    width       = myVariableWidths,
+    col.names   = myVariableNames,
+    row.names   = "id",
+    na.strings  = "",
+    fill        = TRUE,
+    strip.white = TRUE)

Warning message:

In readLines(file, n = thisblock) :
  incomplete final line found on  mydataFWF.txt

  workshop gender q1 q2 q3 q4 q5 q6 q7 q8
1        1      f  1  1  5  1  2  1  4  1
3        1      f  2  2  4  3  3  1 NA  3
5        1      m  3  5  2  4  5  4  5  5
7        1      m  5  3  4  4  4  5  5  5
```

You can see we now have only four records and eight q variables, so it has worked well. It is also finally obvious that the row names do not always come out as simple sequential numbers. It just so happened that that is what we have had until now. Because we are setting our row names from our id variable, and we are reading two records per case, we end up with only the odd-numbered values. However, if we had let R create its own row names, they would have ended up, “1,” “2,” “3,” and “4.” The odd-numbered row names also help us understand why no value of gender is now missing: we did not read gender from the fourth record in the file.

I did not press the Enter key at the end of the last line of data, causing R to think that the final line was incomplete. That does not cause problems.

6.7.1 Example Programs to Read Fixed-Width Text Files with Two Records per Case

SAS Program to Read Two Records per Case

```
* Filename: ReadFWF2.sas ;

DATA temp;
INFILE \myRfolder\mydataFWF.txt MISSOVER;
INPUT
  #1 id 1-2 workshop 3 gender 4  q1 5  q2 6  q3 7  q4 8
  #2                             q5 5  q6 6  q7 7  q8 8;

PROC PRINT;
RUN;
```

SPSS Program to Read Two Records per Case

```
* Filename: ReadFWF2.sps .
```

```
DATA LIST FILE= \myRfolder\mydataFWF.txt  RECORDS=2
 /1 id 1-2 workshop 3 gender 4 (A) q1 5 q2 6 q3 7 q4 8
 /2                                q5 5 q6 6 q7 7 q8 8.
LIST.
```

R Program to Read Two Records per Case

```
# Filename: ReadFWF2.R

setwd("C:/myRfolder")

# Set all the values to use.
myfile <- "mydataFWF.txt"
myVariableNames  <- c("id", "workshop", "gender",
  "q1", "q2", "q3", "q4",
  "q5", "q6", "q7", "q8")
myRecord1Widths  <- c( 2, 1, 1, 1, 1, 1, 1)
myRecord2Widths  <- c(-2,-1,-1, 1, 1, 1, 1)
myVariableWidths <- list(myRecord1Widths, myRecord2Widths)

#Now plug them in and read the data:
mydata <- read.fwf(
   file        = myfile,
   width       = myVariableWidths,
   col.names   = myVariableNames,
   row.names   = "id",
   na.strings  = "",
   fill        = TRUE,
   strip.white = TRUE )
mydata
```

6.8 Reading Excel Files

The easiest way to read or write Excel files is to use Hans-Peter Suter's aptly named `xlsReadWrite` package [53]. You begin its installation as usual:

```
install.packages("xlsReadWrite")
```

However, when you load the package from your library for the first time, it will tell you that you need an additional command to complete the installation:

```
library("xlsReadWrite")
xls.getshlib()
```

The xls.getshlib gets a binary file that is not distributed through CRAN. You only need to run that function once when you first install xlsReadWrite.

Using xlsReadWrite is very easy. You can read a file using:

```
> setwd("c:/myRfolder")

> mydata <- read.xls("mydata.xls")

> mydata
  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
...
```

If you have an id variable in the first column and you do not name it, unlike other R functions, it will not assume that it should go in the row names attribute. Instead, it will name the variable V1. You can transfer the values from any variable to the row names attribute by adding the rowNames argument.

As of this writing, the package is not able to read or write files in Excel's newer XLSX format. You can read such files using ODBC as shown in the next section. You can also read XLSX files and save them as XLS files using Excel or the free OpenOffice.org (http://www.openoffice.org/) or LibreOffice (http://www.documentfoundation.org/).

There is a "Pro" version of xlsReadWrite that has added features, such as the ability to read specific cell ranges or to to append what it writes to the bottom of an existing Excel file. It also has more functions to convert Excel date and time variables. It is available at http://www.swissr.org/.

6.8.1 Example Programs for Reading Excel Files

SAS Program for Reading Excel Files

```
* Filename: ReadExcel.sas;

LIBNAME myLib "c:\myRfolder";

PROC IMPORT OUT = mylib.mydata
            DATAFILE = "C:\myRfolder\mydata.xls"
            DBMS = EXCELCS REPLACE;
     RANGE    = "Sheet1$";
     SCANTEXT = YES;
     USEDATE  = YES;
     SCANTIME = YES;
RUN;
```

SPSS Program for Reading Excel Files

```
* Filename: ReadExcel.sps.

GET DATA
  /TYPE=XLS
  /FILE= C:\myRfolder\mydata.xls
  /SHEET=name  Sheet1
  /CELLRANGE=full
  /READNAMES=on
  /ASSUMEDSTRWIDTH=32767.
EXECUTE.
```

R Program for Reading Excel Files

```
# Filename: ReadExcel.R

# Do this once:
install.packages("xlsReadWrite")
library("xlsReadWrite")
xls.getshlib()

# Do this each time:
library("xlsReadWrite")
setwd("c:/myRfolder")

mydata <- read.xls("mydata.xls")
mydata

save(mydata, "mydata.RData")
```

6.9 Reading from Relational Databases

R has the ability to access data in most popular database programs. The *R Data Import/Export* manual that appears in the *Help> Manuals (in PDF)* menu covers this area thoroughly. I will give a brief overview of it by using Ripley and Lapsley's `RODBC` package [47]. This package comes with the main R installation. However, it requires Microsoft's Open Database Connectivity standard (ODBC). That comes standard with Windows, but you must add it yourself if you use Macintosh, Linux, or UNIX.

Accessing a database normally requires installing one on your computer and then using your operating system to establish a connection to it. Instead, we will simulate this process by using ODBC to access our practice Excel file. You do not have to have Excel installed for this to work, but if you are not a

Windows user, you will have to install an ODBC driver to use this approach. Here is how to read an Excel file using ODBC:

```
library("RODBC")
myConnection <- odbcConnectExcel("mydata.xls")
```

Now that the connection is established, we can read it using the `sqlFetch` function and then close the connection:

```
> mydata <- sqlFetch(myConnection, "Sheet1")

> close(myConnection)

> mydata
  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
3  3        1      f  2  2  4  3
4  4        2         3  1 NA  3
5  5        1      m  4  5  2  4
6  6        2      m  5  4  5  5
7  7        1      m  5  3  4  4
8  8        2      m  4  5  5  5
```

If you do not name the id variable, R will not assume the first column is an id variable and so will not transfer its contents to the row names attribute. Instead, it will name the first column "F1."

6.10 Reading Data from SAS

If you have SAS installed on your computer, you can read SAS data sets from within R. If you have Revolution R Enterprise, a commercial version of R from Revolution Analytics, you can read SAS data sets without having SAS installed. Finally, without having SAS or Revolution R Enterprise, R can still read SAS XPORT files.

If you do have SAS installed on your machine, you can use SAS itself to help read and translate any SAS data set using the `read.ssd` function in the `foreign` package that comes with the main R distribution:

```
> library("foreign")

> mydata <- read.ssd("c:/myRfolder", "mydata",
+   sascmd = "C:/Program Files/SAS/SASFoundation/9.2/sas.exe")

> mydata
```

```
  ID WORKSHOP GENDER Q1 Q2 Q3 Q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
...
```

The `read.ssd` function call above uses three arguments:

1. The libname, or path where your SAS data set is stored. In this example, the file is stored in myRfolder on my C: drive.
2. The member name(s). In this example, I am reading mydata.sas7bdat. You do *not* list the file extension along with the member name.
3. The `sascmd` argument. This shows the full path to the sas.exe command.

If you do not have SAS installed, you can read SAS data sets in XPORT format. Although the `foreign` package reads XPORT files too, it lacks important capabilities. Functions in Harrell's `Hmisc` package add the ability to read formatted values, variable labels, and lengths.

SAS users rarely use the `LENGTH` statement, accepting the default storage method of double precision. This wastes a bit of disk space but saves programming time. However, since R saves all its data in memory, space limitations are far more important. If you use the SAS LENGTH statement to save space, the `sasxport.get` function in `Hmisc` will take advantage of it. However, unless you know a lot about how computers store data, it is probably best to only shorten the length used to store integers. The `Hmisc` package does not come with R but it is easy to install. For instructions, see Sect. 2.1, "Installing Add-on Packages."

The example below loads the two packages we need and then translates the data.

```
library("foreign")

library("Hmisc")

mydata <- sasxport.get("mydata.xpt")
```

The `sasxport.get` function has many arguments to control its actions. It is documented in An Introduction to S and the Hmisc and Design Libraries [2].

Another way to read SAS files is via the SAS ODBC Driver. It lets you read files using the `RODBC` package described in Sec. 6.9.

6.10.1 Example Programs to Write Data from SAS and Read It into R

Unlike most of our example programs, the SAS and R code here do opposite things rather than the same thing. The first program writes the data from SAS, and the second reads into R both the original SAS data set and the XPORT file created by the SAS program.

SAS Program to Write Data from SAS

```
* Filename: WriteXPORT.sas ;

LIBNAME myLib  C:\myRfolder ;
LIBNAME To_R xport  \myRfolder\mydata.xpt ;

DATA To_R.mydata;
  SET  myLib.mydata;
  RUN;
```

R Program to Read a SAS Data Set

```
# Filename: ReadSAS.R

setwd("c:/myRfolder")

# Reads ssd or sas7bdat if you have SAS installed.
library("foreign")

mydata <- read.ssd("c:/myRfolder", "mydata",
  sascmd = "C:/Program Files/SAS/SASFoundation/9.2/sas.exe")
mydata

# Reads SAS export format without installing SAS
library("foreign")
library("Hmisc")

mydata <- sasxport.get("mydata.xpt")
mydata
```

6.11 Reading Data from SPSS

If you have SPSS 16 or later, the best way to read data into R from SPSS is by using the SPSS-R Integration Plug-in. It includes support of recent features such as long file names. For details, see Sect. 3.7.

You can also install a free ODBC driver in the IBM SPSS Data Access Pack that will let you read SPSS files using the `RODBC` package described in Sec. 6.9.

If you do not have SPSS, you can read an SPSS save file using the `spss.get` function in the `foreign` package:

```
> library("foreign")
> mydata <- read.spss("mydata.sav",
+   use.value.labels = TRUE,
+   to.data.frame    = TRUE)

> mydata
  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
3  3        1      f  2  2  4  3
4  4        2         3  1 NA  3
...
```

Setting `to.data.frame` to TRUE gets the data into a data frame rather than as a list. Setting the `use.value.labels` argument to TRUE causes it to convert any variable with value labels to R factors with those labels. That keeps the labels but turns them into categorical variables. This is the default value, so I list it here only to point out its importance. Setting it to FALSE will leave your variables numeric, allowing you to calculate means and standard deviations more easily. SPSS users often have Likert scale 1 through 5 items stored as scale variables (numeric vectors in R) and have labels assigned to them. For more details about factors, Read Sect. 11.1, "Value Labels or Formats (and Measurement Level)."

See `help("read.spss")` when the `foreign` package is loaded for many more arguments that control the way the file is read.

Note that it left gender with a blank value instead of setting that to missing. You could fix that with:

```
mydata[mydata == " "] <- NA
```

or any of the other methods discussed in Section10.5, "Missing Values".

If you have an SPSS portable file, you can read that using the `spss.get` function in the `Hmisc` package. For instructions on installing Hmisc, see Sect. 2.1, "Installing Add-on Packages".

Here is an example:

```
library("Hmisc")
mydata <- spss.get("mydata.por")
```

It also has a `use.value.labels` argument, but I did not use it here.

Other useful arguments include `lowernames = TRUE` to convert all names to lowercase and `datevars` to tell R about date variables to convert. After you have loaded the `Hmisc` package, you can use `help("spss.get")` for more information.

6.11.1 Example Programs for Reading Data from SPSS

Unlike most of our example programs, the SPSS and R code here do opposite things rather than the same thing.

SPSS Program to Write a Portable Format File

```
* Filename: WritePortable.sps

CD  C:\myRfolder .
GET FILE= mydata.sav .
EXPORT OUTFILE= C:\myRfolder\mydata.por .
```

R Program to Read an SPSS Data File

```
# Filename: ReadSPSS.R

setwd("c:/myRfolder")

library("foreign")
mydata <- read.spss("mydata.sav",
  use.value.labels = TRUE,
  to.data.frame    = TRUE)
mydata

mydata[mydata == " "] <- NA

library("Hmisc")
mydata <- spss.get("mydata.por",
  use.value.labels = TRUE)
mydata

save(mydata, "mydata.RData")
```

6.12 Writing Delimited Text Files

Writing text files from R is generally much easier than reading them because you know exactly what you already have. You have no worries about extraneous commas or tab characters causing trouble.

Writing a comma-delimited file is as simple as:

```
write.csv(mydata, "mydataFromR.csv")
```

Of course you have to consider who is likely to read the file and what their concerns are. For example, an American sending a file to Europe might want to look at the help files for `read.csv2`, which uses commas for decimal points and semicolons for delimiters.

To write a tab-delimited file, there is no direct equivalent to `read.delim`. Instead, you use `write.table`. Accepting all the defaults will give you a space-delimited file with "NA" written for missing values. Many packages, SAS and

SPSS included, will print warnings when they read "NA" strings in numeric variables and then set them to missing. Here is how it works:

```
write.table(mydata, "mydata.txt")
```

If necessary, you can specify many different arguments to control what `write.table` does. Here is a more complicated example:

```
write.table(mydata,
  file      = "mydataFromR.tab",
  quote     = FALSE,
  sep       = "\t",
  na        = "",
  row.names = TRUE,
  col.names = TRUE)
```

This function call uses seven arguments:

1. The name of the R data frame to write out.
2. The `file` argument names the output text data file. R will write it to the working directory.
3. The `quote = FALSE` argument tells R not to write quotes around character data like "m" and "f." By default, it will write the quotes.
4. The `sep = "\t"` that tells it the separator (delimiter) to use between values is one single tab. Changing that to `sep = ","` would write out a comma-delimited file instead. If you did that, you would want to change the filename to "mydata2.csv."
5. The `na = ""` argument tells R not to write anything to represent missing values. By default, it will write out "NA" instead. That is what you want only if you plan to read the data back into R. Few other packages recognize NA as a code for missing values. SAS and SPSS will convert it to missing, but they will generate a lot of irritating messages, so it is probably best to use a blank.
6. The `row.names = TRUE` argument tells R to write row names in the first column of the file. In other words, it will write out an ID-type variable. This is the default value, so you do not actually need to list it here. If you do not want it to write row names, then you must use `row.names = FALSE`.
7. The `col.names = TRUE` argument tells R to write variable names in the first row of the file. This is the default value, so you do not actually need to list it here. If you do not want it to write variable names, then you must use `col.names = FALSE`. Unlike most programs, R will not write out a name for an ID variable.

6.12.1 Example Programs for Writing Delimited Text Files

SAS Program for Writing Delimited Text Files

```
* Filename: WriteDelimited.sas;
```

```
LIBNAME myLib  C:\myRfolder ;

PROC PRINT DATA=myLib.mydata; run;

PROC EXPORT DATA= MYLIB.MYDATA
            OUTFILE= "C:\myFolder\mydataFromSAS.csv"
            DBMS=CSV REPLACE;
     PUTNAMES=YES;
RUN;

PROC EXPORT DATA= MYLIB.MYDATA
            OUTFILE= "C:\myFolder\mydataFromSAS.txt"
            DBMS=TAB REPLACE;
     PUTNAMES=YES;
RUN;
```

SPSS Program for Writing Delimited Text Files

```
* Filename: WriteDelimited.sps

GET
  FILE= C:\myRfolder\mydata.sav .
DATASET NAME DataSet2 WINDOW=FRONT.

SAVE TRANSLATE OUTFILE= C:\myRfolder\mydataFromSPSS.csv
  /TYPE=CSV
  /MAP
  /REPLACE
  /FIELDNAMES
  /CELLS=VALUES.

SAVE TRANSLATE OUTFILE= C:\myRfolder\mydataFromSPSS.dat
  /TYPE=TAB
  /MAP
  /REPLACE
  /FIELDNAMES
  /CELLS=VALUES.
```

R Program for Writing Delimited Text Files

```
# Filename: WriteDelimited.R

setwd("c:/myRfolder")
```

```
write.csv(mydata, "mydataFromR.csv")

write.table(mydata, "mydataFromR.txt")

write.table(mydata,
  file      = "mydataFromR.txt",
  quote     = FALSE,
  sep       = "\t",
  na        = " ",
  row.names = TRUE,
  col.names = TRUE)

# Look at the contents of the last file.
file.show("mydataFromR.txt")
```

6.13 Viewing a Text File

When you are writing data from R, it is helpful to be able to open the file(s) under program control. That way you can adjust the arguments until you get what you need.

To look at the contents of any text file in R, you can use the `file.show` function. On Windows or Macintosh, it will open a read-only window showing you the file's contents. On Linux or UNIX it will simply list the file's contents.

Here is an example. Note that it did not write out a name for the row names variable, so the name workshop appears in the first column:

```
> file.show("mydataFromR.csv")

workshop        gender  q1      q2      q3      q4
1       R       f       1       1       5       1
2       SAS     f       2       1       4       1
...
```

6.14 Writing Excel Files

Writing an Excel file is easy using the `xlsReadWrite` package described in Sect. 6.8.

```
library("xlsReadWrite")

write.xls(mydata, "mydataFromR.xls")
```

As of this writing, `xlsReadWrite` is not able to write files in Excel's newer XLSX format. However, since almost any package can read the older XLS format, it is not a cause for concern as it was when we read Excel files.

6.14.1 Example Programs for Writing Excel Files

SAS Program for Writing Excel Files

```
* Filename: WriteExcel.sas;

LIBNAME mylib "c:\myRfolder";

PROC EXPORT DATA= MYLIB.MYDATA
            OUTFILE= "C:\myRfolder\mydata.xls"
            DBMS=EXCELCS LABEL REPLACE;
     SHEET="mydata";
RUN;
```

SPSS Program for Writing Excel Files

```
* Filename: WriteExcel.sps .

GET FILE =  C:\myRfolder\mydata.sav .
DATASET NAME DataSet2 WINDOW=FRONT.

SAVE TRANSLATE OUTFILE= C:\myRfolder\mydataFromSPSS.xls
  /TYPE=XLS
  /VERSION=2
  /MAP
  /REPLACE
  /FIELDNAMES.
EXECUTE.
```

R Program for Writing Excel Files

```
# Filename: WriteExcel.R

# Do this once:
install.packages("xlsReadWrite")
library("xlsReadWrite")
xls.getshlib()

# Do this each time:
library("xlsReadWrite")
setwd("c:/myRfolder")

load("mydata.RData")
write.xls(mydata, "mydataFromR.xls")
```

6.15 Writing to Relational Databases

Writing to databases is done in a very similar manner to reading them. See Sect. 6.9 for the software requirements. The main difference is the `readonly = FALSE` argument on the odbcConnectExcel function call:

```
library("RODBC")
myConnection <- odbcConnectExcel("mydataFromR.xls",
  readOnly = FALSE)
sqlSave(myConnection, mydata)
close(myConnection)
```

The SAS and SPSS approaches to writing to relational databases are beyond our scope.

6.16 Writing Data to SAS and SPSS

In Sect. 6.12, "Writing Delimited Text Files," we examined several ways to write text files. In this section we will use the `write.foreign` function to write out a comma-delimited text file *along with* either a SAS or SPSS program file to match. To complete the importation into SAS or SPSS, you must edit the program file in SAS or SPSS and then execute it to read the text file and finally create a data set. To begin the process, you must load the foreign package that comes with the main R distribution.

```
library("foreign")

write.foreign(mydata,
  datafile = "mydataFromR.csv",
  codefile = "mydata.sas",
  package  = "SAS")
```

This function call uses four arguments:

1. The name of the R data frame you wish to write out.
2. The `datafile` argument tells R the name of the text data file. R will write it to the current working directory unless you specify the full path in the filename.
3. The `codefile` argument tells R the filename of a program that SAS or SPSS can use to read the text data file. You will have to use this file in SAS or SPSS to read the data file and create a SAS- or SPSS-formatted file. R will write it to the current working directory unless you specify the full path in the filename.
4. The `package` argument takes the values "SAS" or "SPSS" to determine which type of program R writes to the `codefile` location. Note that these two examples write out the gender values as 1 and 2 for "f" and "m,"

respectively. It then creates SAS formats or SPSS value labels for those values, so they will display as f and m when you read them into your other package. Some people prefer other values, or they prefer converting factors to character variables before writing the file out. To change those values, read Sect. 11.1, "Value Labels or Formats (and Measurement Level)."

Here is the SAS program that R wrote:

```
*  Written by R;
*  write.foreign(mydata, datafile = "mydataFromR.txt",
   codefile = "mydataFromR.sas",  ;

PROC FORMAT;
value gender
     1 = " "
     2 = "f"
     3 = "m"
;

DATA  rdata ;
INFILE  "mydataFromR.txt"
     DSD
     LRECL= 15 ;
INPUT
 workshop
 gender
 q1
 q2
 q3
 q4
;
FORMAT gender gender. ;
RUN;
```

You can see it needs a bit of work, but you could use this to read the data into R fairly quickly and you would have the formats that you would otherwise have lacked using the other approaches to write text files.

6.16.1 Example Programs to Write Data to SAS and SPSS

This section presents programs to write a text file from R for use in any program. They can also be used to write text files and matching SAS and SPSS programs to read them.

R Program to Write Data to SAS

This program writes data to one file and a SAS program to another file. You run the SAS program in SAS to read the data into that package.

```
# Filename: WriteSAS.R

setwd("c:/myRfolder")
library("foreign")

write.foreign(mydata,
  datafile = "mydataFromR.txt",
  codefile = "mydataFromR.sas",
  package  = "SAS")

# Look at the contents of our new files.
file.show("mydataFromR.txt")
file.show("mydataFromR.sas")
```

R Program to Write Data to SPSS

This program exports data to one file and an SPSS program to another file. You run the SPSS program in SPSS to read the data into that package.

```
# Filename: WriteSPSS.R

setwd("c:/myRfolder")
library("foreign")

write.foreign(mydata,
  datafile = "mydataFromR.txt",
  codefile = "mydataFromR.sps",
  package  = "SPSS")

# Look at the contents of our new files.
file.show("mydataFromR.txt")
file.show("mydataFromR.sps")
```

7

Selecting Variables

In SAS and SPSS, selecting variables for an analysis is simple, while selecting observations is often much more complicated. In R, these two processes can be almost identical. As a result, variable selection in R is both more flexible and quite a bit more complex. However, since you need to learn that complexity to select observations, it does not require much added effort.

Selecting observations in SAS or SPSS requires the use of logical conditions with commands like IF, WHERE, SELECT IF, or FILTER. You do not usually use that logic to select variables. It is possible to do so, through the use of macros or, in the case of SPSS, Python, but it is not a standard approach. If you have used SAS or SPSS for long, you probably know dozens of ways to select observations, but you did not see them all in the first introductory guide you read. With R, it is best to dive in and see all of the methods of selecting variables because understanding them is the key to understanding other documentation, especially the help files and discussions on the R-help mailing list. Even though you select variables and observations in R using almost identical methods, I will describe them in two different chapters, with different example programs. This chapter focuses only on selecting variables. In the next chapter I will use almost identical descriptions with very similar examples for selecting observations. I do so to emphasize the similarity of the two tasks, as this is such an alien concept to SAS and SPSS users. In the relatively short Chap. 9, I will combine the methods and show how to select variables and observations simultaneously.

7.1 Selecting Variables in SAS and SPSS

Selecting variables in SAS or SPSS is quite simple. It is worth reviewing their basic methods before discussing R's approach. Our example data set contains the following variables: workshop, gender, q1, q2, q3, and q4. SAS lets you refer to them by individual name or in contiguous order separated by double dashes, “--,” as in

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_7, © Springer Science+Business Media, LLC 2011

```
PROC MEANS DATA=myLib.mydata; VAR workshop--q4;
```

SAS also uses a single dash, "-," to request variables that share a numeric suffix, even if they are not next to each other in the data set:

```
PROC MEANS DATA=myLib.mydata; VAR q1-q4;
```

You can select all variables beginning with the letter "q" using the colon operator.

```
PROC MEANS DATA=myLib.mydata; VAR q: ;
```

Finally, if you do not tell it which variable to use, SAS uses them all.

SPSS allows you to list variables names individually or with contiguous variables separated by "TO," as in

```
DESCRIPTIVES VARIABLES=gender to q4.
```

If you want SPSS to analyze all variables in a data set, you use the keyword ALL.

```
DESCRIPTIVES VARIABLES=ALL.
```

SPSS's main command language does not offer a built-in way to easily select variables that begin with a common root like "q". However, the company provides the SPSS extension command SPSSINC SELECT VARIABLES that can make this type of selection.

Now let us turn our attention to how R selects variables.

7.2 Subscripting

In Chap. 5, "Programming Language Basics," I described how you could select the elements (values) of a vector or matrix or the components (often variables) of a data frame or list using *subscripting*. Subscripting allows you to follow an object's name with selection information in square brackets:

```
vector[elements]

matrix[rows, columns]

data[rows, columns]

list[[component]]
```

As you will see throughout this chapter and the next, the selection information you place in the subscript brackets can be index values (e.g., 1, 2, 3, etc.), logical selections (e.g., gender == "f"), or names (e.g., "gender").

If you leave the subscripts out, R will process all rows and all columns. Therefore, the following three statements have the same result, a summary of all the rows (variables) and all the columnns (observations or cases) of mydata:

```
summary( mydata )
summary( mydata[ ] )
summary( mydata[ , ] )
```

This chapter focuses on the second parameter, the columns (variables).

7.3 Selecting Variables by Index Number

Coming from SAS or SPSS, you would think a discussion of selecting variables in R would begin with variable names. R can use variable names, of course, but column index numbers are more fundamental to the way R works. That is because objects in R do not have to have names for the elements or components they contain, but they always have index numbers.

Our data frame has six variables or columns, which are automatically given index numbers, or indices, of 1, 2, 3, 4, 5, and 6. You can select variables by supplying one index number or a vector of indices in subscript brackets. For example,

```
summary( mydata[ ,3] )
```

selects all rows of the third variable or column, q1. If you leave out a subscript, it will assume you want them all. If you leave the comma out completely, R assumes you want a column, so

```
summary( mydata[3] )
```

is almost the same as

```
summary( mydata[ ,3] )
```

Both refer to our third variable, q1. While the `summary` function treats the presence or absence of the comma in the same way, some functions will have problems. That is because with a comma, the variable selection passes a *vector* and without a comma, it passes a *data frame* that contains only one vector. To the `summary` function the result is the same, but some functions prefer one form or the other. See Chap. 10.19, “Converting Data Structures,” for details.

To select more than one variable using indices, you combine the indices into a vector using the `c` function. Therefore, this will analyze variables 3 through 6.

```
summary( mydata[ c(3,4,5,6) ] )
```

You will see the c function used in many ways in R. Whenever R requires one object and you need to supply it several, it combines the several into one. In this case, the several index numbers become a single numeric vector.

The colon operator ":" can generate a numeric vector directly, so

```
summary( mydata[3:6] )
```

will use the same variables.

Unlike SAS's use of

```
workshop--q4
```

or SPSS's use of

```
workshop TO q4
```

the colon operator is not just shorthand. We saw in an earlier chapter that entering 1:N causes R to generate the sequence, 1, 2, 3,. . . N. If you use a negative sign on an index, you will exclude those columns. For example,

```
summary( mydata[ -c(3,4,5,6) ] )
```

will analyze all variables *except for* variables 3, 4, 5, and 6. Your index values must be either all positive or all negative. Otherwise, the result would be illogical. You cannot say, "include only these" and "include all but these" at the same time. Index values of zero are accepted but ignored.

The colon operator can abbreviate patterns of numbers, but you need to be careful with negative numbers. If you want to exclude columns 3:6, the following approach will not work:

```
> -3:6
 [1] -3 -2 -1 0 1 2 3 4 5 6
```

This would, of course, generate an error since you cannot exclude 3 and include 3 at the same time. Adding parentheses will clarify the situation, showing R that you want the minus sign to apply to just the set of numbers from +3 through +6 rather than –3 through +6:

```
> -(3:6)
 [1] -3 -4 -5 -6
```

Therefore, we can exclude variables 3 through 6 with

```
summary(  mydata[ -(3:6) ] )
```

If you find yourself working with a set of variables repeatedly, you can easily save a vector of indices so you will not have to keep looking up index numbers:

```
myQindices <- c(3, 4, 5, 6)
summary( mydata[myQindices] )
```

You can list indices individually or, for contiguous variables, use the colon operator. For a large data set, you could use variables 1, 3, 5 through 20, 25, and 30 through 100 as follows:

```
myindices <- c(1, 3, 5:20, 25, 30:100)
```

This is an important advantage of this method of selecting variables. Most of the other variable selection methods do not easily allow you to select mixed sets of contiguous and noncontiguous variables, as you are used to doing in either SAS or SPSS. For another way to do this, see "Selecting Variables Using the `subset` Function", Sect. 7.9.

If your variables follow patterns such as every other variable or every tenth, see Chap. 12 for ways to generate other sequences of index numbers.

The `names` function will extract a vector of variable names from a data frame. The `data.frame` function, as we have seen, combines one or more vectors into a data frame and creates default row names of "1," "2," "3," etc. Combining these two functions is one way to quickly generate a numbered list of variable names that you can use to look up index values:

```
> data.frame( names(mydata) )


  names.mydata.
1      workshop
2        gender
3            q1
4            q2
5            q3
6            q4
```

It is easy to rearrange the variables to put the four q variables in the beginning of the data frame. In that way, you will easily remember, for example, that q3 has an index value of 3 and so on.

Storing them in a separate data frame is another way to make indices easy to remember for sequentially numbered variables like these. However, that approach runs into problems if you sort one data frame, as the rows then no longer match up in a sensible way. Correlations between the two sets would be meaningless.

The `ncol` function will tell you the number of columns in a data frame. Therefore, another way to analyze all your variables is

```
summary( mydata[ 1:ncol(mydata)  ]  )
```

If you remember that q1 is the third variable and you want to analyze all of the variables from there to the end, you can use

```
summary( mydata[ 3:ncol(mydata)  ]  )
```

7.4 Selecting Variables by Column Name

Variables in SAS and SPSS are required to have names, and those names must be unique. In R, you do not need them since you can refer to variables by index number as described in the previous section. Amazingly enough, the names do not have to be unique, although having two variables with the same name would be a terrible idea! R data frames usually include variable names, as does our example data: workshop, gender, q1, q2, q3, q4.

Both SAS and SPSS store their variable names within their data sets. However, you do not know exactly where they reside within the data set. Their location is irrelevant. They are in there somewhere, and that is all you need to know. However, in R, they are stored within a data frame in a place called the *names attribute*. The `names` function accesses that attribute, and you can display them by entering

```
> names(mydata)

 [1] "workshop" "gender"   "q1"      "q2"      "q3"      "q4"
```

To select a column by name, you put it in quotes, as in

```
summary( mydata["q1"] )
```

R still uses the form

```
mydata[row, column]
```

However, when you supply only one index value, it assumes it is the column. So

```
summary( mydata[ ,"q1"] )
```

works as well. Note that the addition of the comma before the variable name is the only difference between the two examples above. While the `summary` function treats the presence or absence of a comma the same, some functions will have problems. That is because with a comma, the selection results in a vector, and without a comma, the selection is a data frame containing only that vector. See Sect. 10.19 for details.

If you have more than one name, combine them into a single character vector using the `c` function. For example,

```
summary( mydata[ c("q1","q2","q3","q4") ] )
```

Since it is tedious to write out so many variables repeatedly, sets of variable names are often stored in character vectors. This allows you to easily use the vector as what SAS or SPSS would call *macro substitution*. For example, we can make that same selection with:

```
myQnames <- c("q1", "q2", "q3", "q4")
summary( mydata[myQnames] )
```

When I start working with a new data set, I often create several sets of variables like that and use them throughout my analysis. I usually try to make them as short and descriptive as possible For example, "Qs" for questions and "demos" for demographics. However, throughout this chapter I have selected the questions using several methods and I want the names longer to clarify the examples.

In that last example, the q variable names all ended in numbers, which would have allowed SAS users to refer to them as q1-q4. Although we have seen that R's colon operator can use `1:4` to generate 1, 2, 3, 4, it does not work directly with character prefixes. So the form q1:q4 does not work in this context. However, you can paste the letter "q" onto the numbers you generate using the `paste` function:

```
myQnames <- paste( "q", 1:4, sep = "")

summary( mydata[myQnames] )
```

The `paste` function call above has three arguments:

1. The string to paste, which for this example is just the letter "q."
2. The object to paste it to, which is the numeric vector 1, 2, 3, 4 generated by the colon operator 1:4.
3. The *sep*arator character to paste between the two. Since this is set to `""`, the function will put nothing between "q" and "1," then "q" and "2," and so on.

R will store the resulting names "q1," "q2," "q3," "q4" in the character vector myQnames. You can use this approach to generate variable names to use in a variety of circumstances. Note that merely changing the 1:4 above to 1:400 would generate the sequence from q1 to q400.

R can easily generate other patterns of repeating values that you can use to create variable names. For details, see Chap. 12, "Generating Data."

For another way to select variables by name using the colon operator, see "Selecting Variables Using the Subset Function," Sect. 7.9.

7.5 Selecting Variables Using Logic

You can select a column by using a logical vector of TRUE/FALSE values. You can enter one manually or create one by specifying a logical condition. Let us begin by entering one manually. For example,

```
summary( mydata[ c(FALSE, FALSE, TRUE, FALSE, FALSE, FALSE) ] )
```

will select the third column, q1, because the third value is TRUE and the third column is q1. In SAS or SPSS, the digits 1 and 0 can represent TRUE and FALSE, respectively. They can do this in R, but they first require processing by the `as.logical` function. Therefore, we could also select the third variable with

```
summary( mydata[ as.logical( c(0, 0, 1, 0, 0, 0) ) ] )
```

If we had not converted the 0/1 values to logical FALSE/TRUE, the above function call would have asked for two variables with index values of zero. Zero is a valid value, but it is ignored. It would have then asked for the variable in column 1, which is workshop. Finally, it would have asked for three more variables in column zero. The result would have been an analysis only for the first variable, workshop. It would have been a perfectly valid, if odd, request!

Luckily, you do not have to actually enter logical vectors like those above. Instead, you will generate a vector by entering a logical statement such as

```
names(mydata) == "q1"
```

That logical comparison will generate the following logical vector for you:

```
FALSE, FALSE, TRUE, FALSE, FALSE, FALSE
```

Therefore, another way of analyzing q1 is

```
summary( mydata[ names(mydata) == "q1" ] )
```

While that example is good for educational purposes, in actual use you would prefer one of the shorter approaches using variable names:

```
summary( mydata["q1"] )
```

Once you have mastered the various approaches of variable selection, you will find yourself alternating among the methods, as each has its advantages in different circumstances.

The "==" operator compares every element of a vector to a value and returns a logical vector of TRUE/FALSE values. The vector length will match the number of variables, not the number of observations, so we cannot store it in our data frame. So if we assigned it to an object name, it would just exist as a vector in our R workspace. As we will see in the next chapter, a similar selection on observations can be stored in the data frame very much like SPSS's filter variables.

The "!" sign represents NOT, so you can also use that vector to get all of the variables except for q1 using the form

```
summary( mydata[ !names(mydata) == "q1" ] )
```

To use logic to select multiple variable names, we can use the OR operator, "|". For example, select q1 through q4 with the following approach. Complex selections like this are much easier when you do it in two steps. First, create the logical vector and store it; then use that vector to do your selection. In the name myQtf below, I use the "tf" part to represent TRUE/FALSE. That will help us remind that this is a logical vector.

```
myQtf <- names(mydata) == "q1" |
         names(mydata) == "q2" |
         names(mydata) == "q3" |
         names(mydata) == "q4"
```

Then we can get summary statistics on those variables using

```
summary( mydata[myQtf] )
```

Whenever you are making comparisons to many values, you can use the %in% operator. This will generate exactly the same logical vector as the OR example above:

```
myQtf <- names(mydata) %in% c("q1","q2","q3","q4")

summary( mydata[myQtf] )
```

You can easily convert a logical vector into an index vector that will select the same variables. For details, see "Converting Data Structures," Sect. 10.19.

7.6 Selecting Variables by String Search (varname: or varname1-varnameN)

You can select variables by searching all of the variable names for strings of text. This approach uses the methods of selection by index number, name, and logic as discussed above, so make sure you have mastered them before trying this.

SAS uses the form:

```
VAR q: ;
```

to select all of the variables that begin with the letter q. SAS also lets you select variables in the form

```
PROC MEANS; VAR q1-q4;
```

which gets only the variables q1, q2, q3, and q4 regardless of where they occur in the data set or how many variables may lie in between them. The searching approach we will use in R handles both cases.

The main SPSS syntax does not offer this type of selection but it can do full string searches via Python using the SPSS extension command SPSSINC SELECT VARIABLES.

R searches variable names for patterns using the `grep` function. The name `grep` itself stands for *g*lobal *r*egular *e*xpression *p*rint. It is just a fancy name for a type of search.

The `grep` function creates a vector containing variable selection criteria we need in the form of indices, names, or TRUE/FALSE logical values. The `grep` function and the rules that it follows, called *regular expressions*, appear in many different software packages and operating systems.

SAS implements this type of search in the PRX function (Perl Regular eXpressions), although it does not need it for this type of search. Below we will use the `grep` function to find the index numbers for names for those that begin with the letter q:

```
myQindices <- grep("^q", names(mydata), value = FALSE)
```

The `grep` function call above uses three arguments.

1. The first is the command string, or regular expression, "`^p`", which means, "find strings that begin with lowercase p." The symbol "`^`" represents "begins with." You can use any regular expression here, allowing you to search for a wide range of patterns in variable names. We will discuss using wildcard patterns later.
2. The second argument is the character vector that you wish to search, which, in our case, is our variable names. Substituting `names(mydata)` here will extract those names.
3. The `value` argument tells it what to return when it finds a match. The goal of `grep` in any computer language or operating system is to find patterns. A value of TRUE here will tell it to return the variable names that match the pattern we seek. However, in R, indices are more important than names, so the default setting is FALSE to return indices instead. We could leave it off in this particular case, but we will use it the other way in the next example, so we will list it here for educational purposes.

The contents of myQindices will be 3, 4, 5, 6. In all our examples that use that name, it will have those same values.

To analyze those variables, we can then use

```
summary( mydata[myQindices] )
```

Now let us do the same thing but have the `grep` function save the actual variable names. All we have to do is set `value = TRUE`.

```
myQnames <- grep("^q", names(mydata), value = TRUE)
```

The character vector myQnames now contains the variable names "q1," "q2," "q3," and "q4," and we can analyze those variables with

```
summary(  mydata[myQnames]  )
```

This approach gets what we expected: variable names. Since it uses names, it makes much more sense to a SAS or SPSS user. So, why did I not do this first? Because in R, indices are more flexible than variable names.

Finally, let us see how we would use this search method to select variables using logic. The `%in%` operator works just like the IN operator in SAS. It finds things that occur in a set of character strings. We will use it to find when a member of all our variable names (stored in mynames) appears in the list of names beginning with "q" (stored in myQnames). The result will be a logical set of TRUE/FALSE values that indicate that the q variables are the last four:

```
FALSE, FALSE, TRUE, TRUE, TRUE, TRUE
```

We will store those values in the logical vector myQtf:

```
myQtf <- names(mydata) %in% myQnames
```

Now we can use the myQtf vector in any analysis we like:

```
summary( mydata[myQtf] )
```

It is important to note that since have been searching for variables that begin with the letter "q," our program would have also found variables qA and qB if they had existed. We can narrow our search with a more complex search expression that says the letter "q" precedes at least one digit. This would give us the ability to simulate SAS's ability to refer to variables that have a numeric suffix, such as "var1-var100."

This is actually quite easy, although the regular expression is a bit cryptic. It requires changing the myQnames line in the example above to the following:

```
myQnames <- grep("^q[1-9]", names(mydata), value = TRUE)
```

This regular expression means "any string that begins with 'q,' and is followed by one or more numerical digits." Therefore, if they existed, this would select q1, q27, and q1old but not qA or qB. You can use it in your programs by simply changing the letter q to the root of the variable name you are using.

You may be more familiar with the search patterns using wildcards in Microsoft Windows. That system uses "*" to represent any number of characters and "?" to represent any single character. So the wildcard version of any variable name beginning with the letter q is "q*." Computer programmers call this type of symbol a "glob," short for global. R lets you convert globs to regular expressions with the `glob2rx` function. Therefore, we could do our first `grep` again in the form

```
myQindices <- grep(glob2rx("q*"), names(mydata), value = FALSE)
```

Unfortunately, wildcards or globs are limited to simple searches and cannot do our example of q ending with any number of digits.

7.7 Selecting Variables Using $ Notation

You can select a column using $ notation, which combines the name of the data frame and the name of the variable within it, as in

```
summary( mydata$q1 )
```

This is referred to in several ways in R, including "$ prefixing," "prefixing by dataframe$," or "$ notation." When you use this method to select multiple variables, you need to combine them into a single object like a data frame, as in

```
summary( data.frame( mydata$q1, mydata$q2 ) )
```

Having seen the `c` function, your natural inclination might be to use it for multiple variables as in

```
summary( c( mydata$q1, mydata$q2 ) )  # Not good!
```

This would indeed make a single object, but certainly not the one a SAS or SPSS user expects. The `c` function would combine them both into a single variable with twice as many observations! The `summary` function would then happily analyze the new variable. When the `data.frame` function combines vectors into a single data frame, they remain separate vectors within that data frame. That is what we want here.

An important limitation of dollar notation is that you cannot use it with a matrix. Recall that in Sect. 5.3.4 we put our q variables into mymatrix. The variable names went along with the vectors. Therefore, this form would work:

```
mymatrix[ ,"q1"]  # Good
```

but this would not:

```
mymatrix$q1  # Not good!
```

As a result, some R users who use matrices that contain row and column names tend to prefer using names in subscripts since this works with matrices and data frames.

7.8 Selecting Variables by Simple Name

This section introduces the use of short names for variables stored in a data frame, like gender instead of mydata$gender. I will cover the technical details in Chap. 13, "Managing Your Files and Workspace."

In SAS and SPSS, you refer to variables by short names like gender or q1. You might have many data sets that contain a variable named gender, but there is no confusion since you have to specify the data set in advance. In SAS, you can specify the data set by adding the `DATA=` option on every procedure. Alternatively, since SAS will automatically use the last data set you created, you can pretend you just created a data set by using:

```
OPTIONS _LAST_=myLib.mydata;
```

Every variable selection thereafter would use that data set.

In SPSS, you clarify which data set you want to use by opening it with GET FILE. If you have multiple data sets open, you instead use DATASET NAME.

In R, the potential for confusing variable names is greater because it is much more flexible. For example, you can actually correlate a variable stored in one data frame with a variable stored in a *different data frame*! All of the variable selection methods discussed above made it perfectly clear which data frame to use, but they required extra typing. You can avoid this extra typing in several ways.

7.8.1 The `attach` Function

One approach R offers to simplify the selection of variables is the `attach` function. You attach a data frame using the following function call:

```
attach(mydata)
```

Once you have done that, you can refer to just q1, and R will know which one you mean. With this approach, getting summary statistics might look like

```
summary(q1)
```

or

```
summary( data.frame(q1, q2, q3, q4) )
```

If you finish with that data set and wish to use another, you can detach it with

```
detach( mydata )
```

Objects will detach automatically when you quit R, so using `detach` is not that important unless you need to use those variable names stored in a different data frame. In that case, detach one file before attaching the next.

The `attach` function works well when selecting existing variables, but it is best avoided when creating them. An attached data frame can be thought of as a temporary copy, so changes to existing variables will be lost. Therefore, when adding new variables to a data frame, you need to use any of the other above methods that make it absolutely clear where to store the variable. Afterward, you can detach the data and attach it again to gain access to the modified or new variables. We will look at the `attach` function more thoroughly in Chap. 13, “Managing Your Files and Workspace.”

7.8.2 The with Function

The with function is another way to use short variable names. It is similar to using the attach function, followed by any other *single* function, and then followed by a detach function. The following is an example:

```
with( mydata, summary( data.frame(q1, q2, q3, q4) ) )
```

It lets you use simple names and even lets you create variables safely. The downside is that you must repeat it with every function, whereas you might need the attach function only once at the beginning of your program. The added set of parentheses also increases your odds of making a mistake. To help avoid errors, you can type this as

```
with( mydata,
  summary( data.frame(q1, q2, q3, q4) )
)
```

7.8.3 Using Short Variable Names in Formulas

A third way to use short variable names works only with *modeling functions*. Modeling functions use formulas to perform analyses like linear regression or analysis of variance. They also have a data argument that specifies which data frame to use. This keeps formulas much shorter.

At first glance, R's data argument looks just like SAS's DATA option. However, while each SAS procedure has a DATA option, R's data argument is found usually only in modeling functions. In addition, R's data argument applies only to the modeling formula itself!

Here are two ways to perform a linear regression. First, using dollar notation:

```
lm( mydata$q4 ~ mydata$q1 + mydata$q2 + mydata$q3 )
```

The following is the same regression, using the data argument to tell the function which data frame to use:

```
lm(q4 ~ q1 + q2 + q3, data = mydata)
```

As formulas get longer, this second approach becomes much easier. For functions that feature a data argument, this is the approach I recommend. It is easier to use than either the attach or with functions. It also offers other benefits when making predictions from a model. We will defer that discussion to Chap. 17, "Statistics."

To use this approach, all of the data must reside in the same data frame, making it less flexible. However, it is usually a good idea to have all of the variables in the same data frame anyway.

That rule has important implications that may not occur to you at first. Recall that we initially created our variables as vectors and then combined them into a data frame. Until we deleted the redundant vectors, they existed in our workspace both in and outside of the data frame. Any nonmodeling function would choose a vector if you referred to it by its short name. But in a modeling function, using the `data` argument would force R to use the variable in the data frame instead. If you used a modeling function and *did not* use the `data` argument, then the function would use the variables stored outside the data frame. In this example, the two sets of variables were identical, but that is not always the case.

It is also important to know that the `data = mydata` argument applies *only* to the variables specified in the `formula` argument. Some modeling functions can specify which variables to use *without* specifying a formula. In that case, you must use an alternate approach (`attach` or `with`) if you wish to use shorter variable names. We will see an example of this when doing t-tests in Chap. 17.

7.9 Selecting Variables with the `subset` Function

R has a `subset` function that you can use to select variables (and observations). It is the easiest way to select contiguous sets of variables by name such as in SAS

```
PROC MEANS; VAR q1--q4;
```

or in SPSS

```
DESCRIPTIVES VARIABLES=q1 to q4.
```

It follows the form

```
subset(mydata, select = q1:q4)
```

For example, when used with the `summary` function, it would appear as

```
summary( subset(mydata, select = q1:q4 ) )
```

or

```
summary( subset(mydata, select = c(workshop, q1:q4) ) )
```

The second example above contains three sets of parentheses. It is very easy to make mistakes with so many nested functions. A syntax-checking editor will help. Another thing that helps is to split them across multiple lines:

```
summary(
  subset(mydata, select = c(workshop, q1:q4) )
)
```

It is interesting to note that when using the `c` function within the `subset` function's `select` argument, it combines the variable names, not the vectors themselves. So the following example will analyze the two variables separately:

```
summary(
  subset(mydata, select = c(q1,q2) )  # Good
)
```

That is very different from

```
summary( c(mydata$q1, mydata$q2) )  # Not good
```

which combines the two vectors into one long one before analysis.

While the form `1:N` works throughout R, the form `var1:varN` is unique to the `subset` function. That and its odd use of the `c` function in combining variable names irritates some R users. I find that its usefulness outweighs its quirks.

7.10 Selecting Variables by List Subscript

Our data frame is also a list. The components of the list are vectors that form the columns of the data frame. You can address these components of the list using a special type of subscripting. You place an index value after the list's name enclosed in two square brackets. For example, to select our third variable, we can use

```
summary( mydata[[3]] )
```

With this approach, the colon operator will not extract variables 3 through 6:

```
mydata[[3:6]]  # Will NOT get variables 3 through 6.
```

7.11 Generating Indices A to Z from Two Variable Names

We have discussed various variable selection techniques. Now we are ready to examine a method that blends several of those methods together. If you have not mastered the previous examples, now would be a good time to review them.

We have seen how the colon operator can help us analyze variables 3 through 6 using the form

```
summary( mydata[3:6] )
```

With that method, you have to know the index numbers, and digging through lists of variables can be tedious work. However, we can have R do that work for us, finding the index value for any variable name we like. This call to the `names` function,

```
names(mydata) == "q1"
```

will generate the logical vector

```
FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE
```

because q1 is the third variable. The `which` function will tell us the index values of any TRUE values in a logical vector, so

```
which( names(mydata) == "q1" )
```

will yield a value of 3. Putting these ideas together, we can find the index number of the first variable we want, store it in myqA, then find the last variable, store it in myqZ, and then use them with the colon operator to analyze our data from A to Z:

```
myqA <- which( names(mydata) == "q1" )
myqZ <- which( names(mydata) == "q4" )

summary( mydata[ ,myqA:myqZ ] )
```

7.11.1 Selecting Numeric or Character Variables

When a data frame contains both numeric and character variables, it can be helpful to select all of one or the other. SAS does this easily; SPSS would require a Python program to do it. For example, in SAS to print only the numeric variables followed by only the character ones you could use:

```
PROC PRINT; VAR _NUMERIC_;
PROC PRINT; VAR _CHARACTER_;
```

If you wanted to limit your selection to the specific type of variables that fall between variables A and Z, you would use:

```
PROC PRINT; VAR A-NUMERIC-Z;
PROC PRINT; VAR A-CHARACTER-Z;
```

This is easy to do in R, since the `class` function can check the type of variable. There is a series of functions that test if a variable's class is numeric, character, factor, or logical. Let us use the `is.numeric` function to see if some variables are numeric:

```
> is.numeric( mydata$workshop )
[1] FALSE

> is.numeric( mydata$q1 )
[1] TRUE
```

So we see that workshop is not numeric (it is a factor) but q1 is. We would like to apply that test to each variable in our data frame. Unfortunately, that puts us into territory that we will not fully cover until Sect. 10.2. However, it is not much of a stretch to discuss some of it here. The `sapply` function will allow us to use the `is.numeric` function with each variable using the following form:

```
> myNums <- sapply(mydata, is.numeric)

> myNums

workshop   gender       q1       q2       q3       q4
   FALSE    FALSE     TRUE     TRUE     TRUE     TRUE
```

Since myNums is a logical vector that contains the selection we seek, we can now easily perform any analysis we like on only those variables using the form:

```
> print( mydata[myNums] )

  q1 q2 q3 q4
1  1  1  5  1
2  2  1  4  1
3  2  2  4  3
...
```

This example could easily be changed to select only character variables using the `is.character` function or factors using the `is.factor` function.

We can also extend this idea to selecting only numeric variables that appear between any two other variables. To do so, we need to refer back to the previous section. Let us assume we want to get all the numeric variables that lie between gender and q3.

First we need to determine the index numbers for the variables that determine our range of interest.

```
> myA <- which( names(mydata) == "gender" )

> myA

[1] 2

> myZ <- which( names(mydata) == "q3" )

> myZ

[1] 5
```

So we see that part of our goal is to focus on variables 2 through 5. In our simple data set, that is obvious, but in a more realistic example we would want to be able to extract the values.

Next we need to create a logical vector that shows when the full range of index values falls within the range we seek. We can do this with:

```
> myRange <- 1:length(mydata) %in% myA:myZ

> myRange

[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

Recall that the length of a data frame is the number of variables it contains, and the `%in%` function finds when the elements of one vector are contained within another. Knowing the values in our data set, we could have written that statement as:

```
myRange <- 1:6 %in% 2:5
```

We now have two logical vectors: myNums, which shows us which are numeric, and myRange which shows the range of variables in which we are interested. We can now combine them and perform an analysis on the numeric variables between gender and q3 with the following:

```
> print( mydata[ myNums & myRange ] )
  q1 q2 q3
1  1  1  5
2  2  1  4
3  2  2  4
...
```

Here is a warning that will remain cryptic until you read Sect. 10.2. The following will not work as you might expect it to:

```
> apply(mydata, 2, is.numeric)

workshop   gender       q1       q2       q3       q4
   FALSE    FALSE    FALSE    FALSE    FALSE    FALSE
```

Why? Because the `apply` function coerces a data frame into becoming a matrix. A matrix that contains *any* factors or character variables will be coerced into becoming a character matrix!

```
> apply(mydata, 2, class)

   workshop      gender          q1          q2          q3...
"character" "character" "character" "character" "character"...
```

7.12 Saving Selected Variables to a New Data Set

You can use any variable selection method to create a new data frame that contains only those variables. If we wanted to create a new data frame that contained only the q variables, we could do so using any method described earlier. Here are a few variations:

```
myqs <- mydata[3:6]

myqs <- mydata[ c("q1","q2","q3","q4") ]
```

This next example will work, but R will name the variables "mydata.q1," "mydata.q2," and so forth, showing the data frame from which they came:

```
myqs <- data.frame(mydata$q1, mydata$q2,
                   mydata$q3, mydata$q4)
```

You can add variable name indicators to give them any name you like. With this next one, we are manually specifying original names:

```
myqs <- data.frame(q1 = mydata$q1, q2 = mydata$q2,
                   q3 = mydata$q3, q4 = mydata$q4)
```

Using the `attach` function, the `data.frame` function leaves the variable names in their original form:

```
attach(mydata)
myqs <- data.frame(q1, q2, q3, q4)
detach(mydata)
```

Finally, we have the `subset` function with its unique and convenient use of the colon operator directly on variable names:

```
myqs <- subset(mydata, select = q1:q4)
```

7.13 Example Programs for Variable Selection

In the examples throughout this chapter, we used the `summary` function to demonstrate how a complete analysis request would look. However, here we will use the `print` function to make it easier to see the result of each selection when you run these programs. Even though

```
mydata["q1"]
```

is equivalent to

```
print( mydata["q1"] )
```

because print is the default function, we will use the longer form because it is more representative of its look with most functions. As you learn R, you will quickly choose the shorter approach when printing.

For most of the programming examples in this book, the SAS and SPSS programs are shorter because the R programs demonstrate R's greater flexibility. However, in the case of variable selection, SAS and SPSS have a significant advantage in ease of use. These programs demonstrate roughly equivalent features.

7.13.1 SAS Program to Select Variables

```
* Filename: SelectingVars.sas;

LIBNAME myLib  C:\myRfolder ;
OPTIONS _LAST_=myLib.mydata;

PROC PRINT; RUN;
PROC PRINT; VAR workshop gender q1 q2 q3 q4; RUN;
PROC PRINT; VAR workshop--q4; RUN;
PROC PRINT; VAR workshop gender q1-q4; RUN;
PROC PRINT; VAR workshop gender q: ;
PROC PRINT; VAR _NUMERIC_;   RUN;
PROC PRINT; VAR _CHARACTER_; RUN;
PROC PRINT; VAR workshop-NUMERIC-q4;   RUN;
PROC PRINT; VAR workshop-CHARACTER-q4; RUN;

* Creating a data set from selected variables;
DATA myLib.myqs;
  SET myLib.mydata(KEEP=q1-q4);
  RUN;
```

7.13.2 SPSS Program to Select Variables

```
* Filename: SelectingVars.sps .

CD  C:\myRfolder .
GET FILE= mydata.sav .

LIST.
LIST VARIABLES=workshop,gender,q1,q2,q3,q4.
LIST VARIABLES=workshop TO q4.

* Creating a data set from selected variables.

SAVE OUTFILE= C:\myRfolder\myqs.sav  /KEEP=q1 TO q4.
```

7.13.3 R Program to Select Variables

```
# Filename: SelectingVars.R

# Uses many of the same methods as selecting observations.
setwd("c:/myRfolder")
load(file = "myData.RData")

# This refers to no particular variables,
# so all are printed.
print(mydata)


# ---Selecting Variables by Index Number---

# These also select all variables by default.
print( mydata[ ] )
print( mydata[ , ] )

# Select just the 3rd variable, q1.
print( mydata[ ,3] ) #Passes q3 as a vector.
print( mydata[3] )   #Passes q3 as a data frame.

# These all select the variables q1,q2,q3 and q4 by indices.
print( mydata[ c(3, 4, 5, 6) ] )
print( mydata[ 3:6 ] )

# These exclude variables q1,q2,q3,q4 by indices.
print( mydata[ -c(3, 4, 5, 6) ] )
print( mydata[ -(3:6) ] )

# Using indices in a numeric vector.
myQindices <- c(3, 4, 5, 6)
myQindices
print( mydata[myQindices] )
print( mydata[-myQindices] )

# This displays the indices for all variables.
print( data.frame( names(mydata) ) )

# Using ncol to find the last index.
print( mydata[ 1:ncol(mydata) ] )
print( mydata[ 3:ncol(mydata) ] )
```

```
# ---Selecting Variables by Column Name---

# Display all variable names.
names(mydata)

# Select one variable.
print( mydata["q1"] ) #Passes q1 as a data frame.
print( mydata[ ,"q1"] ) #Passes q1 as a vector.

# Selecting several.
print( mydata[ c("q1", "q2", "q3", "q4") ] )

# Save a list of variable names to use.
myQnames <- c("q1", "q2", "q3", "q4")
myQnames
print( mydata[myQnames] )

# Generate a list of variable names.
myQnames <- paste( "q", 1:4, sep = "")
myQnames
print( mydata[myQnames] )


# ---Selecting Variables Using Logic---

# Select q1 by entering TRUE/FALSE values.
print( mydata[ c(FALSE,FALSE,TRUE,FALSE,FALSE,FALSE) ] )

# Manually create a vector to get just q1.
print( mydata[ as.logical( c(0, 0, 1, 0, 0, 0) ) ] )

# Automatically create a logical vector to get just q1.
print( mydata[ names(mydata) == "q1" ] )

# Exclude q1 using NOT operator "!".
print( mydata[ !names(mydata) == "q1" ] )

# Use the OR operator, "|" to select q1 through q4,
# and store the resulting logical vector in myqs.
myQtf <- names(mydata) == "q1" |
         names(mydata) == "q2" |
         names(mydata) == "q3" |
         names(mydata) == "q4"
myQtf
print( mydata[myQtf] )
```

```
# Use the %in% operator to select q1 through q4.
myQtf <- names(mydata) %in% c("q1", "q2", "q3", "q4")
myQtf
print( mydata[myQtf] )


# ---Selecting Variables by String Search---

# Use grep to save the q variable indices.
myQindices <- grep("^q", names(mydata), value = FALSE)
myQindices
print( mydata[myQindices] )

# Use grep to save the q variable names (value = TRUE now).
myQnames <- grep("^q", names(mydata), value = TRUE)
myQnames
print( mydata[myQnames] )

# Use %in% to create a logical vector
# to select q variables.
myQtf <- names(mydata) %in% myQnames
myQtf
print( mydata[myQtf] )

# Repeat example above but searching for any
# variable name that begins with q, followed
# by one digit, followed by anything.
myQnames <- grep("^q[[:digit:]]\{1\}",
   names(mydata), value = TRUE)
myQnames
myQtf <- names(mydata) %in% myQnames
myQtf
print( mydata[myQtf] )

# Example of how glob2rx converts q* to ^q.
glob2rx("q*")


# ---Selecting Variables Using $ Notation---

print( mydata$q1 )
print( data.frame(mydata$q1, mydata$q2) )
```

```
# ---Selecting Variables by Simple Name---

# Using the "attach" function.
attach(mydata)
print(q1)
print( data.frame(q1, q2, q3, q4) )
detach(mydata)

# Using the "with" function.
with( mydata,
  summary( data.frame(q1, q2, q3, q4) )
)


# ---Selecting Variables with subset Function---

print( subset(mydata, select = q1:q4) )
print( subset(mydata,
  select = c(workshop, q1:q4)
) )


# ---Selecting Variables by List Subscript---

print( mydata[[3]] )


# ---Generating Indices A to Z from Two Variables---

myqA <- which( names(mydata) == "q1" )
myqA
myqZ <- which( names(mydata) == "q4" )
myqZ
print( mydata[myqA:myqZ] )


# ---Selecting Numeric or Character Variables---

is.numeric( mydata$workshop )
is.numeric( mydata$q1 )

# Find numeric variables
myNums <- sapply(mydata, is.numeric)
myNums
```

```
print( mydata[myNums] )

myA <- which( names(mydata) == "gender" )
myA
myZ <- which( names(mydata) == "q3" )
myZ

myRange <- 1:length(mydata) %in% myA:myZ
myRange

print( mydata[ myNums & myRange ] )

apply(mydata, 2, is.numeric)
apply(mydata, 2, class)

as.matrix(mydata)


# ---Creating a New Data Frame of Selected Variables---

myqs <- mydata[3:6]
myqs
myqs <- mydata[ c("q1", "q2", "q3", "q4") ]
myqs
myqs <- data.frame(mydata$q1, mydata$q2,
                   mydata$q3, mydata$q4)
myqs
myqs <- data.frame(q1 = mydata$q1, q2 = mydata$q2,
                   q3 = mydata$q3, q4 = mydata$q4)
myqs

attach(mydata)
myqs <- data.frame(q1, q2, q3, q4)
myqs
detach(mydata)

myqs <- subset(mydata, select = q1:q4)
myqs
```

8

Selecting Observations

It bears repeating that the approaches that R uses to select observations are, for the most part, the same as those discussed in the previous chapter for selecting variables. This chapter builds on that one, so if you have not read it recently, now would be a good time to do so.

Here I focus only on selecting observations. The amount of repetition between this chapter and the last may seem tedious, but, I have found from teaching that people learn much more easily when these topics are presented separately.

If you followed the last chapter easily, feel free to skip this one until you have problems using one of the approaches for selecting observations. The next chapter will cover the selection of variables and observations at the same time but will do so in much less detail.

8.1 Selecting Observations in SAS and SPSS

There are many ways to select observations in SAS and SPSS, and it is beyond our scope to discuss them all here. However, we will look at some approaches for comparison purposes. For both SAS and SPSS, if you do not select observations, they assume you want to analyze all of the data. So in SAS

```
PROC MEANS;
RUN;
```

will analyze all of the observations, and in SPSS

```
DESCRIPTIVES VARIABLES=ALL.
```

will also use all observations.

To select a subset of observations (e.g., the males), SAS uses the WHERE statement.

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_8, © Springer Science+Business Media, LLC 2011

```
PROC MEANS;
WHERE gender="m";
RUN;
```

It is also common to create a logical 0/1 value in the form

```
female = gender= f ;
```

which you could then apply with

```
PROC MEANS;
WHERE female;
RUN;
```

SPSS does the same selection using both the TEMPORARY and the SELECT IF commands:

```
TEMPORARY.
SELECT IF(gender EQ "m").
DESCRIPTIVES VARIABLES=ALL.
```

If we had not used the TEMPORARY command, the selection would have deleted the females from the data set. We would have had to open the data set again if we wanted to analyze both groups in a later step. R has no similar concept. Alternatively, we could create a variable that has a value of 1 for observations we want and zero otherwise. Using that variable on the FILTER command leaves a selection in place until a USE ALL brings the data back. As we will see, R uses a similar filtering approach.

```
COMPUTE male=(gender="m").
FILTER BY male.
DESCRIPTIVES VARIABLES=workshop TO q4.
* more stats could follow for males.
USE ALL.
```

8.2 Selecting All Observations

In R, if you perform an analysis without selecting any observations, the function will use all of the observations it can. That is how both SAS and SPSS work. For example, to get summary statistics on all observations (and all variables), we could use

```
summary(mydata)
```

The methods to select observations apply to all R functions that accept variables (vectors and so forth) as input. We will use the `summary` function so you will see the selection in the context of an analysis.

8.3 Selecting Observations by Index Number

Although it is as easy to use subscripting to select observations by index number, you need to be careful doing it. This is because sorting a data frame is something you do often, and sorting changes the index number of each row (if you save the sorted version, of course). Variables rarely change order, so this approach is much more widely used to select them. That said, let us dive in and see how R does it.

Since this chapter focuses on selecting observations, we will now discuss just the first subscript, the rows. Our data frame has eight observations or rows, which are automatically given index numbers, or indices, of 1, 2, 3, 4, 5, 6, 7, and 8. You can select observations by supplying one index number or a vector of indices. For example,

```
summary( mydata[5, ] )
```

selects all of the variables for only row 5. There is not much worth analyzing with that selection! Note that when selecting observations, the comma is *very* important, even though we request no columns in the example above. If you leave the comma out, R will assume that any index values it sees are *column* indices, and you will end up selecting *variables* instead of observations!

As long as you include the comma, this selection goes across columns of a data frame, so it must return a one-row data frame. A data frame can contain numeric, character, or factor variables. Only a data frame could store such a mixture. That is the opposite of selecting the fifth variable with `mydata[ ,5]` because that would select a vector. In many cases, this distinction might not matter, but in some cases it will. The difference will become clear as we work through the rest of the book.

To select more than one observation using indices, you must combine them into a numeric vector using the `c` function. Therefore, this will select rows 5 through 8, which happen to be the males:

```
summary( mydata[ c(5, 6, 7, 8),  ] )
```

You will see the `c` function used in many ways in R. Whenever R requires one object and you need to supply it several, it combines the several into one. In this case, the several index numbers become a single numeric vector. Again, take note of the comma that precedes the right square bracket. If we left that comma out, R would try to analyze variables 5 through 8 instead of observations 5 through 8! Since we have only six variables, that would generate an error message. However, if we had more variables, the analysis would run, giving us the wrong result with no error message. I added extra spaces in this example to help you notice the comma. You do not need additional spaces in R, but you can have as many as you like to enhance legibility.

The colon operator “:” can generate a numeric vector directly, so

```
summary( mydata[5:8, ] )
```

selects the same observations.

The colon operator is not just shorthand. Entering 1:N in an R program will cause it to generate the sequence, 1,2,3,. . . ,N.

If you use a negative sign on an index, you will exclude those observations. For example,

```
summary( mydata[ -c(1,2,3,4) , ] )
```

will exclude the first four records, three females and one with a gender of NA. R will then analyze the males.

Your index values must be either all positive or all negative. Otherwise, the result would be illogical. You cannot say "include only these observations" and "include all but these observations" at the same time.

The colon operator can abbreviate sequences of numbers, but you need to be careful with negative numbers. If you want to exclude rows 1 through 4, the following sequence will not work:

```
> -1:4

 [1] -1  0  1  2  3  4
```

This would, of course, generate an error because they must all have the same sign. Adding parentheses will clarify the situation, showing R that you want the minus sign to apply to just the set of numbers from +1 through +4 rather than –1 through +4:

```
> -(1:4)

 [1] -1 -2 -3 -4

 > summary( mydata[ -(1:4) , ] )
```

If you find yourself working with a set of observations repeatedly, you can easily save a vector of indices so you will not have to keep looking up index numbers. In this example, we are storing the indices for the males in myMindices (M for male). If I were not trying to make a point about indices, I would choose a simpler name like just "males."

```
myMindices <- c(5,6,7,8)
```

From now on, we can use that variable to analyze the males:

```
summary( mydata[myMindices, ] )
```

For a more realistic data set, typing all of the observation index numbers you need would be absurdly tedious and error prone. We will use logic to create that vector in Sect. 8.6. You can list indices individually or, for contiguous observations, use the colon operator. For a larger data set, you could use observations 1, 3, 5 through 20, 25, and 30 through 100 as follows:

```
mySubset <- c(1, 3, 5:20, 25, 30:100)
```

See Chap. 12, "Generating Data," for ways to generate other sequences of index numbers.

It is easy to have R list the index for each observation in a data frame. Simply create an index using the colon operator and append it to the front of the data frame.

```
> data.frame(myindex = 1:8, mydata)

  myindex workshop gender q1 q2 q3 q4
1       1        R      f  1  1  5  1
2       2      SAS      f  2  1  4  1
3       3        R      f  2  2  4  3
4       4      SAS   <NA>  3  1 NA  3
5       5        R      m  4  5  2  4
6       6      SAS      m  5  4  5  5
7       7        R      m  5  3  4  4
8       8      SAS      m  4  5  5  5
```

Note that the unlabeled column on the left contains the row names. In our case, the row names look like indices. However, the row names could have been descriptive strings like "Bob," so there is no guarantee of a relationship between row names and indices. Index values are dynamic, like the case numbers displayed in the SAS or SPSS data editors. When you sort or rearrange the data, they change. Row names, on the other hand, are fixed when you create the data frame. Sorting or rearranging the rows will not change row names.

You can use the `nrow` function to find the number of rows in a data frame. Therefore, another way to analyze all your observations is

```
summary( mydata[ 1:nrow(mydata) ,  ]  )
```

If you remember that the first male is the fifth record and you want to analyze all of the observations from there to the end, you can use

```
summary( mydata[ 5:nrow(mydata) ,  ]  )
```

8.4 Selecting Observations Using Random Sampling

Selecting random samples of observations in SAS and SPSS is done in two to three steps:

1. Create a variable whose values are uniformly random between zero and one.

2. Select observations whose values on that variable fall at or below the proportion you seek. If an approximate number of observations is sufficient, you are done.
3. If you seek an exact number of observations, you can assure that by sorting on the random variable and then choosing the first n you want. Alternatively, you might count each observation as you select it and stop selecting when your goal is met. SPSS will even write the latter steps out for you if you use the Data> Select Cases dialog box.

R uses an approach that takes advantage of the topic we just learned in the previous section: subscripting by index value. The sample function will select n values at random from any vector. If that vector holds the numbers 1, 2, 3. . . ,N, where N is the number of observations in our data set, then we end up sampling the index values for our rows. All that remains is to use those values to select the observations.

Let us do an example where we want to select 25% of our data, a massive data set of two whole records! The index values to sample are the values 1:8, or, more generally, 1:nrow(mydata). To ensure that you get the same selection as I do, I will use the set.seed function:

```
> set.seed(123)

> myindices <- sample( 1:nrow(mydata), 2 )

> myindices

[1] 3 6
```

The sample function call used just two arguments, the vector to sample and how many to get: 2. We see the two index values are 3 and 6. Let us now use them to select our sample from the main data set. I will put myindices in the row position and leave the column position empty so that I will select all the variables:

```
> mySample  <- mydata[myindices, ]

> print(mySample)

  workshop gender q1 q2 q3 q4
3        R      f  2  2  4  3
6      SAS      m  5  4  5  5
```

We see that our sample consists of one female who took the R workshop and a male who took the SAS one.

8.5 Selecting Observations by Row Name

SAS and SPSS data sets have variable names but not observation or case names. In R, data frames always name the observations and store those names in the *row names attribute*. When we read our data set from a text file, we told it that the first column would be our row names. The row.names function will display them:

```
row.names(mydata)
```

R will respond with

```
"1", "2", "3", "4", "5", "6", "7", "8"
```

The quotes show that R treats these as characters, not as numbers. If you do not provide an ID or name variable for R to use as row names, it will always create them in this form. Therefore, if we had not had an ID variable, we would have ended up in exactly the same state. I included an ID variable because it emphasizes the need to be able to track your data back to its most original source when checking for data entry errors. With such boring row names, there is little need to use them. indices are numerically more useful. So let us change the names; we will then have an example that makes more sense.

I will use common first names to keep the example easy to follow. First, let us create a new character vector of names:

```
> mynames <- c("Ann", "Cary", "Sue", "Carla",
               "Bob", "Scott", "Mike", "Rich")
```

Now we will write those names into the row names attribute of our data frame:

```
row.names(mydata) <- mynames
```

This is a very interesting command! It shows that the row.names function does not just show you the names, it provides access to the names attribute itself. Assigning mynames to that vector renames all of the rows! In Sect. 10.6, "Renaming Variables (and Observations)," we will see this again with several variations.

Let us see how this has changed our data frame.

```
> mydata

      workshop gender q1 q2 q3 q4
Ann          R      f  1  1  5  1
Cary       SAS      f  2  1  4  1
Sue          R      f  2  2  4  3
Carla      SAS   <NA>  3  1 NA  3
Bob          R      m  4  5  2  4
```

```
Scott      SAS     m  5  4  5  5
Mike         R     m  5  3  4  4
Rich       SAS     m  4  5  5  5
```

Now that we have some interesting names to work with, let us see what we can do with them. If we wanted to look at the data for “Ann,” we could use

```
mydata["Ann", ]
```

You might think that if we had several records per person, we could use row names to select all of the rows for any person. R, however, requires that row names be unique, which is a good idea.[1] You could always use an ID number that is unique for row names, then have the subjects' names on each record in their set and a counter like time 1, 2, 3, 4. We will look at just that structure in Sect. 10.17, “Reshaping Variables to Observations and Back.”

To select more than one row name, you must combine them into a single character vector using the `c` function. For example, we could analyze the females using

```
summary( mydata[ c("Ann","Cary","Sue","Carla"), ] )
```

With a more realistically sized data frame, we would probably want to save the list of names to a character vector that we could use repeatedly. Here, I use *F* to represent females and *names* to remind me of what is in the vector:

```
myFnames <- c("Ann","Cary","Sue","Carla")
```

Now we will analyze the females again using this vector:

```
summary( mydata[ myFnames, ] )
```

8.6 Selecting Observations Using Logic

You can select observations by using a logical vector of TRUE/FALSE values. You can enter one manually or create one by specifying a logical condition. Let us begin by entering one manually. For example, the following will print the first four rows of our data set:

```
> myRows <- c(TRUE, TRUE, TRUE, TRUE,
+   FALSE, FALSE, FALSE, FALSE)

> print( mydata[myRows, ]  )
```

[1] Recall that R does allow for duplicate variable names, although that is a bad idea.

```
  workshop gender q1 q2 q3 q4
1        R      f  1  1  5  1
2      SAS      f  2  1  4  1
3        R      f  2  2  4  3
4      SAS   <NA>  3  1 NA  3
```

In SAS or SPSS, the digits 1 and 0 can represent TRUE and FALSE, respectively. Let us see what happens when we try this in R.

```
> myBinary <- c(1, 1, 1, 1, 0, 0, 0, 0)

> print( mydata[myBinary, ] )

    workshop gender q1 q2 q3 q4
1          R      f  1  1  5  1
1.1        R      f  1  1  5  1
1.2        R      f  1  1  5  1
1.3        R      f  1  1  5  1
```

What happened? Remember that putting a 1 in for the row subscript asks for row 1. So our request asked for row 1 four consecutive times and then asked for row 0 four times. Index values of zero are ignored. We can get around this problem by using the `as.logical` function:

```
> myRows <- as.logical(myBinary)
```

Now, myRows contains the same TRUE/FALSE values it had in the previous example and would work fine.

While the above examples make it clear how R selects observations using logic, they are not very realistic. Hundreds of records would require an absurd amount of typing. Rather than typing such logical vectors, you can generate them with a logical statement such as

```
> mydata$gender == "f"

 [1]  TRUE  TRUE  TRUE    NA FALSE FALSE FALSE FALSE
```

The "==" operator compares every value of a vector, like gender, to a value, like "f", and returns a logical vector of TRUE/FALSE values. These logical conditions can be as complex as you like, including all of the usual logical conditions. See Table 10.3, "Logical operators," for details.

The length of the resulting logical vector will match the number of observations in our data frame. Therefore, we could store it in our data frame as a new variable. That is essentially the same as the SPSS filter variable approach.

Unfortunately, we see that the fourth logical value is NA. That is because the fourth observation has a missing value for gender. Up until this point, we have been mirroring Chap. 7, "Selecting Variables." There, logical comparisons of variable names did not have a problem with missing values. Now, however, we must take a different approach. First, let us look at what would happen if we continued down this track.

```
> print( mydata[ mydata$gender == "f", ] )

   workshop gender q1 q2 q3 q4
1         R      f  1  1  5  1
2       SAS      f  2  1  4  1
3         R      f  2  2  4  3
NA     <NA>   <NA> NA NA NA NA
```

What happened to the fourth observation? It had missing values only for gender and q3. Now *all* of the values for that observation are missing. R has noticed that we were selecting rows based on only gender. Not knowing what we would do with the selection, it had to make all of the other values missing, too. Why? Because we might have been wanting to correlate q1 and q4. Those two had no missing values in the original data frame. If we want to correlate them only for the females, even their values must be set to missing.

We could select observations using this logic and then count on R's other functions to remove the bad observations as they would any others with missing values. However, there is little point in storing them. Their presence could also affect future counts of missing values for other analyses, perhaps when females are recombined with males.

Luckily, there is an easy way around this problem. The `which` function gets the index values for the TRUE values of a logical vector. Let us see what it does.

```
> which( mydata$gender == "f" )

 [1] 1 2 3
```

It has ignored both the NA value and the FALSE values to show us that only the first three values of our logical statement were TRUE. We can save these index values in myFemales.

```
> myFemales <- which( mydata$gender == "f" )

> myFemales
 [1] 1 2 3
```

We can then analyze just the females with the following function call:

```
summary( mydata[ myFemales , ] )
```

Negative index values exclude those rows, so we could analyze the non-females (males and missing) with the following function call:

```
summary( mydata[-myFemales , ] )
```

We could, of course, get males and exclude missing the same way we got the females.

We can select observations using logic that is more complicated. For example, we can use the AND operator "&" to analyze subjects who are both male and who "strongly agree" that the workshop they took was useful. Compound selections like this are much easier when you do it in two steps. First, create the logical vector and store it; then use that vector to do your selection.

```
> HappyMales <- which(mydata$gender == "m"
+   & mydata$q4 == 5)

> HappyMales
 [1] 6 8
```

So we could analyze these observations with

```
summary( mydata[HappyMales , ] )
```

Whenever you are making comparisons to many values, you can use the `%in%` operator. Let us select observations who have taken the R or SAS workshop. With just two target workshops, you could use a simple `workshop == "R" | workshop == "SPSS"`, but the longer the target list, the happier you will be to save all of the repetitive typing.

```
> myRsas <-
+   which( mydata$workshop %in% c("R","SAS") )

> myRsas
 [1] 1 3 5 7
```

Then we can get summary statistics on those observations using

```
summary( mydata[myRsas, ] )
```

The various methods we described in Chap. 7, "Selecting Variables," make a big difference in how complicated the logical commands to select observations appear. Here are several different ways to analyze just the females:

```
myFemales <- which( mydata$gender == "f")

myFemales <- which( mydata[2] == "f")

myFemales <- which( mydata["gender"] == "f")

with(mydata,
  myFemales <- which(gender == "f")
)

attach(mydata)
  myFemales <- which(gender == "f")
detach(mydata)
```

You could then use any of these to analyze the data using

```
summary( mydata[ myFemales, ] )
```

You can easily convert a logical vector into an index vector that will select the same observations. For details, see Sect. 10.19, “Converting Data Structures.”

8.7 Selecting Observations by String Search

If you have character variables, or useful row names, you can select observations by searching their values for strings of text. This approach uses the methods of selection by indices, row names, and logic discussed earlier, so make sure you have mastered them before trying these.

R searches variable names for patterns using the `grep` function. We previously replaced our original row names, “1,” “2,” etc., with more interesting ones, “Ann”, “Cary,” and so forth. Now we will use the `grep` function to search for row names that begin with the letter “C”:

```
myCindices <- grep("^C", row.names(mydata), value = FALSE)
```

This `grep` function call uses three arguments.

1. The first is the command string, or regular expression, “`^C`,” which means “find strings that begin with a capital letter C.” The symbol “`^`” represents “begins with.” You can use any regular expression here, allowing you to search for a wide range of patterns in variable names. We will discuss using wildcard patterns later.
2. The second argument is the character vector that you wish to search. In our case, we want to search the row names of mydata, so I call the `row.names` function here.
3. The `value` argument tells it what to store when it finds a match. The goal of `grep` in any computer language or operating system is to find patterns. A value of TRUE here will tell it to save the row names that match the pattern we seek. However, in R, indices are more fundamental than names, which are optional, so the default setting is FALSE to save indices instead. We could leave it off in this particular case, but we will use it in the other way in the next example, so we will list it here for educational purposes. The contents of myCindices will be 2 and 4 because Cary and Carla are the second and fourth observations, respectively. If we wanted to save this

variable, it does not match the eight values of our other variables, so we cannot store it in our data frame. We would instead just store it in the workspace as a vector outside our data frame.

To analyze those observations, we can then use

```
summary( mydata[myCindices , ] )
```

Now let us do the same thing but have `grep` save the actual variable names. All we have to do is change to `value = TRUE`:

```
myCnames <- grep("^C", row.names(mydata), value = TRUE)
```

The character vector myCnames now contains the row names "Cary" and "Carla," and we can analyze those observations with

```
summary(  mydata[myCnames ,  ]  )
```

Finally, let us do a similar search using the `%in%` function. In R, it works just like the IN operator in SAS. It finds matches between two sets of values. We will use it to find which of our row names appears in this set of target names:

```
myTargetNames <- ("Carla","Caroline","Cary","Cathy","Cynthia")

myMatches <- row.names(mydata) %in% myTargetNames
```

The result will be a logical set of TRUE/FALSE values that indicate that the names that match are in the second and fourth positions:

```
FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE
```

Now we can use the myMatches vector in any analysis like `summary`:

```
summary( mydata[myMatches, ] )
```

You may be more familiar with the search patterns using wildcards in Microsoft Windows. They use "*" to represent any number of characters and "?" to represent any single character. So the wildcard version of any variable name beginning with the letter "C" is "C*." Computer programmers call this type of symbol a "glob," short for *glob*al. R lets you convert globs to *r*egular e*x*pressions with the `glob2rx` function. Therefore, we could do our first `grep` again in the form

```
myCindices <- grep( glob2rx("C*"),
  row.names(mydata), value = FALSE)
```

8.8 Selecting Observations with the subset Function

You can select observations using the subset function. You simply list your logical condition under the subset argument, as in

```
subset(mydata, subset = gender == "f")
```

Note that an equal sign follows the subset argument because that is what R uses to set argument values. The gender == "f" comparison is still done using "==" because that is the symbol R uses for logical comparisons. You can use subset to analyze your selection using the form

```
summary(
  subset(mydata, subset = gender == "f")
)
```

The following selection, in which we select the males who were happy with their workshop, is slightly more complicated . In R, the logic is a single object, a logical vector, regardless of its complexity.

```
summary(
  subset( mydata, subset = gender == "m" & q4 == 5 )
)
```

Since the first argument to the subset function is the data frame to use, you do not have to write out the longer forms of names like mydata$q1 or mydata$gender. Also, its logical selections automatically exclude cases for which the logic would be missing. So it acts like the which function that is built into every selection. That is a very helpful function!

8.9 Generating Indices A to Z from Two Row Names

This method uses several of the approaches from the previous examples. We have seen how the colon operator can help us analyze the males, who are observations 5 through 8, using the form

```
summary( mydata[5:8, ] )
```

However, you had to know the index numbers, and digging through lists of observation numbers can be tedious work. However, we can use the row.names function and the which function to get R to find the index values we need. The function call

```
row.names(mydata) == "Bob"
```

will generate the logical vector

```
FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE
```

because Bob is the fifth observation. The `which` function will tell us the index values of any TRUE values in a logical vector, so

```
which(FALSE, FALSE, FALSE, FALSE,
  TRUE, FALSE, FALSE, FALSE)
```

will yield a value of 5. Putting these ideas together, we can find the index number of the first observation we want, store it in myMaleA, then find the last observation, store it in myMaleZ, and then use them with the colon operator to analyze our data from A to Z:

```
myMaleA <- which( names(mydata) == "Bob" )

myMaleZ <- which( names(mydata) == "Rich" )

summary( mydata[ myMaleA:myMaleZ , ] )
```

8.10 Variable Selection Methods with No Counterpart for Selecting Observations

As we have seen, the methods that R uses to select variables and observations are almost identical. However, there are several techniques for selecting variables that have no equivalent in selecting observations:

The $ prefix form (e.g., mydata$gender),
The `attach` function's approach to short variable names,
The `with` function's approach to short variable names,
The use of formulas.
The list form of subscripting that uses double brackets (e.g., mydata[[2]]),
Using variable types to select only numeric variables, character variables, or factors.

We also had one method of selecting observations, random sampling, that we used to select observations but not variables. That would be a most unusual approach to selecting variables, but one that might be useful in an area such as genetic algorithms.

8.11 Saving Selected Observations to a New Data Frame

You can create a new data frame that is a subset of your original one by using any of the methods for selecting observations. You simply assign the data to a new data frame. The examples below all select the males and assign them to the myMales data frame:

```
myMales <- mydata[5:8, ]

myMales <- mydata[ which(mydata$gender == "m") , ]

myMales <- subset( mydata, subset = gender == "m" )
```

8.12 Example Programs for Selecting Observations

The SAS and SPSS programs in this section demonstrate standard ways to select observations in those packages, and they match each other. The R program uses different methods, especially subscripting, and is much more detailed.

8.12.1 SAS Program to Select Observations

```
* Filename: SelectingObs.sas ;

LIBNAME myLib  C:\myRfolder ;

* Ways to Select Males and Females;
PROC PRINT DATA=myLib.mydata;
  WHERE gender="m";
  RUN;

PROC PRINT DATA=myLib.mydata;;
  WHERE gender="m" & q4=5;

DATA myLib.males;
  SET myLib.mydata;
  WHERE gender="m";
  RUN;
PROC PRINT; RUN;

DATA myLib.females;
  SET myLib.mydata;
  WHERE gender="f";
  RUN;
PROC PRINT; RUN;

* Random Sampling;
DATA  myLib.sample;
  SET myLib.mydata;
  WHERE UNIFORM(123) <= 0.25;
  RUN;
PROC PRINT; RUN;
```

8.12.2 SPSS Program to Select Observations

Note that the UNIFORM function in SPSS is quite different from that of SAS. Its only parameter is its highest value (usually 1), not the random generator seed.

```
* Filename: SelectingObs.sps .

CD  c:\myRfolder .
GET FILE= mydata.sav .

* Ways to Select Males and Females.
COMPUTE   male=(gender="m").
COMPUTE female=(gender="f").

FILTER BY male.
LIST.
* analyses of males could follow here.

FILTER BY female.
LIST.
* analyses of females could follow here.

USE ALL.

DO IF male.
XSAVE OUTFILE= males.sav .
ELSE IF female.
XSAVE OUTFILE= females.sav .
END IF.

* Selecting a Random Sample.
SET SEED=123.
DO IF uniform(1) LE 0.25.
XSAVE OUTFILE= sample.sav .
END IF.
LIST.
```

8.12.3 R Program to Select Observations

Throughout this chapter we have used the `summary` function to demonstrate how a complete analysis request would look. Here we will instead use the `print` function to make it easier to see the result of each selection when you run the programs. Even though

```
mydata[5:8, ]
```

is equivalent to

```
print( mydata[5:8, ] )
```

because `print` is the default function, we will use the longer form because it is more representative of its look with most functions. As you learn R, you will quickly opt for the shorter approach when you only want to print data.

```
# Filename: SelectingObs.R

setwd("c:/myRfolder")
load(file = "myWorkspace.RData")
print(mydata)


# ---Selecting Observations by Index---

# Print all rows.
print( mydata[ ] )
print( mydata[ , ] )
print( mydata[1:8, ] )

# Just observation 5.
print( mydata[5 , ] )

# Just the males:
print( mydata[ c(5, 6, 7, 8) , ] )
print( mydata[ 5:8, ] )

# Excluding the females with minus sign.
print( mydata[ -c(1, 2, 3, 4), ] )
print( mydata[ -(1:4), ] )

# Saving the Male (M) indices for reuse.
myMindices <- c(5, 6, 7, 8)
summary( mydata[myMindices, ] )

# Print a list of index numbers for each observation.
data.frame(myindex = 1:8, mydata)

# Select data using length as the end.
print( mydata[ 1:nrow(mydata),  ]  )
print( mydata[ 5:nrow(mydata),  ]  )


# ---Selecting Observations by Row Name---
# Display row names.
row.names(mydata)

# Select rows by their row name.
print( mydata[ c("1", "2", "3", "4"), ] )

# Assign more interesting names.
mynames <- c("Ann", "Cary", "Sue", "Carla",
             "Bob", "Scott", "Mike", "Rich")
print(mynames)

# Store the new names in mydata.
row.names(mydata) <- mynames
print(mydata)

# Print Ann s data.
print( mydata["Ann" , ] )
mydata["Ann" , ]

# Select the females by row name.
print( mydata[ c("Ann", "Cary", "Sue", "Carla"), ] )

# Save names of females to a character vector.
myFnames <- c("Ann", "Cary", "Sue", "Carla")
print(myFnames)

# Use character vector to select females.
print( mydata[ myFnames, ] )


# ---Selecting Observations Using Logic---

#Selecting first four rows using TRUE/FALSE.
myRows <- c(TRUE, TRUE, TRUE, TRUE,
  FALSE, FALSE, FALSE, FALSE)
print( mydata[myRows, ]  )

# Selecting first four rows using 1s and 0s.
myBinary <- c(1, 1, 1, 1, 0, 0, 0, 0)
print( mydata[myBinary, ] )
myRows <- as.logical(myBinary)
print( mydata[ myRows, ] )

# Use a logical comparison to select the females.
```

```
mydata$gender == "f"
print( mydata[ mydata$gender == "f", ] )
which( mydata$gender == "f" )
print( mydata[ which(mydata$gender == "f") , ] )

# Select females again, this time using a saved vector.
myFemales <- which( mydata$gender == "f" )
print(myFemales)
print( mydata[ myFemales , ] )

# Excluding the females using the "!" NOT symbol.
print( mydata[-myFemales , ] )

# Select the happy males.
HappyMales <- which(mydata$gender == "m"
  & mydata$q4 == 5)
print(HappyMales)
print( mydata[HappyMales , ] )

# Selecting observations using %in%.
myRsas <-
  which( mydata$workshop %in% c("R", "SAS") )
print(myRsas)
print( mydata[myRsas , ] )

# Equivalent selections using different
# ways to refer to the variables.

print( subset(mydata, gender ==  f ) )

attach(mydata)
  print(  mydata[ which(gender == "f") , ] )
detach(mydata)

with(mydata,
  print ( mydata[ which(gender == "f"), ] )
)

print( mydata[ which(mydata["gender"] == "f") , ] )

print( mydata[ which(mydata$gender == "f") , ] )


# ---Selecting Observations by String Search---
# Search for row names that begin with "C".
myCindices <- grep("^C", row.names(mydata), value = FALSE)
print( mydata[myCindices , ] )

# Again, using wildcards.
myCindices <- grep( glob2rx("C*") ,
  row.names(mydata), value = FALSE)
print( mydata[myCindices , ] )


# ---Selecting Observations by subset Function---

subset(mydata, subset=gender == "f")

summary(
  subset( mydata, subset = gender == "m" & q4 == 5 )
)


# ---Generating indices A to Z from Two Row Names---

myMaleA <- which( row.names(mydata) == "Bob" )
print(myMaleA)

myMaleZ <- which( row.names(mydata) == "Rich" )
print(myMaleZ)
print( mydata[myMaleA:myMaleZ , ] )


# ---Creating a New Data Frame of Selected Observations---

# Creating a new data frame of only males (all equivalent).
myMales <- mydata[5:8, ]
print(myMales)

myMales <- mydata[ which( mydata$gender == "m" ) , ]
print(myMales)

myMales <- subset( mydata, subset = gender == "m" )
print(myMales)

# Creating a new data frame of only females (all equivalent).
myFemales <- mydata[1:3, ]
print(myFemales)
```

```
myFemales <- mydata[ which( mydata$gender == "f" ) , ]
print(myFemales)

myFemales <- subset( mydata, subset = gender == "f" )
print(myFemales)
```

9

Selecting Variables and Observations

In SAS and SPSS, variable selection is done using a very simple yet flexible set of commands using variable names, and the selection of observations is done using logic. Combining the two approaches is quite simple. For example, selecting the variables workshop and q1 to q4 for the males only is done in SAS with

```
PROC PRINT;
  VAR workshop q1-q4;
  WHERE gender="m";
```

SPSS uses a very similar approach:

```
TEMPORARY.
SELECT IF (gender EQ "m").
LIST workshop q1 TO q4.
```

In the previous two chapters, we focused on selecting variables and observations separately, and we examined a very wide range of ways to do both. Different books and help files use various approaches, so it is important to know the range of options to perform these basic tasks in R. However, you can still use the approach that is already most familiar to you: using names to select variables and logic to select observations.

As an example, we will use the various methods to select the variables workshop and q1 to q4 for only the males.

The explanations in this chapter are much sparser. If you need clarification, see the detailed discussions of each approach in the previous two chapters.

9.1 The `subset` Function

Although you can use any of the methods introduced in the previous two chapters to select both variables and observations, variables are usually chosen

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_9, © Springer Science+Business Media, LLC 2011

by name and observations by logic. The `subset` function lets you use that combination easily.

When selecting variables, `subset` allows you to use the colon operator on lists of contiguous variables, like `gender:q4`. Variable selections that are more complex than a single variable or two contiguous variables separated by a colon must be combined with the `c` function.

When selecting observations, you perform logic like `gender == "m"` without having to use `which(gender == "m")` to get rid of the observations that have missing values for gender. The logic can be as complex as you like, so we can select the males who are happy with their workshop using `gender == "m" & q4 == 5`. Note that the result of a logical condition is always a single logical vector, so you never need the `c` function for logic. See Table 10.3, "Logical Operators," for details.

We can perform our selection by nesting the `subset` function directly within other functions:

```
summary(
  subset(mydata,
    subset = gender == "m",
    select = c(workshop, q1:q4) )
)
```

Since R allows you to skip the names of arguments as long as you have them in proper order, you often see `subset` used in the form

```
summary(
  subset(mydata, gender == "m",
    c(workshop, q1:q4) )
)
```

If you plan to use a subset like this repeatedly, it would make more sense to save the subset in a new data frame. Here we will add the `print` function just to make the point that selection is done once and then used repeatedly with different functions. Here I am using the name myMalesWQ to represent the males with workshop and the q variables.

```
myMalesWQ <- subset(mydata,
    subset = gender == "m",
    select = c(workshop,q1:q4)
)

print(myMalesWQ)
summary(myMalesWQ)
```

Performing the task in two steps like that often makes the code easier to read and less error prone.

9.2 Subscripting with Logical Selections and Variable Names

Another very useful approach is to use subscripting with logic to select observations and names to select variables. For example:

```
summary(
  mydata[ which(gender == "m") ,
          c("workshop","q1","q2","q3","q4") ]
)
```

This is very similar to what we did with the `subset` function, but we cannot use the form q1:q4 to choose contiguous variables. That shortcut works only with `subset`. So if you had many variables, you could instead use the shortcut described in Sect. 7.11, "Generating indices A to Z from Two Variable Names."

We could make our example more legible by defining the row and column indices in a separate step:

```
myMales <- which(gender == "m")
myVars  <- c("workshop", "q1", "q2", "q3", "q4")
```

Since the q variables make up most of the list and we have seen how to paste the letter q onto the numeric list of 1:4, we can make the same variable list using

```
myVars <- c("workshop", paste(q, 1:4, sep = "") )
```

I used the `c` function to combine just workshop with the results of the `paste` function, q1, q2, etc. Regardless of how you choose to create myVars, you can then make the selection with:

```
summary( mydata[ myMales, myVars ] )
```

This has the added benefit of allowing us to analyze just the males, for all variables (we are not selecting any specifically) with

```
summary( mydata[ myMales, ] )
```

We can also analyze males and females (by *not* choosing only males) for just myVars:

```
summary( mydata[ , myVars ] )
```

If we did not need that kind of flexibility and we planned to use this subset repeatedly, we would save it to a data frame:

```
myMalesWQ <- mydata[ myMales, myVars ]

summary(myMalesWQ)
```

9.3 Using Names to Select Both Observations and Variables

The above two approaches usually make the most sense. You usually know variable names and the logic you need to make your selection. However, for completeness' sake, we will continue on with additional combinations, but if you feel you understood the previous two chapters and the examples above, feel free to skip these examples and go to Sect. 9.6, "Saving and Loading Subsets."

Since the males have character row names of "5" through "8," we could use both row names and column with

```
summary( mydata[
  c("5","6","7","8"),
  c("workshop","q1","q2","q3","q4")
] )
```

This is an odd approach for selecting rows. We do not often bother to learn such meaningless row names. If we had row names that made more sense, like "Ann," "Bob," "Carla,"..., this approach would make more sense. However, we can at least be assured that the row names will not be affected by the addition of new observations or by sorting. Such manipulations do not change row names as they do numeric index values for rows.

If you plan on using these character index vectors often or if you have many values to specify, it is helpful to store them separately. This also helps document your program, since a name like myMales will remind you, or your colleagues, what you were selecting.

```
myMales <- c("5","6","7","8")

myVars  <- c("workshop","q1","q2","q3","q4")
```

Now we can repeat the *exact* same examples that we used in the section immediately above. Once you have a vector of index values, it does not matter if they are character names, numeric indices, or logical values.

Here we analyze our chosen observations and variables:

```
summary( mydata[ myMales, myVars] )
```

Here we analyze only the males, but include all variables:

```
summary( mydata[ myMales, ] )
```

Here we select all of the observations but analyze only our chosen variables:

```
summary( mydata[ , myVars] )
```

9.4 Using Numeric Index Values to Select Both Observations and Variables

The males have numeric index values of 5 through 8, and we want the first variable and the last four, so we can use numeric index vectors to choose them as in either of these two equivalent approaches:

```
summary( mydata[ c(5, 6, 7, 8), c(1, 3, 4, 5, 6) ] )

summary( mydata[ 5:8, c(1, 3:6) ] )
```

This selection is impossible to interpret without a thorough knowledge of the data frame. When you are hard at work on an analysis, you may well recall these values. However, such knowledge fades fast, so you would do well to add comments to your programs reminding yourself what these values select. Adding new variables or observations to the beginning of the data frame, or sorting it, would change these index values. This is a risky approach!

As we discussed in the last section, we can save the numeric index vectors for repeated use.

```
myMales <- c(5, 6, 7, 8)

myVars  <- c(1, 3:6)
```

Again, we can repeat the *exact* same examples that we used in the sections above. Once you have a vector of index values, it does not matter if they are character names or numeric indices.

Here we analyze our chosen observations and variables:

```
summary( mydata[ myMales, myVars ] )
```

Here we analyze only the males but include all variables:

```
summary( mydata[ myMales, ] )
```

Here we select all of the observations but analyze only our chosen variables:

```
summary( mydata[ , myVars ] )
```

9.5 Using Logic to Select Both Observations and Variables

Selecting observations with logic makes perfect sense, but selecting variables using logic is rarely worth the effort. Here is how we would use this combination for our example:

```
summary(
  mydata[which(gender == "m"),
    names(mydata) %in% c("workshop", "q1", "q2", "q3", "q4") ]
)
```

Let us reconsider using variable names directly. For this example, it is clearly simpler:

```
summary(
  mydata[ which(gender == "m") ,
    c("workshop", "q1", "q2", "q3", "q4") ]
)
```

However, once we save these values, we use them with no more work than earlier.

```
myMales <- which(gender == "m")

myVars  <- names(mydata) %in%
  c("workshop","q1","q2","q3","q4")
```

Here we analyze our chosen observations and variables:

```
summary( mydata[ myMales, myVars ] )
```

Here we analyze only the males but include all variables:

```
summary( mydata[ myMales, ] )
```

Here we select all of the observations but analyze only our chosen variables:

```
summary( mydata[ , myVars] )
```

9.6 Saving and Loading Subsets

Every method you use to create a subset results in a temporary copy that exists only in your workspace. To use it in future R sessions, you need to write it out to your computer's hard drive using the save or save.image functions. The more descriptive a name you give it, the better.

```
myMalesWQ <- subset(mydata,
    subset = gender == "m",
    select = c(workshop,q1:q4)
)
```

If your files are not too large, you can save your original data and your subset with

```
save(mydata, myMalesWQ, file = "mydata.RData")
```

The next time you start R, you can load both data frames with

```
load("mydata.RData")
```

If you are working with large files, you might save only the subset.

```
save(myMalesWQ, file = "myMalesWQ.RData")
```

Now when you start R, you can load and work with just the subset to save space.

```
load("myMalesWQ.RData")

summary(myMalesWQ)
```

9.7 Example Programs for Selecting Variables and Observations

9.7.1 SAS Program for Selecting Variables and Observations

```
* Filename: SelectingVarsAndObs.sas;

LIBNAME myLib  C:\myRfolder ;
OPTIONS _LAST_=myLib.mydata;

PROC PRINT; VAR workshop q1 q2 q3 q4;
WHERE gender="m";
RUN;

* Creating a data set from selected variables;
DATA myLib.myMalesWQ;
  SET myLib.mydata;
  WHERE gender="m";
  KEEP workshop q1-q4;
  RUN;

  PROC PRINT DATA=myLib.myMalesWQ; RUN;
```

9.7.2 SPSS Program for Selecting Variables and Observations

```
* Filename: SelectVarsAndObs.sps.

CD  c:\myRfolder .
GET FILE= mydata.sav .
```

```
SELECT IF (gender EQ "m").
LIST workshop q1 TO q4.

SAVE OUTFILE= myMalesWQ.sav .
EXECUTE.
```

9.7.3 R Program for Selecting Variables and Observations

```
# Filename: SelectingVarsAndObs.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
attach(mydata)
print(mydata)

# ---The subset Function---

print(
  subset(mydata,
    subset = gender == "m",
    select = c(workshop, q1:q4) )
)

myMalesWQ <- subset(mydata,
    subset = gender == "m",
    select = c(workshop, q1:q4)
)

print(myMalesWQ)
summary(myMalesWQ)

# ---Logic for Obs, Names for Vars---

print(
  mydata[ which(gender == "m") ,
          c("workshop", "q1", "q2", "q3", "q4") ]
)

myMales <- which(gender == "m")
myVars  <- c("workshop", "q1", "q2", "q3", "q4")
myVars
myVars <- c("workshop", paste(q, 1:4, sep="") )
myVars

print( mydata[myMales, myVars] )
print( mydata[myMales, ] )
print( mydata[ , myVars] )

myMalesWQ <- mydata[myMales, myVars]
print(myMalesWQ)

# ---Row and Variable Names---

print( mydata[
  c("5", "6", "7", "8"),
  c("workshop", "q1", "q2", "q3", "q4")
] )

myMales <- c("5", "6", "7", "8")
myVars  <- c("workshop", "q1", "q2", "q3", "q4")

print( mydata[myMales, myVars] )
print( mydata[myMales, ] )
print( mydata[ , myVars] )

# ---Numeric Index Vectors---

print( mydata[ c(5, 6, 7, 8), c(1, 3, 4, 5, 6) ] )
print( mydata[ 5:8, c(1, 3:6) ] )

myMales <- c(5,6,7,8)
myVars  <- c(1,3:6)

print( mydata[myMales, myVars] )
print( mydata[myMales, ] )
print( mydata[ , myVars] )

# ---Saving and Loading Subsets---

myMalesWQ <- subset(mydata,
    subset = gender == "m",
    select = c(workshop,q1:q4)
)

save(mydata, myMalesWQ, file = "myBoth.RData")
load("myBoth.RData")

save(myMalesWQ, file = "myMalesWQ.RData")
load("myMalesWQ.RData")
print(myMalesWQ)
```

10

Data Management

An old rule of thumb says that 80% of your data analysis time is spent transforming, reshaping, merging, and otherwise managing your data. SAS and SPSS have a reputation of being more flexible than R for data management. However, as you will see in this chapter, R can do everything SAS and SPSS can do on these important tasks.

10.1 Transforming Variables

Unlike SAS, R has no separation of phases for data modification (data step) and analysis (proc step). It is more like SPSS where as long as you have data read in, you can modify it using COMPUTE commands, or, via the Python plug-in, the SPSSINC TRANSFORM extension command. Anything that you have read into or created in your R workspace you can modify at any time.

R performs transformations such as adding or subtracting variables on the whole variable at once, as do SAS and SPSS. It calls that vector arithmetic. R has loops, but you do not need them for this type of manipulation. R can nest one function call within another within any other. This applies to transformations as well. For example, taking the logarithm of our q4 variable and then getting summary statistics on it, you have a choice of a two-step process like

```
mydata$q4Log <- log(mydata$q4)
summary( mydata$q4Log )
```

or you could simply nest the log function: within the `summary` function

```
summary( log(mydata$q4) )
```

If you planned to do several things with the transformed variable, saving it under a new name would lead to less typing and quicker execution. Table 10.1

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_10, © Springer Science+Business Media, LLC 2011

Table 10.1. Mathematical operators and functions

	R	SAS	SPSS
Addition	`x + y`	`x + y`	`x + y`
Antilog, base 10	`10^x`	`10**x`	`10**x`
Antilog, base 2	`2^x`	`2**x`	`2**x`
Antilog, natural	`exp(x)`	`exp(x)`	`exp(x)`
Division	`x / y`	`x / y`	`x / y`
Exponentiation	`x^2`	`x**2`	`x**2`
Logarithm, base 10	`log10(x)`	`log10(x)`	`lg10(x)`
Logarithm, base 2	`log2(x)`	`log2(x)`	`lg10(x)*3.3212`
Logarithm, natural	`log(x)`	`log(x)`	`ln(x)`
Multiplication	`x * y`	`x * y`	`x * y`
Round off	`round(x)`	`round(x)`	`rnd(x)`
Square root	`sqrt(x)`	`sqrt(x)`	`sqrt(x)`
Subtraction	`x - y`	`x - y`	`x - y`

shows basic transformations in both packages. In Chap. 7, "Selecting Variables," we chose variables using various methods: by index, by column name, by logical vector, using the style mydata$myvar, by simply using the variable name after you have attached a data frame, and by using the `subset` or `with` functions.

Here are several examples that perform the same transformation using different variable selection approaches. The `within` function is a variation of the `with` function that has some advantages for variable creation that are beyond our scope. We have seen that R has a `mean` function, but we will calculate the mean the long way just for demonstration purposes.

```
mydata$meanQ <- (mydata$q1 + mydata$q2
               + mydata$q3 + mydata$q4) / 4

mydata[,"meanQ"] <- (mydata[ ,"q1"] + mydata[ ,"q2"]
                   + mydata[ ,"q3"] + mydata[ ,"q4"] ) / 4

within( mydata,
  meanQ <- (q1 + q2 + q3 + q4) / 4
)
```

Another way to use the shorter names is with the `transform` function. It is similar to attaching a data frame, performing as many transformations as you like using short variable names, and then detaching the data (we do that example next). It looks like this:

```
mydata <- transform(mydata, meanQ=(q1 + q2 + q3 + q4) / 4)
```

It may seem strange to use the "`=`" now in an equation instead of "`<-`," but in this form, meanQ is the name of an argument, and arguments are always specified using "=." If you have many transformations, it is easier to read them on separate lines:

```
mydata <- transform(mydata,
  score1=(q1 + q2) / 2,
  score2=(q3 + q4) / 2
)
```

Before beginning, the `transform` function reads the data, so if you want to continue to transform variables you just created, you must do it in a second call to that function. For example, to get the means of score1 and score2, you cannot do the following:

```
mydata <- transform(mydata,
  score1=(q1 + q2) / 2,
  score2=(q3 + q4) / 2,
  meanscore=score1 + score2 / 2  # Does not work!
)
```

It will not know what score1 and score2 are for the creation of meanscore. You can do that in two steps:

```
mydata <- transform(mydata,
  score1=(q1 + q2) / 2,
  score2=(q3 + q4) / 2
)
mydata <- transform(mydata,
  meanscore=score1 + score2 / 2  # This works.
)
```

Wickham's `plyr` package [73] has a `mutate` function that is very similar to `transform`, but it *can* use variables that it just created.

You can create a new variable using the index method, but it requires a bit of extra work. Let us load the data set again since we already have a variable named meanQ in the current one:

```
load(file = "mydata.RData")
```

Now we will add a variable at index position 7 (we currently have six variables). Using the index approach, it is easier to initialize a new variable by binding a new variable to mydata. Otherwise, R will automatically give it a column name of V7 that we would want to rename later. We used the column bind function, `cbind`, to create mymatrix earlier. Here we will use it to name the new variable, meanQ, initialize it to zero, and then bind it to mydata:

```
mydata <- data.frame( cbind( mydata, meanQ = 0.) )
```

Now we can add the values to column 7.

```
mydata[7] <- (mydata$q1 + mydata$q2 +
              mydata$q3 + mydata$q4)/4
```

Let us examine what happens when you create variables using the `attach` function. You can think of the `attach` function as creating a temporary copy of the data frame, so changing that is worthless. See Sect.13.3 for details. However, you can safely use the attach method to simplify naming variables on the *right side* of the equation. This is a safe example because the variable being created is clearly going into our data frame since we are using the long dataframe$varname style:

```
attach(mydata)

mydata$meanQ <- (q1 + q2 + q3 + q4) / 4

detach(mydata)
```

If you were to modify an existing variable in your data frame, you would have to reattach it before you would see it. In the following example, we attach mydata and look at q1:

```
> attach(mydata)

> q1

[1] 1 2 2 3 4 5 5 4
```

So we see what q1 looks like. Next, we will see what it looks like squared and then write it to mydata$q1 (choosing a new name would be wiser but would not make this point clear). By specifying the full name mydata$q1, we know R will write it to the original data frame, not the temporary working copy:

```
> mydata$q1^2

[1]  1  4  4  9 16 25 25 16

> mydata$q1 <- q1^2
```

However, what does the short name of q1 show us? The unmodified temporary version!

```
> q1

[1] 1 2 2 3 4 5 5 4
```

If we `attach` the file again, it will essentially make a new temporary copy and q1 finally shows that we did indeed square it:

```
> attach(mydata)

The following object(s) are masked from mydata (position 3):
  gender q1 q2 q3 q4 workshop

> q1

[1]  1  4  4  9 16 25 25 16
```

The message warning about masked objects is telling you that there were other objects with those names that are now not accessible. Those are just the ones we attached earlier, so that is fine. We could have avoided this message by detaching mydata before attaching it a second time. The only problem that confronts us now is a bit of wasted workspace.

Just like SAS or SPSS, R does *all* of its calculations in the computer's main memory. You can use them immediately, but they will exist only in your current session unless you save your workspace. You can use either the `save` or the `save.image` function to write your work to a file:

```
setwd("c:/myRfolder")
save.image("mydataTransformed.RData")
```

See Chap. 13, "Managing Your Files and Workspace," for more ways to save new variables.

10.1.1 Example Programs for Transforming Variables

SAS Program for Transforming Variables

```
* Filename: Transform.sas ;

LIBNAME myLib  C:\myRfolder ;

DATA myLib.mydataTransformed;
SET  myLib.mydata;
totalq = (q1 + q2 + q3 + q4);
logtot = log10(totalq);
mean1 = (q1 + q2 + q3 + q4) / 4;
mean2 = mean(of q1-q4);

PROC PRINT; RUN;
```

SPSS Program for Transforming Variables

```
* Filename: Transform.sps .
```

```
CD  C:\myRfolder .
GET FILE= mydata.sav .

COMPUTE Totalq = q1 + q2 + q3 + q4.
COMPUTE Logtot = lg10(totalq).
COMPUTE Mean1  = (q1 + q2 + q3 + q4) / 4.
COMPUTE Mean2  = MEAN(q1 TO q4).

SAVE OUTFILE= C:\myRfolder\mydataTransformed.sav .
LIST.
```

R Program for Transforming Variables

```
# Filename: Transform.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
mydata

# Transformation in the middle of another function.
summary( log(mydata$q4) )

# Creating meanQ with dollar notation.
mydata$meanQ <- (mydata$q1 + mydata$q2
               + mydata$q3 + mydata$q4) / 4
mydata

# Creating meanQ using attach.
attach(mydata)
mydata$meanQ <- (q1 + q2 + q3 + q4) / 4
detach(mydata)
mydata

# Creating meanQ using transform.
mydata <- transform(mydata,
  meanQ=(q1 + q2 + q3 + q4) / 4 )
mydata

# Creating two variables using transform.
mydata <- transform(mydata,
  score1 = (q1 + q2) / 2,
  score2 = (q3 + q4) / 2 )
mydata

# Creating meanQ using index notation on the left.
```

```
load(file = "mydata.RData")
mydata <- data.frame( cbind( mydata, meanQ = 0.) )
mydata[7] <- (mydata$q1 + mydata$q2 +
              mydata$q3 + mydata$q4) / 4
mydata
```

10.2 Procedures or Functions? The apply Function Decides

The last section described simple data transformations, using mathematics and algebra. We applied functions like logarithms to one variable at a time. I avoided the use of statistical functions.

SAS and SPSS each have two independent ways to calculate statistics: functions and procedures. Statistical *functions* work within each observation to calculate a statistic like the mean of our q variables for each observation. Statistical *procedures* work within a variable to calculate statistics like the mean of our q4 variable across all observations. Mathematical transformations affect one variable at a time, unless you use a DO loop to apply the same function to variable after variable.

R, on the other hand, has only one way to calculate: *functions*. What determines if a function is working on variables or observations is how you *apply* it! How you apply a function also determines how many variables or observations a function works on, eliminating much of the need for DO loops. This is a *very* different perspective!

Let us review an example from the previous section:

```
mydata$meanQ <- (mydata$q1 + mydata$q2
                 mydata$q3 + mydata$q4) / 4
```

This approach gets tedious with long lists of variables. It also has a problem with missing values. The meanQ variable will be missing if any of the variables has a missing value. The mean function solves that problem.

10.2.1 Applying the mean Function

We saw previously that R has both a mean function and a summary function. For numeric objects, the mean function returns a single value, whereas the summary function returns the minimum, first quartile, median, mean, third quartile, and maximum. We could use either of these functions to create a meanQ variable. However, the mean function returns only the value we need, so it is better for this purpose.

Let us first call the mean function on mydata while selecting just the q variables:

```
> mean(mydata[3:6], na.rm = TRUE)

      q1       q2       q3       q4
3.250000 2.750000 4.142857 3.250000
```

We see that the `mean` function went down the columns of our data frame like a SAS procedure or SPSS command would do.

To try some variations, let us put our q variables into a matrix. Simply selecting the variables with the command below will not convert them into matrix form. Even though variables 3 through 6 are all numeric, the selection will maintain its form as a data frame:

```
mymatrix <- mydata[ ,3:6]  # Not a matrix!
```

The proper way to convert the data is with the `as.matrix` function:

```
> mymatrix <- as.matrix( mydata[3:6] )

> mymatrix

     q1 q2 q3 q4
[1,]  1  1  5  1
[2,]  2  1  4  1
[3,]  2  2  4  3
[4,]  3  1 NA  3
[5,]  4  5  2  4
[6,]  5  4  5  5
[7,]  5  3  4  4
[8,]  4  5  5  5
```

Let us review what happens if we use the `mean` function on mymatrix:

```
> mean(mymatrix, na.rm = TRUE)

[1] 3.322581
```

This is an interesting ability, but it is not that useful in our case. What is of much more interest is the mean of each variable, as a SAS/SPSS procedure would do, or the mean of each observation, as a SAS/SPSS function would do. We can do either by using the `apply` function. Let us start by getting the means of the variables:

```
> apply(mymatrix, 2, mean, na.rm = TRUE)

      q1       q2       q3       q4
3.250000 2.750000 4.142857 3.250000
```

That is the same result as we saw from simply using the `mean` function on the q variables.

The `apply` function call above has three arguments and passes a fourth on to the `mean` function.

1. The name of the matrix (or array) you wish to analyze. If you supply a data frame instead, it will coerce it into a matrix if possible (i.e., if all its variables are of the same type). In our case we could have used `mydata[ ,3:6]` since `apply` would have coerced it into a matrix on the fly. I coerced it into a matrix manually to emphasize that that is what R is doing behind the scenes. It also clarifies the call to the `apply` function.
2. The *margin* you want to apply the function over, with 1 representing rows and 2 representing columns. This is easy to remember since R uses the subscript order of [rows, columns], so the margin values are [1, 2], respectively.
3. The function you want to apply to each row or column. In our case, this is the `mean` function. It is important to note that you can only apply only a single function. If you wish to apply a formula, perhaps involving multiple functions, you must first create a new function that does what you need, and then apply it.
4. The `apply` function passes any other arguments on to the function you are applying. In our case, `na.rm = TRUE` is an argument for the `mean` function, not the `apply` function. If you look at the help file for the `apply` function, you will see its form is `apply(X, MARGIN, FUN, ...)`. That means it only uses three arguments, but the *triple dot* argument shows that it will pass other arguments, indicated by the ellipsis "...", to the function "FUN" (`mean` in our case).

Applying the `mean` function to *rows* is as easy as changing the value 2, representing columns, to 1, representing rows:

```
> apply(mymatrix, 1, mean, na.rm = TRUE)

       1        2        3        4        5
2.000000 2.000000 2.750000 2.333333 3.750000
       6        7        8
4.750000 4.000000 4.750000
```

Since means and sums are such popular calculations, there are specialized functions to get them: `rowMeans`, `colMeans`, `rowSums`, and `colSums`. For example, to get the row means of mymatrix, we can do

```
> rowMeans(mymatrix, na.rm = TRUE)

       1        2        3        4        5
2.000000 2.000000 2.750000 2.333333 3.750000
       6        7        8
```

```
4.750000 4.000000 4.750000
```

To add a new variable to our data frame that is the mean of the q variables, we could any *one* of the following forms:

```
> mydata$meanQ <- apply(mymatrix, 1, mean, na.rm = TRUE)

> mydata$meanQ <- rowMeans(mymatrix, na.rm = TRUE)

> mydata <- transform(mydata,
+   meanQ = rowMeans(mymatrix, na.rm = TRUE)
+ )

> mydata

  workshop gender q1 q2 q3 q4    meanQ
1        R      f  1  1  5  1 2.000000
2      SAS      f  2  1  4  1 2.000000
3        R      f  2  2  4  3 2.750000
4      SAS   <NA>  3  1 NA  3 2.333333
5        R      m  4  5  2  4 3.750000
6      SAS      m  5  4  5  5 4.750000
7        R      m  5  3  4  4 4.000000
8      SAS      m  4  5  5  5 5.750000
```

Finally, we can apply a function to each vector in a data frame by using the lapply function. A data frame is a type of list, and the letter "l" in lapply stands for *l*ist. The function applies other functions to lists, and it returns its results in a list. Since it is clear we want to apply the function to each component in the list, there is no need for a row/column margin argument.

```
> lapply(mydata[ ,3:6], mean, na.rm = TRUE)

$q1
[1] 3.25

$q2
[1] 2.75

$q3
[1] 4.1429

$q4
[1] 3.25
```

Since the output is in the form of a list, it takes up more space when printed than the vector output from the apply function. You can also use the `sapply` function on a data frame. The “s” in `sapply` means it *s*implifies its output whenever possible to vector, matrix, or array form. Its simplified vector output would be much more compact:

```
> sapply(mydata[ ,3:6], mean, na.rm = TRUE)

      q1       q2       q3       q4
3.250000 2.750000 4.142857 3.250000
```

Since the result is a vector, it is very easy to get the mean of the means:

```
> mean(
+   sapply(mydata[ ,3:6], mean, na.rm = TRUE)
+  )

[1] 3.3482
```

Other statistical functions that work very similarly are shown in Table 10.2. The `length` function is similar to the SAS `N` function or SPSS `NVALID` function, but different enough to deserve its own section (below).

10.2.2 Finding N or NVALID

In SAS, saying, `N(q1, q2, q3, q4)` or in SPSS saying, `NVALID(Q1 TO Q4)` would count the valid values of those variables for each observation. Running descriptive statistical procedures would give you the number of valid observations for each variable. R has several variations on this theme. First, let us look at the `length` function:

```
> length( mydata[ ,"q3"] )

[1] 8
```

The variable q3 has seven valid values and one missing value. The `length` function tells us the number of total responses. Oddly enough, it does not have an `na.rm` argument to get rid of that missing value.

Since every variable in a data frame must have the same length, the `nrow` function will give us the same answer as the previous function call:

```
> nrow(mydata)

[1] 8
```

If you were seeking the number of observations on the data frame, that would be the best way to do it. However, we are after the number of valid observations *per variable*. One approach is to ask for values that are not missing. The “!” sign means “not,” so let us try the following:

```
> !is.na( mydata[ ,"q3"] )

[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

This identifies them logically. Since statistical functions will interpret TRUE as 1 and FALSE as 0, summing them will give us the number of valid values:

```
> sum( !is.na( mydata[ ,"q3"] ) )

[1] 7
```

It boggles my mind that such complexity is considered the standard approach to calculating such a simple and frequently needed measure! Luckily, Lemon and Grosjean's `prettyR` package has a `valid.n` function that does that very calculation. Let us load that package from our library and apply the function to our data frame using `sapply`:

```
> library("prettyR")

> sapply(mydata, valid.n)

workshop   gender        q1        q2        q3        q4
       8        7         8         8         7         8
```

That is the kind of output we would get from descriptive statistics procedures in SAS or SPSS. In Chap. 17, "Statistics," we will see functions that provide that information and much more, like means and standard deviations.

What about applying it across rows, like the SAS `N` function or the SPSS `NVALID` function? Let us create a myQn variable that contains the number of valid responses in q1 through q4. First, we will pull those variables out into a matrix. That will let us use the `apply` function on the rows:

```
> mymatrix <- as.matrix( mydata[ ,3:6] )

> mymatrix

  q1 q2 q3 q4
1  1  1  5  1
2  2  1  4  1
3  2  2  4  3
4  3  1 NA  3
5  4  5  2  4
6  5  4  5  5
7  5  3  4  4
8  4  5  5  5
```

Now we use the `apply` function with the `margin` argument set to 1, which asks it to go across rows:

```
> apply(mymatrix, 1, valid.n)

1 2 3 4 5 6 7 8
4 4 4 3 4 4 4 4
```

So we see that all of the observations have four valid values except for the fourth. Now let us do that again, but this time save it in our data frame as the variable myQn.

```
> mydata$myQn <- apply(mymatrix, 1, valid.n)

> mydata

  workshop gender q1 q2 q3 q4 myQn
1        1      f  1  1  5  1    4
2        2      f  2  1  4  1    4
3        1      f  2  2  4  3    4
4        2   <NA>  3  1 NA  3    3
5        1      m  4  5  2  4    4
6        2      m  5  4  5  5    4
7        1      m  5  3  4  4    4
8        2      m  4  5  5  5    4
```

Another form of the `apply` function is `tapply`. It exists to create *t*ables by applying a function repeatedly to groups in the data. For details, see Sect. 10.12, "Creating Summarized or Aggregated Data Sets."

There is also the `mapply` function, which is a multivariate version of `sapply`. See `help("mapply")` for details. Wickham's `plyr` package has a complete set of applying functions that are very popular.

The functions we have examined in this section are very basic. Their sparse output is similar to the output from SAS and SPSS functions. For R functions that act more like SAS or SPSS procedures, see Chap. 17, "Statistics." Still, R does not differentiate one type of function from another as SAS and SPSS do for their functions and procedures.

10.2.3 Standardizing and Ranking Variables

The previous section showed how to apply functions to matrices and data frames. To convert our variables to Z scores, we could subtract the mean of each variable and divide by its standard deviation. However, the built-in `scale` function will do that for us:

```
> myZs    <- apply(mymatrix, 2, scale)

> myZs
```

Table 10.2. Basic statistical functions. Note that `valid.n` is in the `prettyR` package.

	R	SAS	SPSS
Maximum	`max(x)`	`MAX(varlist)`	`MAX(varlist)`
Mean	`mean(x)`	`MEAN(varlist)`	`MEAN(varlist)`
Median	`median(x)`	`MEDIAN(varlist)`	`MEDIAN(varlist)`
Minimum	`min(x)`	`MIN(varlist)`	`MIN(varlist)`
N	`valid.n(x)`	`N(varlist)`	`NVALID(varlist)`
Range	`range(x)`	`RANGE(varlist)`	Not equivalent
Rank	`scale(x)`	`PROC RANK`	`RANK Command`
Std. dev.	`sd(x)`	`STD(varlist)`	`SD(varlist)`
Variance	`var(x)`	`VAR(varlist)`	`VARIANCE(varlist)`
Z scores	`scale(x)`	`PROC STANDARD`	`DESCRIPTIVES Cmd.`

```
             q1         q2         q3         q4
[1,] -1.5120484 -0.9985455  0.8017837 -1.4230249
[2,] -0.8400269 -0.9985455 -0.1336306 -1.4230249
[3,] -0.8400269 -0.4279481 -0.1336306 -0.1581139
[4,] -0.1680054 -0.9985455         NA -0.1581139
[5,]  0.5040161  1.2838442 -2.0044593  0.4743416
[6,]  1.1760376  0.7132468  0.8017837  1.1067972
[7,]  1.1760376  0.1426494 -0.1336306  0.4743416
[8,]  0.5040161  1.2838442  0.8017837  1.1067972
```

If you wanted to add these to mydata, you could rename the variables (see Sect. 10.6) and then merge them with mydata (see Sect. 10.11).

Converting the variables to ranks is also easy using the built-in `rank` function:

```
> myRanks <- apply(mymatrix, 2, rank)
> myRanks
      q1  q2 q3  q4
[1,] 1.0 2.0  6 1.5
[2,] 2.5 2.0  3 1.5
[3,] 2.5 4.0  3 3.5
[4,] 4.0 2.0  8 3.5
[5,] 5.5 7.5  1 5.5
[6,] 7.5 6.0  6 7.5
[7,] 7.5 5.0  3 5.5
[8,] 5.5 7.5  6 7.5
```

If you need to apply functions like these by groups, Wickham's `plyr` package makes the task particularly easy.

10.2.4 Applying Your Own Functions

We have seen throughout this book that R has an important feature: Its functions are controlled by arguments *that accept only a single value.* When we provided value labels for a factor, we had to first combine them into a single character vector. When we read two records per case, the field widths of each record were stored in two numeric vectors. We had to combine them into a single list before supplying the information. When using the apply family of functions, this rule continues: You can apply only a single function. If that does not meet your needs, you must create a new function that does, then apply it.

Let us try to apply two functions, mean and sd:

```
> apply(mymatrix, 2, mean, sd)  # No good!

Error in mean.default(newX[, i], ...) :
  trim  must be numeric of length one
```

R warns us that only one is possible. Previously in Sect. 5.9 "Writing Your Own Functions (Macros)," we created several versions of a function called mystats. It returned both the mean and standard deviation. Let us recreate a simple version of it here:

```
mystats <- function(x) {
  c( mean = mean(x, na.rm = TRUE),
       sd = sd  (x, na.rm = TRUE) )
}
```

Now we can apply it:

```
> apply(mymatrix, 2, mystats)

           q1       q2       q3       q4
mean 3.250000 2.750000 4.142857 3.250000
sd   1.488048 1.752549 1.069045 1.581139
```

That worked well.

The apply family of functions also lets you do something unusual: create a function on the fly and use it without even naming it. Functions without names are called *anonymous functions.*

Let us run the example again using an anonymous function:

```
> apply(mymatrix, 2, function(x){
+     c( mean=mean(x, na.rm = TRUE),
+    sd=sd(x, na.rm = TRUE) )
+ } )

           q1       q2       q3       q4
```

```
mean 3.250000 2.750000 4.142857 3.250000
sd   1.488048 1.752549 1.069045 1.581139
```

We have essentially nested the creation of the function within the call to the `apply` function and simply left off its name.

This makes for dense code, so I seldom use this approach. However, you will see it in other books and help files, so it is important to know how it works.

10.2.5 Example Programs for Applying Statistical Functions

SAS Program for Applying Statistical Functions

```
* Filename: ApplyingFunctions.sas;

LIBNAME myLib  C:\myRfolder ;

DATA myLib.mydata;
SET  myLib.mydata;
myMean = MEAN(OF q1-q4);
myN = N(OF q1-q4);
RUN;

PROC MEANS;
  VAR q1-q4 myMean myN;
RUN;

* Get Z Scores;
PROC STANDARD DATA=mylib.mydata
  MEAN=0 STD=1 out=myZs;
RUN;
PROC PRINT;
RUN;

* Get Ranks;
PROC RANK DATA=mylib.mydata OUT=myRanks;
RUN;
PROC PRINT;
RUN;
```

SPSS Program for Applying Statistical Functions

```
* Filename: ApplyingFunctions.SPS.

CD  C:\myRfolder .
```

```
GET FILE= mydata.sav .

* Functions work for each observation (row).
COMPUTE myMean = Mean(q1 TO q4).
COMPUTE mySum  = Sum(q1 TO q4).
COMPUTE myN = mySum / myMean.

* Procedures work for all observations (column).
DESCRIPTIVES  VARIABLES=q1 q2 q3 q4 myMean myN.

* Get Z Scores.
DESCRIPTIVES VARIABLES=q1 q2 q3 q4
  /SAVE
  /STATISTICS=MEAN STDDEV MIN MAX.

* Get Ranks.
RANK VARIABLES=q1 q2 q3 q4 (A)
  /RANK
  /PRINT=YES
  /TIES=MEAN.
EXECUTE.
```

R Program for Applying Statistical Functions

```
# Filename: ApplyingFunctions.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
mydata
attach(mydata)

# Mean of the q variables
mean(mydata[3:6], na.rm = TRUE)

# Create mymatrix.
mymatrix <- as.matrix( mydata[ ,3:6] )
mymatrix

# Get mean of whole matrix.
mean(mymatrix, na.rm = TRUE)

# Get mean of matrix columns.
apply(mymatrix, 2, mean, na.rm = TRUE)

# Get mean of matrix rows.
```

```
apply(mymatrix, 1, mean, na.rm = TRUE)
rowMeans(mymatrix, na.rm = TRUE)

# Add row means to mydata.
mydata$meanQ <- apply(mymatrix, 1, mean, na.rm = TRUE)
mydata$meanQ <- rowMeans(mymatrix, na.rm = TRUE)
mydata <- transform(mydata,
  meanQ = rowMeans(mymatrix, na.rm = TRUE)
)
mydata

# Means of data frames & their vectors.
lapply(mydata[ ,3:6], mean, na.rm = TRUE)
sapply(mydata[ ,3:6], mean, na.rm = TRUE)

mean(
  sapply(mydata[ ,3:6], mean, na.rm = TRUE)
)

# Length of data frames & their vectors.
length(mydata[ ,"q3"] )
nrow(mydata)
is.na( mydata[ ,"q3"] )
!is.na( mydata[ ,"q3"] )
sum( !is.na( mydata[ ,"q3"] ) )

# Like the SAS/SPSS n from stat procedures.
library("prettyR")
sapply(mydata, valid.n)

apply(myMatrix, 1, valid.n)
mydata$myQn <- apply(myMatrix, 1, valid.n)
mydata

# Applying Z Transformation.

myZs    <- apply(mymatrix, 2, scale)
myZs
myRanks <- apply(mymatrix, 2, rank)
myRanks

# Applying Your Own Functions.

apply(mymatrix, 2, mean, sd)  # No good!
```

```
mystats <- function(x) {
  c( mean=mean(x, na.rm = TRUE),
       sd=sd  (x, na.rm = TRUE) )
}
apply(mymatrix, 2, mystats)

apply(mymatrix, 2, function(x){
    c( mean = mean(x, na.rm = TRUE),
     sd = sd(  x, na.rm = TRUE) )
} )
```

10.3 Conditional Transformations

Conditional transformations apply different formulas to various groups in your data. For example, the formulas for recommended daily allowances of vitamins differ for males and females. Conditional transformations are also used to chop up continuous variables into categories like low and high, or grades like A, B, C, D, F. Such transformations can be useful in understanding and explaining relationships in our data, but they also result in a dramatic loss of information. Your chance of finding a significant relationship when using such simplified variables is shockingly worse than with the original variables. Analyses should always be done using the most continuous form of your measures, saving chopped-up versions for simplifying explanations of the results.

R's `ifelse` function does conditional transformations in a way that is virtually identical to what SAS and SPSS do. R also has a variety of cutting functions that are very useful for chopping up variables into groups. We will consider both in this section. Sect. 10.7, "Recoding Variables," offers a different solution to similar problems.

10.3.1 The `ifelse` Function

The general form of the `ifelse` function is:

```
ifelse(logic, true, false)
```

where "logic" is a logical condition to test, "true" is the value to return when the logic is true, and "false" is the value to return when the logic is false. For example, to create a variable that has a value of 1 for people who strongly agree with question 4 on our survey, we could use

```
> mydata$q4Sagree <- ifelse( q4 == 5, 1, 0 )

> mydata$q4Sagree

[1] 0 0 0 0 0 1 0 1
```

Table 10.3. Logical operators. See also `help("Logic")` and `help("Syntax")`.

<table>
<tr><th></th><th>R</th><th>SAS</th><th>SPSS</th></tr>
<tr><td>Equals</td><td>==</td><td>= or EQ</td><td>= or EQ</td></tr>
<tr><td>Less than</td><td><</td><td>< or LT</td><td>< or LT</td></tr>
<tr><td>Greater than</td><td>></td><td>> or GT</td><td>> or GT</td></tr>
<tr><td>Less than or equal</td><td><=</td><td><= or LE</td><td><= or LE</td></tr>
<tr><td>Greater than or equal</td><td>>=</td><td>>= or GE</td><td>>= or GE</td></tr>
<tr><td>Not equal</td><td>!=</td><td>^=, <> or NE</td><td>~= or NE</td></tr>
<tr><td>And</td><td>&</td><td>&, AND</td><td>&, AND</td></tr>
<tr><td>Or</td><td>|</td><td>|, OR</td><td>|, OR</td></tr>
<tr><td>0<=x<=1</td><td>(0<=x) & (x<=1)</td><td>0<=x<=1</td><td>(0<=x) & (x<=1)</td></tr>
<tr><td>Missing value size</td><td>Missing values have no size</td><td>Minus infinity</td><td>Missing values have no size</td></tr>
<tr><td>Identify missing values</td><td>is.na(x)==TRUE (x==NA can never be true.)</td><td>x=.</td><td>MISSING(x)=1</td></tr>
</table>

This is such a simple outcome that we can also do this using

```
mydata$q4Sagree <- as.numeric(q4 == 5)
```

However, the latter approach only allows the outcomes of 1 and 0, whereas the former version allows for any value. The statement `q4 == 5` will result in a vector of logical TRUE/FALSE values. The `as.numeric` function converts it into zeros and ones.

R uses some different symbols for logical comparisons, such as the "`==`" for logical equality. Table 10.3 shows the different symbols used by each package.

If we want a variable to indicate when people agree with question 4, (i.e., they responded with agree or strongly agree), we can use

```
> mydata$q4agree <- ifelse(q4 >= 4, 1, 0)

> mydata$q4agree

[1] 0 0 0 0 1 1 1 1
```

The logical condition can be as complicated as you like. Here is one that creates a score of 1 when people took workshop 1 (abbreviated ws1) and agreed that it was good:

```
> mydata$ws1agree <- ifelse(workshop == 1 & q4 >=4 , 1, 0)

> mydata$ws1agree

[1] 0 0 0 0 1 0 1 0
```

We can fill in equations that will supply values under the two conditions. The following equations for males and females are a bit silly, but they make the point obvious. SAS users might think that if gender were missing, the second equation would apply. Luckily, R is more like SPSS in this regard, and if gender is missing, it sets the result to missing.

```
mydata$score <- ifelse(gender == "f",
  (2 * q1) + q2,
  (3 * q1) + q2
)
```

What follows is our resulting data frame:

```
> mydata

  workshop gender q1 q2 q3 q4 q4Sagree q4agree ws1agree score
1      1      f   1  1  5  1        0       0        0     3
2      2      f   2  1  4  1        0       0        0     5
3      1      f   2  2  4  3        0       0        0     6
4      2    <NA>  3  1 NA  3        0       0        0    NA
5      1      m   4  5  2  4        0       1        1    17
6      2      m   5  4  5  5        1       1        0    19
7      1      m   5  3  4  4        0       1        1    18
8      2      m   4  5  5  5        1       1        0    17
```

Let us now consider conditional transformations that divide or cut a continuous variable into a number of groups. One of the most common examples of this is cutting test scores into grades. Our practice data set mydata100 is just like mydata, but it has 100 observations and includes a pretest and posttest score. Let us read it in and cut posttest into groups:

```
detach(mydata)
load("mydata100.RData")
attach(mydata100)

> head(mydata100)

  gender workshop q1 q2 q3 q4 pretest posttest
1 Female        R  4  3  4  5      72       80
2   Male     SPSS  3  4  3  4      70       75
3   <NA>     <NA>  3  2 NA  3      74       78
4 Female     SPSS  5  4  5  3      80       82
5 Female    Stata  4  4  3  4      75       81
6 Female     SPSS  5  4  3  5      72       77
```

First we will use `elseif` in a form that is not very efficient but easy to understand:

```
> postGroup <- posttest
> postGroup <- ifelse(posttest< 60              , 1, postGroup)
> postGroup <- ifelse(posttest>=60 & posttest<70, 2, postGroup)
> postGroup <- ifelse(posttest>=70 & posttest<80, 3, postGroup)
> postGroup <- ifelse(posttest>=80 & posttest<90, 4, postGroup)
> postGroup <- ifelse(posttest>=90              , 5, postGroup)

> table(postGroup)

postGroup
 1  2  3  4  5
 1  3 31 52 13
```

The very first statement, `postGroup <- posttest`, is important just to give postGroup some initial values. It must exist and have the same length as posttest (the initial values are not important) since it is used at the end of the very first `ifelse`.

A much more efficient approach nests each `ifelse` in the FALSE position of the previous `ifelse` call:

```
> postGroup <-
+ ifelse(posttest <  60                , 1,
+   ifelse(posttest >= 60 & posttest < 70, 2,
+     ifelse(posttest >= 70 & posttest < 80, 3,
+       ifelse(posttest >= 80 & posttest < 90, 4,
+         ifelse(posttest >= 90                , 5, posttest)
+ ))))

> table(postGroup)

postGroup
 1  2  3  4  5
 1  3 31 52 13
```

With this approach, as soon as the TRUE state of a value is determined, the following `ifelse` function calls are not checked. Note now that posttest itself appears only one time, in the final FALSE position. Therefore, we did not need to initialize postGroup with `postGroup <- posttest`.

You can also perform this transformation by taking advantage of a logical TRUE being equivalent to a mathematical 1. Each observation gets a 1, and then another 1 is added for every condition that is true:

```
> postGroup <- 1+
+   (posttest >= 60)+
+   (posttest >= 70)+
+   (posttest >= 80)+
+   (posttest >= 90)

> table(postGroup)

postGroup
 1  2  3  4  5
 1  3 31 52 13
```

These examples use code that is quite long, so I have avoided using the even longer form of mydata100$postGroup. When you do this type of transformation, remember to save your new variable in your data frame with:

```
mydata100$postGroup <- postGroup
```

or an equivalent command.

10.3.2 Cutting Functions

R has a more convenient approach to cutting continuous variables into groups. They are called *cutting functions*. R's built-in function for this is named cut, but the similar cut2 function from Harrell's Hmisc package has some advantages. To use it, you simply provide it with the variable to cut and a vector of cut points:

```
> library("Hmisc")

> postGroup <- cut2(posttest, c(60, 70, 80, 90) )

> table(postGroup)

postGroup
     59 [60,70) [70,80) [80,90) [90,98]
      1       3      31      52      13
```

It creates a factor with default labels that indicate where the data got cut. The labels even show you that 59 was the smallest score and 98 was the largest.

SAS and SPSS do not offer the approach just presented, but they do offer a way to cut a variable up into groups determined by percentiles using their rank procedures (see example programs at the end of this section).

The cut2 function can do this, too.[1] To cut a variable up into equal-sized groups, you specify the number of groups you want using the g (for group) argument:

[1] R's built-in cut function cannot do this directly, but you could nest a call to quantile within it.

```
> postGroup <- cut2(posttest, g = 5)

> table(postGroup)

postGroup
[59,78) [78,82) [82,85) [85,89) [89,98]
     26      22      17      19      16
```

Previously we had equally spaced intervals. You can see that `cut2` has tried instead to create groups of equal size.

If you know in advance the size of groups you would like, you can also use `cut2` and specify the minimum number of observations in each group using the `m` (for minimum) argument:

```
> postGroup <- cut2(posttest, m = 25)

> table(postGroup)

postGroup
[59,78) [78,83) [83,87) [87,98]
     26      27      23      24
```

Another method of cutting is to form the groups into naturally occurring clusters. Williams' `Rattle` package includes a `binning` function that does that.

10.3.3 Example Programs for Conditional Transformations

SAS Program for Conditional Transformations

```
* Filename: TransformIF.sas ;

LIBNAME myLib  C:\myRfolder ;

DATA myLib.mydataTransformed;
SET myLib.mydata;
  IF q4  = 5 then x1 = 1; else x1 = 0;
  IF q4 >= 4 then x2 = 1; else x2 = 0;
  IF workshop = 1 & q4 >= 5 then  x3 = 1;
     ELSE x3 = 0;
  IF gender = "f" then scoreA = 2 * q1 + q2;
                  Else scoreA = 3 * q1 + q2;
  IF workshop = 1 and q4 >= 5
     THEN scoreB = 2 * q1 + q2;
     ELSE scoreB = 3 * q1 + q2;
 RUN;
 PROC PRINT; RUN;

DATA myLib.mydataTransformed;
SET myLib.mydata100;

IF      (posttest <  60)                 THEN postGroup = 1;
ELSE IF (posttest >= 60 & posttest < 70) THEN postGroup = 2;
ELSE IF (posttest >= 70 & posttest < 80) THEN postGroup = 3;
ELSE IF (posttest >= 80 & posttest < 90) THEN postGroup = 4;
ELSE IF (posttest >= 90)                 THEN postGroup = 5;
RUN;

PROC FREQ; TABLES postGroup; RUN;

PROC RANK OUT=myLib.mydataTransformed GROUPS=5;
  VAR posttest;
  RUN;

PROC FREQ; TABLES posttest; RUN;
```

SPSS Program for Conditional Transformations

```
*Filename: TransformIF.sps .

CD  C:\myRfolder .
GET FILE =  mydata.sav .

DO IF (Q4 eq 5).
+ COMPUTE X1 = 1.
ELSE.
+ COMPUTE X1 = 0.
END IF.

DO IF (Q4 GE 4).
+ COMPUTE X2 = 1.
ELSE.
+ COMPUTE X2 = 0.
END IF.

DO IF (workshop EQ 1 AND q4 GE 4).
+ COMPUTE X3 = 1.
ELSE.
+ COMPUTE X3 = 0.
END IF.
```

```
DO IF (gender =  f ).
+ COMPUTE scoreA = 2 * q1 + q2.
ELSE.
+ COMPUTE scoreA = 3 * q1 + q2.
END IF.

DO IF (workshop EQ 1 AND q4 GE 5).
+ COMPUTE scoreB = 2 * q1 + q2.
ELSE.
+ COMPUTE scoreB = 3 * q1 + q2.
END IF.
EXECUTE.

GET FILE= mydata100.sav .
DATASET NAME DataSet2 WINDOW=FRONT.

DO IF (posttest LT  60).
+ COMPUTE  postGroup = 1.
ELSE IF (posttest GE 60  AND posttest LT 70).
+ COMPUTE postGroup = 2.
ELSE IF (posttest GE 70 AND posttest LT 80).
+ COMPUTE postGroup = 3.
ELSE IF (posttest GE 80 AND posttest LT 90).
+ COMPUTE postGroup = 4.
ELSE IF (posttest GE 90).
+ COMPUTE postGroup = 5.
END IF.
EXECUTE.

DATASET ACTIVATE DataSet2.
FREQUENCIES VARIABLES=postGroup
  /ORDER=ANALYSIS.

DATASET ACTIVATE DataSet1.
RANK VARIABLES=posttest (A)
  /NTILES(5)
  /PRINT=YES
  /TIES=MEAN.

FREQUENCIES VARIABLES=Nposttes
  /ORDER=ANALYSIS.
```

R Program for Conditional Transformations

```
# Filename: TransformIF.R
```

```
setwd("c:/myRfolder")
load(file = "mydata.RData")
mydata
attach(mydata)

mydata$q4Sagree <- ifelse(q4 == 5, 1, 0)
mydata$q4Sagree

mydata$q4Sagree <- as.numeric(q4 == 5 )
mydata$q4Sagree

mydata$q4agree <- ifelse(q4 >= 4, 1, 0)
mydata$q4agree

mydata$ws1agree <- ifelse(workshop == 1 & q4 >=4 , 1, 0)
mydata$ws1agree

mydata$score <- ifelse(gender == "f",
  (2 * q1) + q2,
  (3 * q1) + q2
)
mydata

# ---Cutting posttest---

detach(mydata)
load("mydata100.RData")
attach(mydata100)
head(mydata100)

# An inefficient approach:
postGroup <- posttest
postGroup <- ifelse(posttest< 60             , 1, postGroup)
postGroup <- ifelse(posttest>=60 & posttest<70, 2, postGroup)
postGroup <- ifelse(posttest>=70 & posttest<80, 3, postGroup)
postGroup <- ifelse(posttest>=80 & posttest<90, 4, postGroup)
postGroup <- ifelse(posttest>=90             , 5, postGroup)

table(postGroup)

# An efficient approach:
postGroup <-
ifelse(posttest <  60                , 1,
  ifelse(posttest >= 60 &  posttest < 70, 2,
    ifelse(posttest >= 70 &  posttest < 80, 3,
      ifelse(posttest >= 80 &  posttest < 90, 4,
        ifelse(posttest >= 90                , 5, posttest)
))))
table(postGroup)

# Logical approach:
postGroup <- 1+
  (posttest >= 60)+
  (posttest >= 70)+
  (posttest >= 80)+
  (posttest >= 90)
table(postGroup)

# ---Cutting Functions---

# Hmisc cut2 function
library("Hmisc")
postGroup <- cut2(posttest, c(60, 70, 80, 90) )
table(postGroup)

postGroup <- cut2(posttest, g = 5)
table(postGroup)

postGroup <- cut2(posttest, m = 25)
table(postGroup)
```

10.4 Multiple Conditional Transformations

Conditional transformations apply different formulas to different subsets of your data. If you have only a single formula to apply to each group, read Sect. 10.3, "Conditional Transformations." SAS and SPSS both offer distinctly different approaches to *single* conditional transformations and *multiple* conditional transformations. However, R uses the same approach regardless of how many transformations you need to apply to each group. It does, however, let us look at some interesting variations in R.

The simplest approach is to use the `ifelse` function a few times. Let us create two scores, cleverly named score1 and score2, which are calculated differently for the males and the females. Here are the two scores for the females:

```
mydata$score1 <- ifelse( gender == "f",
  (2 * q1) + q2,  # Score 1 for females.
  (20* q1) + q2   # Score 1 for males.
)
mydata$score2 <- ifelse( gender == "f",
  (3  * q1) + q2,  # Score 2 for females.
  (30 * q1) + q2   # Score 2 for males.
)
```

As earlier, the calls to the `ifelse` functions have three arguments.

1. The `gender == "f"` argument provides the logical condition to test.
2. The first formula applies to the TRUE condition, for the females.
3. The second formula applies to the FALSE condition, for the males.

We can do the same thing using the index approach, but it is a bit trickier. First, let us add the new score names to our data frame so that we can refer to the columns by name:

```
> mydata <- data.frame(mydata, score1 = NA, score2 = NA)

> mydata

  workshop gender q1 q2 q3 q4 score1 score2
1        R      f  1  1  5  1     NA     NA
2      SAS      f  2  1  4  1     NA     NA
3        R      f  2  2  4  3     NA     NA
4      SAS   <NA>  3  1 NA  3     NA     NA
5        R      m  4  5  2  4     NA     NA
6      SAS      m  5  4  5  5     NA     NA
7        R      m  5  3  4  4     NA     NA
8      SAS      m  4  5  5  5     NA     NA
```

This initializes the scores to missing. We could also have initialized them to zero by changing "`score1=NA, score2=NA`" to "`score1=0, score2=0`".

Next, we want to differentiate between the genders. We can use the form `gender == "f"`, but we do not want to use it directly as indices to our data frame because gender has a missing value. What would R do with `mydata[NA, ]`? Luckily, the `which` function only cares about TRUE values, so we will use that to locate the observations we want:

```
> gals <- which( gender == "f" )

> gals

[1] 1 2 3

> guys <- which( gender == "m" )
> guys

[1] 5 6 7 8
```

We can now use the gals and guys variables to make the actual formula with the needed indices much shorter:

```
> mydata[gals, "score1"] <-  2 * q1[gals] + q2[gals]
> mydata[gals, "score2"] <-  3 * q1[gals] + q2[gals]
> mydata[guys, "score1"] <- 20 * q1[guys] + q2[guys]
> mydata[guys, "score2"] <- 30 * q1[guys] + q2[guys]

> mydata

  workshop gender q1 q2 q3 q4 score1 score2
1        R      f  1  1  5  1      3      4
2      SAS      f  2  1  4  1      5      7
3        R      f  2  2  4  3      6      8
4      SAS   <NA>  3  1 NA  3     NA     NA
5        R      m  4  5  2  4     85    125
6      SAS      m  5  4  5  5    104    154
7        R      m  5  3  4  4    103    153
8      SAS      m  4  5  5  5     85    125
```

We can see that this approach worked, but look closely at the index values. We are selecting observations based on the rows. So where is the required comma? When we attached the data frame, variables q1 and q2 became accessible by their short names. In essence, they are vectors now, albeit temporary ones. Vectors can use index values, too, but since they only have one dimension, they only use one index. If we had not attached the file, we would have had to write the formulas as:

```
2* mydata[gals, "q1"] + mydata[gals, "q2"]
```

We no longer need the guys and gals variables, so we can remove them from our workspace.

```
rm(guys,gals)
```

10.4.1 Example Programs for Multiple Conditional Transformations

SAS Program for Multiple Conditional Transformations

```
* Filename: TransformIF2.sas ;

LIBNAME myLib  C:\myRfolder ;
```

```
DATA myLib.mydata;
SET  myLib.mydata;

IF gender="f" THEN DO;
  score1 = (2 * q1) + q2;
  score2 = (3 * q1) + q2;
END;

ELSE IF gender="m" THEN DO;
  score1 = (20 * q1) + q2;
  score2 = (30 * q1) + q2;
END;

RUN;
```

SPSS Program for Multiple Conditional Transformations

```
* Filename: TransformIF2.sps .

CD  C:\myRfolder .
GET FILE =  mydata.sav .

DO IF (gender EQ  m ).
+  COMPUTE score1 = (2*q1) + q2.
+  COMPUTE score2 = (3*q1) + q2.
ELSE IF (gender EQ  f ).
+  COMPUTE  score1 = (20*q1) + q2.
+  COMPUTE  score2 = (30*q1) + q2.
END IF.

EXECUTE.
```

R Program for Multiple Conditional Transformations

```
# Filename: TransformIF2.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
attach(mydata)
mydata

# Using the ifelse approach.
mydata$score1 <- ifelse( gender == "f",
  (2 * q1) + q2,  # Score 1 for females.
  (20* q1) + q2   # Score 1 for males.
)
mydata$score2 <- ifelse( gender == "f",
  (3  * q1) + q2,  # Score 2 for females.
  (30 * q1) + q2   # Score 2 for males.
)
mydata

# Using the index approach.

load(file = "mydata.RData")

# Create names in data frame.
detach(mydata)
mydata <- data.frame(mydata, score1 = NA, score2 = NA)
mydata
attach(mydata)

# Find which are males and females.
gals <- which( gender == "f" )
gals
guys <- which( gender == "m" )
guys

mydata[gals, "score1"] <-  2 * q1[gals] + q2[gals]
mydata[gals, "score2"] <-  3 * q1[gals] + q2[gals]
mydata[guys, "score1"] <- 20 * q1[guys] + q2[guys]
mydata[guys, "score2"] <- 30 * q1[guys] + q2[guys]
mydata

# Clean up.
rm(guys, gals)
```

10.5 Missing Values

We discussed missing values briefly in several previous chapters. Let us bring those various topics together to review and expand on them. R represents missing values with NA, for Not Available. The letters NA are also an object in R that you can use to assign missing values. Unlike SAS, the value used to store NA is not the smallest number your computer can store, so logical comparisons such as $x < 0$ will result in NA when x is missing. SAS would give a result of TRUE instead, while the SPSS result would be missing.

When importing numeric data, R reads blanks as missing (except when blanks are delimiters). R reads the string NA as missing for both numeric and character variables. When importing a text file, both SAS and SPSS would recognize a period as a missing value for numeric variables. R will, instead, read the whole variable as a character vector! If you have control over the source of the data, it is best not to write them out that way. If not, you can use a text editor to replace the periods with NA, but you have to be careful to do so in a way that does not also replace valid decimal places. Some editors make that easier than others. A safer method would be to fix it in R, which we do below.

When other values represent missing, you will, of course, have to tell R about them. The `read.table` function provides an argument, `na.strings`, that allows you to provide a set of missing values. However, it applies that value to every variable, so its usefulness is limited. Here is a data set that we will use to demonstrate the various ways to set missing values. The data frame we use, mydataNA, is the same as mydata in our other examples, except that it uses several missing-value codes:

```
> mydataNA <- read.table("mydataNA.txt")

> mydataNA

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4 99
3        .      f  9  2  4  3
4        2      .  3  9 99  3
5        1      m  4  5  2  4
6        .      m  9  9  5  5
7        1      .  5  3 99  4
8        2      m  4  5  5 99
```

In the data we see that workshop and gender have periods as missing values, q1 and q2 have 9's, and q3 and q4 have 99s. Do not be fooled by the periods in workshop and gender; they are not already set to missing! If so, they would have appeared as NA instead. R has seen the periods and has converted both variables to character (string) variables. Since `read.table` converts string variables to factors unless the `as.is = TRUE` argument is added, both workshop and gender are now factors. We can set all three codes to missing by simply adding the `na.strings` argument to the `read.table` function:

```
> mydataNA <- read.table("mydataNA.txt",
+   na.strings = c(".", "9", "99") )

> mydataNA

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
```

```
2         2      f  2  1  4 NA
3        NA      f NA  2  4  3
4         2   <NA>  3 NA NA  3
5         1      m  4  5  2  4
6        NA      m NA NA  5  5
7         1   <NA>  5  3 NA  4
8         2      m  4  5  5 NA
```

If the data did not come from a text file, we could still easily scan every variable for 9 and 99 to replace with missing values using:

```
mydata[mydata == " "] <- NA
```

Both of the above approaches treat all variables alike. If any variables, like age, had valid values of 99, each approach would set them to missing too! For how to handle that situation, see Sect. 10.5.3, "When "99" Has Meaning." Of course "." never has meaning by itself, so getting rid of them all with `na.strings = "."` is usually fine.

10.5.1 Substituting Means for Missing Values

There are several methods for replacing missing values with estimates of what they would have been. These methods include simple mean substitution, regression, and – the gold standard– multiple imputation. We will just do mean substitution. For a list of R packages that do missing value imputation, see the table "R-SAS-SPSS Add-on Module Comparison" under "Missing Value Analysis" available at http://r4stats.com.

Any logical comparison on NAs results in an NA outcome, so `q1 == NA` will never be TRUE, even when q1 is indeed NA. Therefore, if you wanted to substitute another value such as the mean, you would need to use the `is.na` function. Its output is TRUE when a value is NA. Here is how you can use it to substitute missing values (this assumes the data frame is attached):

```
mydataNA$q1[ is.na(q1) ] <- mean( q1, na.rm = TRUE )
```

On the left-hand side, the statement above selects mydataNA$q1 as a vector and then finds its missing elements with `is.na(mydata$q1)`. On the right, it calculates the mean of q1 across all observations to assign to those NA values on the left. We are attaching mydata so we can use short variables names to simplify the code, but we are careful to use the long form, mydataNA$q1, where we write the result. This ensures that the result will be stored within the data frame, mydata, rather than in the attached copy. See Sect. 13.3, "Attaching Data Frames," for details.

10.5.2 Finding Complete Observations

You can omit all observations that contain any missing values with the `na.omit` function. The new data frame, myNoMissing, contains no missing values for any variables.

```
> myNoMissing <- na.omit(mydataNA)

> myNoMissing

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
5        1      m  4  5  2  4
```

Yikes! We do not have much data left. Thank goodness this is not our dissertation data. The `complete.cases` function returns a value of TRUE when a case is complete – that is, when an observation has no missing values:

```
> complete.cases(mydataNA)

[1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

Therefore, we can use this to get the cases that have no missing values (the same result as the `na.omit` function) by doing

```
> myNoMissing <- mydataNA[ complete.cases(mydataNA), ]

> myNoMissing

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
5        1      m  4  5  2  4
```

Since we already saw `na.omit` do that, it is of greater interest to do the reverse. If we want to see which observations contain any missing values, we can use “!” for NOT:

```
> myIncomplete <- mydataNA[ !complete.cases(mydataNA), ]

> myIncomplete

  workshop gender q1 q2 q3 q4
2        2      f  2  1  4 NA
3       NA      f NA  2  4  3
4        2   <NA>  3 NA NA  3
6       NA      m NA NA  5  5
7        1   <NA>  5  3 NA  4
8        2      m  4  5  5 NA
```

10.5.3 When "99" Has Meaning

Occasionally, data sets use different missing values for different sets of variables. In that case, the methods described earlier would not work because they assume every missing value code applies to all variables.

Variables often have several missing value codes to represent things like "not applicable," "do not know," and "refused to answer." Early statistics programs used to read blanks as zeros, so researchers got used to filling their fields with as many 9's as would fit. For example, a two-column variable such as years of education would use 99, to represent missing. The data set might also have a variable like age, for which 99 is a valid value. Age, requiring three columns, would have a missing value of 999. Data archives like the Interuniversity Consortium of Political and Social Research (ICPSR) have many data sets coded with multiple values for missing.

We will use conditional transformations, covered earlier in this chapter, to address this problem. Let us read the file again and put NAs in for the values 9 and 99 independently:

```
> mydataNA <- read.table("mydataNA.txt", na.strings = ".")

> attach(mydataNA)

> mydataNA$q1[q1 == 9 ] <- NA
> mydataNA$q2[q2 == 9 ] <- NA
> mydataNA$q3[q3 == 99] <- NA
> mydataNA$q4[q4 == 99] <- NA

> mydataNA

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4 NA
3       NA      f NA  2  4  3
4        2   <NA>  3 NA NA  3
5        1      m  4  5  2  4
6       NA      m NA NA  5  5
7        1   <NA>  5  3 NA  4
8        2      m  4  5  5 NA
```

That approach can handle any values we might have and assign NAs only where appropriate, but it would be quite tedious with hundreds of variables. We have used the apply family of functions to execute the same function across sets of variables. We can use that method here. First, we need to create some functions, letting x represent each variable. We can do this using the index method:

```
my9isNA   <- function(x) { x[x == 9  ] <- NA; x}
my99isNA  <- function(x) { x[x == 99 ] <- NA; x}
```

or we could use the `ifelse` function:

```
my9isNA  <- function(x) { ifelse( x == 9,  NA, x) }
my99isNA <- function(x) { ifelse( x == 99, NA, x) }
```

Either of these approaches creates functions that will return a value of NA when x == 9 or x == 99 and will return a value of just x if they are false. If you leave off that last "`...x}`" above, what will the functions return when the conditions are false? That would be undefined, so every value would become NA!

Now we need to apply each function where it is appropriate, using the `lapply` function.

```
> mydataNA <- read.table("mydataNA.txt", na.strings = ".")

> attach(mydataNA)

> mydataNA[3:4] <- lapply( mydataNA[3:4], my9isNA  )
> mydataNA[5:6] <- lapply( mydataNA[5:6], my99isNA )

> mydataNA

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4 NA
3       NA      f NA  2  4  3
4        2   <NA>  3 NA NA  3
5        1      m  4  5  2  4
6       NA      m NA NA  5  5
7        1   <NA>  5  3 NA  4
8        2      m  4  5  5 NA
```

The `sapply` function could have done this, too. With our small data frame, this has not saved us much effort. However, to handle thousands of variables, all we would need to change are the above indices from 3:4 and 5:6 to perhaps 3:4000 and 4001:6000.

10.5.4 Example Programs to Assign Missing Values

SAS Program to Assign Missing Values

```
* Filename: MissingValues.sas ;
```

```
LIBNAME myLib 'C:\myRfolder\';

DATA myLib.mydata;
SET  myLib.mydata;

*Convert 9 to missing, one at a time.
IF q1=9 THEN q1=.;
IF q2=9 THEN q2=.;
IF q3=99 THEN q2=.;
IF q4=99 THEN q4=.;

* Same thing but is quicker for lots of vars;
ARRAY q9 q1-q2;
DO OVER q9;
   IF q9=9 THEN q=.;
END;

ARRAY q99 q3-q4;
DO OVER q99;
   IF q=99 THEN q99=.;
END;
```

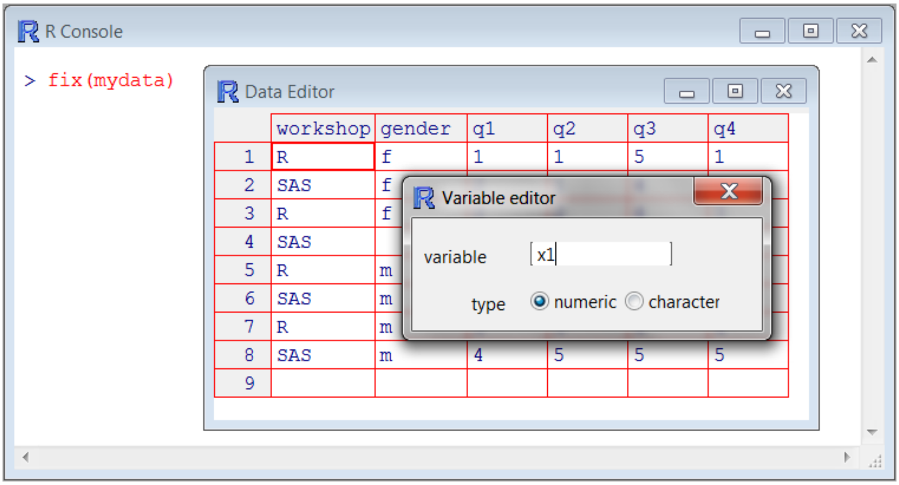
SPSS Program to Assign Missing Values

```
* Filename: MissingValues.sps .

CD 'C:\myRfolder'.
GET FILE=('mydata.sav').

MISSING q1 TO q2 (9) q3 TO q4 (99).

SAVE OUTFILE='mydata.sav'.
```

R Program to Assign Missing Values

```
# Filename: MissingValues.R

setwd("c:/myRfolder")

mydataNA <- read.table("mydataNA.txt")
mydataNA

# Read it so that ".", 9, 99 are missing.
mydataNA <- read.table("mydataNA.txt",
  na.strings = c(".", "9", "99") )
```

```
mydataNA

# Convert 9 and 99 manually
mydataNA <- read.table("mydataNA.txt",
  na.strings=".")
mydataNA[mydataNA == 9 | mydataNA == 99] <- NA
mydataNA
# Substitute the mean for missing values.
mydataNA$q1[is.na(mydataNA$q1)] <-
  mean(mydataNA$q1, na.rm = TRUE)
mydataNA

# Eliminate observations with any NAs.
myNoMissing <- na.omit(mydataNA)
myNoMissing

# Test to see if each case is complete.
complete.cases(mydataNA)

# Use that result to select complete cases.
myNoMissing <- mydataNA[ complete.cases(mydataNA), ]
myNoMissing

# Use that result to select incomplete cases.
myIncomplete <- mydataNA[ !complete.cases(mydataNA), ]
myIncomplete

# When "99" Has Meaning...
mydataNA <- read.table("mydataNA.txt", na.strings = ".")
mydataNA
attach(mydataNA)

# Assign missing values for q variables.
mydataNA$q1[q1 ==  9] <- NA
mydataNA$q2[q2 ==  9] <- NA
mydataNA$q3[q3 == 99] <- NA
mydataNA$q4[q4 == 99] <- NA
mydataNA
detach(mydataNA)

# Read file again, this time use functions.
mydataNA <- read.table("mydataNA.txt", na.strings = ".")
mydataNA
attach(mydataNA)
```



```
> fix(mydata)
```

	workshop	gender	q1	q2	q3	q4
1	R	f	1	1	5	1
2	SAS	f				
3	R	f				
4	SAS					
5	R	m				
6	SAS	m				
7	R	m				
8	SAS	m	4	5	5	5
9						

variable x1

type ● numeric ○ character

Fig. 10.1. Renaming a variable using R's data editor

```
#Create a functions that replaces 9, 99 with NAs.
my9isNA   <- function(x) { x[x ==  9] <- NA; x}
my99isNA  <- function(x) { x[x == 99] <- NA; x}

# Now apply our functions to the data frame using lapply.
mydataNA[3:4] <- lapply( mydataNA[3:4], my9isNA )
mydataNA[5:6] <- lapply( mydataNA[5:6], my99isNA )
mydataNA
```

10.6 Renaming Variables (and Observations)

In SAS and SPSS, you do not know where variable names are stored or how. You just know they are in the data set somewhere. Renaming is simply a matter of matching the new name to the old name with a RENAME statement. In R however, both row and column names are stored in attributes – essentially character vectors – within data objects. In essence, they are just another form of data that you can manipulate.

If you use Microsoft Windows, you can see the names in R's data editor, and changing them there manually is a very easy way to rename them. The function call `fix(mydata)` brings up the data editor. Clicking on the name of a variable opens a box that enables you to change its name. In Fig. 10.1, I am in the midst of changing the name of the variable q1 (see the name in the spreadsheet) to x1.

Closing the variable editor box, the data editor completes your changes. If you use Macintosh or Linux, the `fix` function does not work this way.

However, on any operating system, you can use functions to change variable names.

The programming approach to changing names that feels the closest to SAS or SPSS is the `rename` function in Wickham's `reshape2` package [70]. It works with the names of data frames or lists. To use it, install the package and then load it with the `library` function. Then create a character vector whose *values* are the new names. The *names* of the vector elements are the old variable names. This approach makes it particularly easy to to rename only a subset of your variables. It also feels very familiar to SAS or SPSS users since it follows the `old-name = new-name` style of their `RENAME` commands.

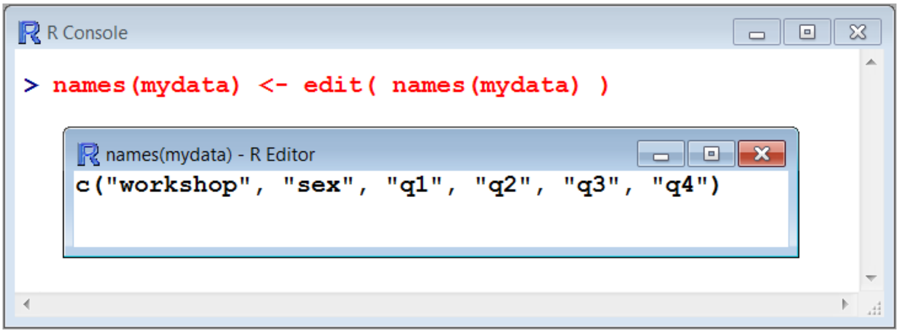
```
> library("reshape2")

> myChanges <- c(q1 = "x1", q2 = "x2", q3 = "x3", q4 = "x4")

> myChanges

  q1   q2   q3   q4
"x1" "x2" "x3" "x4"
```

Now it is very easy to change names with

```
> mydata <- rename(mydata, myChanges)

> mydata

  workshop gender x1 x2 x3 x4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2   <NA>  3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

R's built-in approach to renaming variables is very different. It takes advantage of the fact that the variable names are simply stored as a character vector. Therefore, any method for manipulating vectors or character strings will work with them. You just have to know how to access the name vector. That is what the `names` function does. Simply entering `names(mydata)` causes R to print out the names vector:

```
> names(mydata)

[1] "workshop" "gender" "q1" "q2" "q3" "q4"
```



```
> names(mydata) <- edit( names(mydata) )
```

```
c("workshop", "sex", "q1", "q2", "q3", "q4")
```

Fig. 10.2. Renaming variables using the `edit` function

The `names` function works for data frames, lists, and vectors. You can also assign a character vector of equal length to that function call, which renames the variables. With this approach, you supply a name for every variable.

```
> names(mydata) <- c("workshop","gender","x1","x2","x3","x4")

> mydata

  workshop gender x1 x2 x3 x4
1      1        f  1  1  5  1
2      2        f  2  1  4  1
...
```

If you work with matrices, use the very similar `rownames` and `colnames` functions. You access the names of arrays via the `dimnames` function.

You can also use subscripting for this type of renaming. Since gender is the second variable in our data frame, you could change just the name gender to sex as follows:

```
names(mydata)[2] <- "sex"
```

The `edit` function, described in Sect. 6.1, “The R Data Editor,” will generate a character vector of variable names, complete with the `c` function and parentheses. In Fig. 10.2, you can see the command I entered and the window that it opened. I have changed the name of the variable “gender” to “sex.” When I finish my changes, closing the box will execute the command.

10.6.1 Advanced Renaming Examples

The methods shown above are often sufficient to rename your variables. You can view this section either as beating the topic to death, or as an opportunity to gain expertise in vector manipulations that will be helpful well outside the topic at hand. The approach used in Sect. 10.6.4, “Renaming Many Sequentially Numbered Variable Names,” can be a real time saver.

10.6.2 Renaming by Index

Let us extract the names of our variables using the `names` function.

```
> mynames <- names(mydata)

> mynames

[1] "workshop"  "gender" "q1"      "q2"      "q3"      "q4"
```

Now we have a character vector whose values we can change using the R techniques we have covered elsewhere. This step is not really necessary since the names are already stored in a vector inside our data frame, but it will make the examples easier to follow. We would like to obtain the index value of each variable name. Recall that whenever a data frame is created, row names are added that are sequential numbers by default. So we can use the `data.frame` function to number our variable names:

```
> data.frame(mynames)

  mynames
1  workshop
2  gender
3      q1
4      q2
5      q3
6      q4
```

We see from the above list that q1 is the third name and q4 is the sixth. We can now use that information to enter new names directly into this vector and print the result so that we can see if we made any errors:

```
> mynames[3] <- "x1"
> mynames[4] <- "x2"
> mynames[5] <- "x3"
> mynames[6] <- "x4"

> mynames

[1] "workshop"  "gender" "x1"      "x2"      "x3"      "x4"
```

That looks good, so let us place those new names into the names attribute of our data frame and look at the results:

```
> names(mydata) <- mynames
> mydata
```

```
  workshop gender x1 x2 x3 x4
1     1      f  1  1  5  1
2     2      f  2  1  4  1
...
```

As you will see in the program below, each time we use another method of name changes, we need to restore the old names to demonstrate the new techniques. We can accomplish this by either reloading our original data frame or using

```
names(mydata) <- c("workshop", "gender", "q1", "q2", "q3", "q4")
```

10.6.3 Renaming by Column Name

If you prefer to use variable names instead of index numbers, that is easy to do. We will make another copy of mynames:

```
> mynames <- names(mydata)

> mynames

[1] "workshop"  "gender" "q1"     "q2"     "q3"     "q4"
```

Now we will make the same changes but using a logical match to find where `mynames == "q1"` and so on and assigning the new names to those locations:

```
> mynames[ mynames == "q1" ] <- "x1"
> mynames[ mynames == "q2" ] <- "x2"
> mynames[ mynames == "q3" ] <- "x3"
> mynames[ mynames == "q4" ] <- "x4"

> mynames

[1] "workshop"  "gender"  "x1"     "x2"     "x3"     "x4"
```

Finally, we put the new set mynames into the names attribute of our data frame, mydata:

```
> names(mydata) <- mynames

> mydata

  workshop gender x1 x2 x3 x4
1     1      f     1  1  5  1
2     2      f     2  1  4  1
...
```

You can combine all these steps into one, but it can be very confusing to read at first:

```
names(mydata)[names(mydata) == "q1"] <- "x1"
names(mydata)[names(mydata) == "q2"] <- "x2"
names(mydata)[names(mydata) == "q3"] <- "x3"
names(mydata)[names(mydata) == "q4"] <- "x4"
```

10.6.4 Renaming Many Sequentially Numbered Variable Names

Our next example works well if you are changing many variable names, like 100 variables named x1, x2, etc. over to similar names like y1, y2, and so forth. You occasionally have to make changes like this when you measure many variables at different times and you need to rename the variables in each data set before joining them all.

In Sect. 7.4, "Selecting Variables by Column Name," we learned how the `paste` function can append sequential numbers onto any string. We will use that approach here to create the new variable names:

```
> myXs <- paste( "x", 1:4, sep = "")

> myXs

[1] "x1" "x2" "x3" "x4"
```

Now we need to find out where to put the new names. We already know this of course, but we found that out in the previous example by listing all of the variables. If we had thousands of variables, that would not be a very good method. We will use the method we covered previously (and in more detail) in the Sect. 7.11, "Generating indices A to Z from Two Variable Names":

```
> myA <- which( names(mydata) == "q1" )

> myA

[1] 3

> myZ <- which( names(mydata) == "q4" )

> myZ

[1] 6
```

Now we know the indices of the variable names to replace; we can replace them with the following command:

```
> names(mydata)[myA:myZ] <- myXs

> mydata

  workshop gender x1 x2 x3 x4
1     1      f     1  1  5  1
2     2      f     2  1  4  1
...
```

10.6.5 Renaming Observations

R has row names that work much the same as variable names, but they apply to observations. These names must be unique and often come from an ID variable. When reading a text file using functions like `read.csv`, the `row.names` argument allows you to specify an ID variable. See Sect. 6.2, "Reading Delimited Text Files," for details.

Row names are stored in a vector called the *row names attribute*. Therefore, when renaming rows using a variable, you must select it so that it will pass as a vector. In the examples below, the first three select a variable named "id" as a vector, so they work. The last approach looks almost like the first, but it selects id as a data frame, which will not fit in the row names attribute. Recall that leaving out the comma in `mydata["id"]` makes R select a variable as a data frame. The moral of the story is that when renaming observations using index values, *keep the comma*!

```
> row.names(mydata) <- mydata[ ,"id"]  # This works.

> row.names(mydata) <- mydata$id       # This works too.

> row.names(mydata) <- mydata[["id"]]  # This does too.

> row.names(mydata) <- mydata["id"]    # This does not.

Error in  row.names<-.data.frame ( *tmp* ,
  value = list(id = c(1, 2, 3,  :
  invalid  row.names  length
```

10.6.6 Example Programs for Renaming Variables

For many of our programming examples, the R programs are longer because they demonstrate a wider range of functionality. In this case, renaming variables is definitely easier in SAS and SPSS. R does have a greater flexibility in this area, but it is an ability that only a fanatical programmer could love!

SAS Program for Renaming Variables

```
* Filename: Rename.sas ;
LIBNAME myLib  C:\myRfolder ;

DATA myLib.mydata;
  RENAME q1-q4=x1-x4;

  *or;
  *RENAME q1=x1  q2=x2  q3=x3  q4=x4;

RUN;
```

SPSS Program for Renaming Variables

```
* Filename: Rename.sps .

CD  C:\myRfolder .
GET FILE= mydata.sav .
RENAME VARIABLES (q1=x1)(q2=x2)(q3=x3)(q4=x4).
* Or...
RENAME VARIABLES (q1 q2 q3 q4 = x1 x2 x3 x4).
* Or...
RENAME VARIABLES (q1 TO q4 = x1 TO x4).
```

R Program for Renaming Variables

```
# Filename: Rename.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
mydata

# Using the data editor.

fix(mydata)
mydata

# Restore original names for next example.
names(mydata) <- c("workshop", "gender",
  "q1", "q2", "q3", "q4")

# Using the reshape2 package.

library("reshape2")
```

```
myChanges <- c(q1 = "x1", q2 = "x2", q3 = "x3", q4="x4")
myChanges

mydata <- rename(mydata, myChanges)
mydata

# Restore original names for next example.
names(mydata) <- c("workshop", "gender",
  "q1", "q2", "q3", "q4")

# The standard R approach.
names(mydata) <- c("workshop", "gender",
  "x1", "x2", "x3", "x4")
mydata

# Restore original names for next example.
names(mydata) <- c("workshop", "gender",
  "q1", "q2", "q3", "q4")

# Using the edit function.
names(mydata) <- edit( names(mydata) )
mydata

# Restore original names for next example.
names(mydata) <- c ("workshop", "gender",
  "q1", "q2", "q3", "q4")

#---Selecting Variables by Index Number---

mynames <- names(mydata)

# Data.frame adds index numbers to names.
data.frame(mynames))

# Then fill in index numbers in brackets.
mynames[3] <- "x1"
mynames[4] <- "x2"
mynames[5] <- "x3"
mynames[6] <- "x4"

# Finally, replace old names with new.
names(mydata) <- mynames
mydata

# Restore original names for next example.
```

```
names(mydata) <- c("workshop", "gender",
  "q1", "q2", "q3", "q4")

#---Selecting Variables by Name---

# Make a copy to work on.
mynames <- names(mydata)
mynames

# Replace names in copy.
mynames[ mynames == "q1" ] <- "x1"
mynames[ mynames == "q2" ] <- "x2"
mynames[ mynames == "q3" ] <- "x3"
mynames[ mynames == "q4" ] <- "x4"
mynames

# Then replace the old names.
names(mydata) <- mynames
mydata

# Restore original names for next example.
names(mydata) <- c ("workshop", "gender",
  "q1", "q2", "q3", "q4")

#---Same as Above, but Confusing!---

names(mydata)[names(mydata) == "q1"] <- "x1"
names(mydata)[names(mydata) == "q2"] <- "x2"
names(mydata)[names(mydata) == "q3"] <- "x3"
names(mydata)[names(mydata) == "q4"] <- "x4"
print(mydata)

# Restore original names for next example.
names(mydata) <- c("workshop", "gender",
  "q1", "q2", "q3", "q4")

#---Replacing Many Numbered Names---

# Examine names
names(mydata)

# Generate new numbered names.
myXs <- paste( "x", 1:4, sep = "")
myXs
```

```
# Find out where to put the new names.
myA <- which( names(mydata) == "q1" )
myA
myZ <- which( names(mydata) == "q4" )
myZ

# Replace names at index locations.
names(mydata)[myA:myZ] <- myXs(mydata)

#remove the unneeded objects.
rm(myXs, myA, myZ)
```

10.7 Recoding Variables

Recoding is just a simpler way of doing a set of related IF/THEN conditional transformations. Survey researchers often collapse five-point Likert-scale items into simpler three-point Disagree/Neutral/Agree scales to summarize results. This can also help when a cross-tabulation (or similar analysis) with other variables creates tables that are too sparse to analyze.

Recoding can also reverse the scale of negatively worded items so that a large numeric value has the same meaning across all items. It is easier to reverse scales by subtracting each score from 6 as in

```
mydata$qr1 <- 6-mydata$q1
```

That results in `6-5=1`, `6-4=2`, and so on.

There are two important issues to consider when recoding data. First, collapsing a scale loses information and power. You will lessen your ability to find significant, and hopefully useful, relationships. Second, recoding nominal categorical variables like race can be disastrous. For example, inexperienced researchers often recode race into Caucasian and Other without checking to see how reasonable that is beforehand. You should do an analysis to see if the groups you are combining show similar patterns with regard to your dependent variable of interest. Given how much time that can add to the overall analysis, it is often far easier to set values to missing. Simply focus your analysis on the groups for which you have sufficient data rather than combine groups without justification.

SAS does not have a separate recode procedure as SPSS does, but it does offer a similar capability using its value label formats. That has the useful feature of applying the formats in categorical analyses and ignoring them otherwise. For example, PROC FREQ will use the format but PROC MEANS will ignore it. You can also recode the data with a series of IF/THEN statements. I show both methods below. For simplicity, I leave the value labels out of the SPSS and R programs. I cover those in Sect. 11.1, "Value Labels or Formats (and Measurement Level)."

For recoding continuous variables into categorical ones, see the cut function in base R and the cut2 function in Frank Harrell's Hmisc package. For choosing optimal cut points with regard to a target variable, see the rpart function in the rpart package or the tree function in the Hmisc package.

It is wise to avoid modifying your original data, so recoded variables are typically stored under new names. If you named your original variables q1, q2, etc. then you might name the recoded ones qr1, qr2, etc. with "r" representing *r*ecoded.

10.7.1 Recoding a Few Variables

We will work with the recode function from John Fox's car package, which you will have to install before running this. See Chap. 2, "Installing and Updating R," for details. We will apply it below to collapse our five-point scale down to a three-point one representing just disagree, neutral, and agree:

```
> library("car")

> mydata$qr1 <- recode(q1, "1=2; 5=4")
> mydata$qr2 <- recode(q2, "1=2; 5=4")
> mydata$qr3 <- recode(q3, "1=2; 5=4")
> mydata$qr4 <- recode(q4, "1=2; 5=4")

> mydata

  workshop gender q1 q2 q3 q4 qr1 qr2 qr3 qr4
1        1      f  1  1  5  1   2   2   4   2
2        2      f  2  1  4  1   2   2   4   2
3        1      f  2  2  4  3   2   2   4   3
4        2   <NA>  3  1 NA  3   3   2  NA   3
5        1      m  4  5  2  4   4   4   2   4
6        2      m  5  4  5  5   4   4   4   4
7        1      m  5  3  4  4   4   3   4   4
8        2      m  4  5  5  5   4   4   4   4
```

The recode function needs only two arguments: the variable you wish to recode and a string of values in the form "old1=new1; old2=new2;...".

10.7.2 Recoding Many Variables

The above approach worked fine with our tiny data set, but in a more realistic situation, we would have many variables to recode. So let us scale this example up. We learned how to rename many variables in Sect. 10.6.4, so we will use that knowledge here.

```
> myQnames <-  paste( "q",  1:4, sep = "")

> myQnames

[1] "q1" "q2" "q3" "q4"

> myQRnames <- paste( "qr", 1:4, sep = "")

> myQRnames

[1] "qr1" "qr2" "qr3" "qr4"
```

Now we will use the original names to extract the variables we want to recode to a separate data frame:

```
> myQRvars <- mydata[ ,myQnames]
> myQRvars

  q1 q2 q3 q4
1  1  1  5  1
2  2  1  4  1
3  2  2  4  3
4  3  1 NA  3
5  4  5  2  4
6  5  4  5  5
7  5  3  4  4
8  4  5  5  5
```

We will use our other set of variable names to rename the variables we just selected:

```
> names(myQRvars) <- myQRnames

> myQRvars

  qr1 qr2 qr3 qr4
1   1   1   5   1
2   2   1   4   1
3   2   2   4   3
4   3   1  NA   3
5   4   5   2   4
6   5   4   5   5
7   5   3   4   4
8   4   5   5   5
```

Now we need to create a function that will allow us to apply the recode function to each of the selected variables. Our function only has one argument, x, which will represent each of our variables:

```
myRecoder <- function(x) { recode(x,"1=2;5=4") }
```

Here is how we can use `myRecoder` on a single variable. Notice that the qr1 variable had a 1 for the first observation, which myRecoder made a 2. It also had values of 5 for the sixth and seventh observations, which became 4s:

```
> myQRvars$qr1

[1] 1 2 2 3 4 5 5 4

> myRecoder(myQRvars$qr1)

[1] 2 2 2 3 4 4 4 4
```

To apply this function to our whole data frame, myQRvars, we can use the `sapply` function:

```
> myQRvars <- sapply( myQRvars, myRecoder)

> myQRvars

     qr1 qr2 qr3 qr4
[1,]   2   2   4   2
[2,]   2   2   4   2
[3,]   2   2   4   3
[4,]   3   2  NA   3
[5,]   4   4   2   4
[6,]   4   4   4   4
[7,]   4   3   4   4
[8,]   4   4   4   4
```

The `sapply` function has converted our data frame to a matrix, but that is fine. We will use the `cbind` function to bind these columns to our original data frame:

```
> mydata <- cbind(mydata,myQRvars)

> mydata

  workshop gender q1 q2 q3 q4 qr1 qr2 qr3 qr4
1        1      f  1  1  5  1   2   2   4   2
2        2      f  2  1  4  1   2   2   4   2
3        1      f  2  2  4  3   2   2   4   3
4        2   <NA>  3  1 NA  3   3   2  NA   3
5        1      m  4  5  2  4   4   4   2   4
6        2      m  5  4  5  5   4   4   4   4
7        1      m  5  3  4  4   4   3   4   4
8        2      m  4  5  5  5   4   4   4   4
```

Now we can use either the original variables or their recoded counterparts in any analysis we choose. In this simple case, it was not necessary to create the myRecoder function. We could have used the form,

```
sapply(myQRvars, recode, "1=2;5=4")
```

However, you can generalize the approach we took to far more situations.

10.7.3 Example Programs for Recoding Variables

SAS Program for Recoding Variables

```
* Filename: Recode.sas ;

LIBNAME myLib  C:\myRfolder ;
DATA myLib.mydata;
INFILE  Š\myRfolder\mydata.csv  csvŠ delimiter =  Š,  Ś
   MISSOVER DSD LRECL=32767 firstobs=2 ;

INPUT id workshop gender $ q1 q2 q3 q4;

PROC PRINT; RUN;

PROC FORMAT;
  VALUE Agreement 1="Disagree" 2="Disagree"
                  3="Neutral"
                  4="Agree"    5="Agree"; run;

DATA myLib.mydata;
  SET myLib.mydata;
  ARRAY q q1-q4;
  ARRAY qr qr1-qr4; *r for recoded;
  DO i=1 to 4;
     qr{i}=q{i};
     if q{i}=1 then qr{i}=2;
     else
     if q{i}=5 then qr{i}=4;
  END;
  FORMAT q1-q4 q1-q4 Agreement.;
RUN;

* This will use the recoded formats automatically;
PROC FREQ; TABLES q1-q4; RUN;

* This will ignore the formats;
* Note high/low values are 1/5;
```

```
PROC UNIVARIATE; VAR q1-q4; RUN;

* This will use the 1-3 codings, not a good idea!;
* High/Low values are now 2/4;
PROC UNIVARIATE; VAR qr1-qr4;
RUN;
```

SPSS Program for Recoding Variables

```
* Filename: Recode.sps .

CD  C:\myRfolder .
GET FILE= mydata.sav .

RECODE q1 to q4 (1=2) (5=4).

SAVE OUTFILE= myleft.sav .
```

R Program for Recoding Variables

```
# Filename: Recode.R

setwd("c:/myRfolder")
load(file = "myWorkspace.RData")
mydata
attach(mydata)

library("car")
mydata$qr1 <- recode(q1, "1=2; 5=4")
mydata$qr2 <- recode(q2, "1=2; 5=4")
mydata$qr3 <- recode(q3, "1=2; 5=4")
mydata$qr4 <- recode(q4, "1=2; 5=4")
mydata

# Do it again, stored in new variable names.
load(file = "mydata.RData")
attach(mydata)

# Generate two sets of var names to use.
myQnames <-  paste( "q",  1:4, sep = "")
myQnames
myQRnames <- paste( "qr", 1:4, sep = "")
myQRnames

# Extract the q variables to a separate data frame.
```

```
myQRvars <- mydata[ ,myQnames]
myQRvars

# Rename all of the variables with R for Recoded.
names(myQRvars) <- myQRnames
myQRvars

# Create a function to apply the labels to lots of variables.
myRecoder <- function(x) { recode(x,"1=2; 5=4") }

# Here s how to use the function on one variable.
myQRvars$qr1
myRecoder(myQRvars$qr1)

#Apply it to all of the variables.
myQRvars <- sapply( myQRvars, myRecoder)
myQRvars

# Save it back to mydata if you want.
mydata <- cbind(mydata,myQRvars)
mydata
summary(mydata)
```

10.8 Indicator or Dummy Variables

When modeling with categorical variables like workshop, it is often useful to create a series of indicator variables (also called dummy variables) that have the value of one when true and zero when false. We have four workshops, so we would usually need three indicator variables. You often need one less indicator variable than there are categories because if someone took a workshop and it was *not* SAS, SPSS, or Stata, then it must have been R. So there are only $k-1$ unique pieces of information contained in a factor with k levels. However, it is occasionally useful to include all k variables in a model with restrictions on the corresponding parameter estimates.

Let us create indicator variables using mydata100 since it has all four workshops:

```
> setwd("c:/myRfolder")

> load("mydata100.RData")

> attach(mydata100)

> head(mydata100)
```

```
  gender workshop q1 q2 q3 q4 pretest posttest
1 Female        R  4  3  4  5      72       80
2   Male     SPSS  3  4  3  4      70       75
3   <NA>     <NA>  3  2 NA  3      74       78
4 Female     SPSS  5  4  5  3      80       82
5 Female    Stata  4  4  3  4      75       81
6 Female     SPSS  5  4  3  5      72       77
```

You can create indicator variables using the `ifelse` function (Sect. 10.3) or using the `recode` function (Sect. 10.7), but the easiest approach is the same as in SAS or SPSS: using logic to generate a series of zeros and ones:

```
> r     <- as.numeric(workshop == "R"    )
> sas   <- as.numeric(workshop == "SAS"  )
> spss  <- as.numeric(workshop == "SPSS" )
> stata <- as.numeric(workshop == "Stata")
```

In each command the logical comparison resulted in TRUE if the person took that workshop or FALSE if not. Then the `as.numeric` function converted that to one or zero, respectively.

To see the result, let us use the `data.frame` function just to line up the variables neatly and then call `head` to print the top six observations:

```
> head( data.frame(
+   workshop, r, sas, spss, stata) )

  workshop  r sas spss stata
1        R  1   0    0     0
2     SPSS  0   0    1     0
3     <NA> NA  NA   NA    NA
4     SPSS  0   0    1     0
5    Stata  0   0    0     1
6     SPSS  0   0    1     0
```

Now that we have the variables, let us use them in a model. To keep it simple, we will do a linear regression that only allows the y-intercepts of each group to differ:

```
> lm(posttest ~ pretest + sas + spss + stata)

Call:
lm(formula = posttest ~ pretest + sas + spss + stata)

Coefficients:
(Intercept) pretest  sas     spss    stata
    16.3108 0.9473  -8.2068 -7.4949 -7.0646
```

I left out the r variable, so the intercept is for the people in the r workshop; all the other weights would be multiplied against their zero values. We will consider linear regression in more detail in Chap. 17.

While knowing how to create indicator variables is useful in R, we did not need to do it in this case. R, being object oriented, tries to do the right thing with your data. Once you have told it that workshop is a factor, it will create the indicator variables for you. Here I use workshop directly in the model:

```
> lm(posttest ~ pretest + workshop)

Call:
lm(formula = posttest ~ pretest + workshop)

Coefficients:
(Intercept) pretest workshopSAS workshopSPSS workshopStata
   16.3108  0.9473  -8.2068     -7.4949      -7.0646
```

So you see that the result is the same except that in our first example we got to choose the variable names while in the second, R chose reasonable names for us.

It is clear in the first example why the R workshop is the "all zeros" level to which the others are being compared because I chose to leave out the variable r. But why did R choose to do that, too? Let us print a few workshop values and then count them:

```
> head(workshop)

[1] R     SPSS  <NA>  SPSS  Stata SPSS
Levels: R SAS SPSS Stata

> table(workshop)

workshop
    R   SAS  SPSS Stata
   31    24    25    19
```

Notice in both cases that when the levels of the factor are displayed that they are in the same order and "R" is the first level. This is caused by their order on the `levels` and matching `labels` arguments when the factor was created. You can change that by using the `relevel` function:

```
> workshop <- relevel(workshop, "SAS")
```

This `relevel` function call used only two arguments, the factor to relevel and the level that we want to make first in line. Now let us see how it changed things:

```
> table(workshop)

workshop
  SAS     R  SPSS Stata
   24    31    25    19

> coef( lm(posttest ~ pretest + workshop) )

(Intercept) pretest   workshopR workshopSPSS workshopStata
 8.1039484  0.9472987 8.2068027 0.7119439    1.1421791
```

We see that the order in the table output has SAS first, and we see that the equation now includes a parameter for all workshops except for SAS. If a person took a workshop and it was *not* for R, SPSS, or Stata, well, then it was for SAS.

If we did a more realistic model and included the interaction between pretest and workshop, R would have generated another three parameters to allow the slopes of each group to differ.

There is a function in the built-in nnet package called class.ind that creates indicator variables. However, it assigns missing values on a factor to zeros on the indicator variables. That is not something I typically want to have happen.

If you have a factor that is ordinal, that is, created using the ordered function rather than factor, then the dummy variables will be coded using polynomial contrasts. For more details on contrasts, see help("contrast").

10.8.1 Example Programs for Indicator or Dummy Variables

SAS Program for Indicator or Dummy Variables

```
* Filename: IndicatorVars.sas ;

LIBNAME myLib  C:\myRfolder ;
DATA  temp; SET myLib.mydata100;
  r     = workshop = 1;
  sas   = workshop = 2;
  spss  = workshop = 3;
  stata = workshop = 4;
RUN;

PROC REG;
  MODEL posttest = pretest sas spss stata;
  RUN;
```

SPSS Program for Indicator or Dummy Variables

This program uses standard SPSS syntax. An alternative approach is to install the Python plug-in and use the SPSSINC CREATE DUMMIES extension command. It will discover the values and generate and label the indictor variables automatically. Its dialog appears on the *Transform> Create Dummy Variables* menu after installation.

```
* Filename: IndicatorVars.sps.

CD  C:\myRfolder .
GET FILE= mydata100.sav .
DATASET NAME DataSet2 WINDOW=FRONT.

COMPUTE r     = workshop EQ 1.
COMPUTE sas   = workshop EQ 2.
COMPUTE spss  = workshop EQ 3.
COMPUTE stata = workshop EQ 4.
EXECUTE.

REGRESSION
  /DEPENDENT posttest
  /METHOD=ENTER pretest sas spss stata.
EXECUTE.
```

R Program for Indicator or Dummy Variables

```
# Filename: IndicatorVars.R

setwd("c:/myRfolder")
load("mydata100.RData")
attach(mydata100)
head(mydata100)

r     <- as.numeric(workshop == "R"    )
sas   <- as.numeric(workshop == "SAS"  )
spss  <- as.numeric(workshop == "SPSS" )
stata <- as.numeric(workshop == "Stata")
head( data.frame(
  workshop, r, sas, spss, stata) )

lm(posttest ~ pretest + sas + spss + stata)
lm(posttest ~ pretest + workshop)
head(workshop)
table(workshop)
```

```
workshop <- relevel(workshop, "SAS")
table(workshop)
coef( lm(posttest ~ pretest + workshop) )

library("nnet")
head( class.ind(workshop) )
```

10.9 Keeping and Dropping Variables

In SAS, you use the KEEP and DROP statements to determine which variables to save in your data set. The SPSS equivalent is the DELETE VARIABLES command. In R, the main methods to do this within a data frame are discussed in Chap. 7, "Selecting Variables." For example, if we want to keep variables on the left side of our data frame, workshop through q2 (variables 1 through 4), an easy way to do this is with

```
myleft <- mydata[ ,1:4]
```

We will strip off the ones on the right side in a future example on merging data frames.

Another way to drop variables is to assign the NULL object to them:

```
mydata$varname <- NULL
```

This has the advantage of removing a variable without having to make a copy of the data frame. That may come in handy with a data frame so large that your workspace will not hold a copy, but it is usually much safer to work on copies when you can. Mistakes happen! You can apply NULL repeatedly with the form

```
myleft <- mydata

myleft$q3 <- myleft$q4 <- NULL
```

NULL is only used to remove components from data frames and lists. You cannot use it to drop elements of a vector, nor can you use it to remove a vector by itself from your workspace.

In Sect. 13.6, "Removing Objects from Your Workspace," we will discuss removing objects using the `rm` function. That function removes only whole objects; it cannot remove variables from within a data frame:

```
rm( mydata$q4 )  # This does NOT work.
```

10.9.1 Example Programs for Keeping and Dropping Variables

SAS Program for Keeping and Dropping Variables

```
* Filename: DropKeep.sas ;

LIBNAME myLib 'C:\myRfolder';
DATA myleft; SET mydata;
  KEEP id workshop gender q1 q2;
PROC PRINT;
RUN;

*or equivalently;
DATA myleft; SET mydata;
  DROP q3 q4;
PROC PRINT;
RUN;
```

SPSS Program for Keeping and Dropping Variables

```
* Filename: DropKeep.sps ;

CD 'C:\myRfolder'.
GET FILE='mydata.sav'.
DELETE VARIABLES q3 to q4.
LIST.
SAVE OUTFILE='myleft.sav'.
```

R Program for Keeping and Dropping Variables

```
# Filename: DropKeep.R

setwd("c:/myRfolder")
load(file = "mydata.RData")

# Using variable selection.
myleft <- mydata[ ,1:4]
myleft

# Using NULL.
myleft <- mydata
myleft$q3 <- myleft$q4 <- NULL
myleft
```

10.10 Stacking/Concatenating/Adding Data Sets

Often we find data divided into two or more sets due to collection at different times or places. Combining them is an important step prior to analysis. SAS calls this *concatenation* and accomplishes this with the SET statement. SPSS calls it *adding cases* and does it using the ADD FILES statement. R, with its row/column orientation, calls it *binding rows*.

To demonstrate this, let us take our practice data set and split it into separate ones for females and males. Then we will bind the rows back together. A `split` function exists to do this type of task, but it puts the resulting data frames into a list, so we will use an alternate approach.

First, let us get the females:

```
> females <- mydata[ which(gender == "f"), ]
> females

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
```

Now we get the males:

```
> males <- mydata[ which(gender == "m"), ]
> males

  workshop gender q1 q2 q3 q4
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

We can put them right back together by binding their rows with the `rbind` function. The “r” in `rbind` stands for *r*ow.

```
> both <- rbind(females, males)

> both

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

This works fine when the two data frames share the exact same variables. Often the data frames you will need to bind will have a few variables missing. We will drop variable q2 in the males data frame to create such a mismatch:

```
> males$q2 <- NULL
> males

  workshop gender q1 q3 q4
5        1      m  4  2  4
6        2      m  5  5  5
7        1      m  5  4  4
8        2      m  4  5  5
```

Note that variable q2 is indeed gone. Now let us try to put the two data frames together again:

```
> both <- rbind(females, males)

Error in match.names(clabs, names(xi)) :
  names do not match previous names
```

It fails because the `rbind` function needs both data frames to have the exact same variable names. Luckily, Wickham's `plyr` package has a function, `rbind.fill`, that binds whichever variables it finds that match and then fills in missing values for those that do not. This next example assumes that you have installed the `plyr` package. See Chap. 2, "Installing and Updating R," for details.

```
> library("plyr")

> both <- rbind.fill(females, males)

> both

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
5        1      m  4 NA  2  4
6        2      m  5 NA  5  5
7        1      m  5 NA  4  4
8        2      m  4 NA  5  5
```

We can do the same thing with the built-in `rbind` function, but we have to first determine which variables we need to add and then add them manually with the `data.frame` function and set them to NA:

```
> males <- data.frame( males, q2=NA )

> males

  workshop gender q1 q3 q4 q2
5        1      m  4  2  4 NA
6        2      m  5  5  5 NA
7        1      m  5  4  4 NA
8        2      m  4  5  5 NA
```

The males data frame now has a variable q2 again, and so we can bind the two data frames using `rbind`. The fact that q2 is now on at the end will not matter. The data frame you list first on the `rbind` function call will determine the order of the final data frame. However, if you use index values to refer to your variables, you need to be aware of the difference!

```
> both <- rbind(females, males)

> both

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
5        1      m  4 NA  2  4
6        2      m  5 NA  5  5
7        1      m  5 NA  4  4
8        2      m  4 NA  5  5
```

With such a tiny data frame, this is an easy way to address the mismatched variables problem. However, in situations that are more realistic, `rbind.fill` is usually a great time saver.

10.10.1 Example Programs for Stacking/Concatenating/Adding Data Sets

SAS Program for Stacking/Concatenating/Adding Data Sets

```
* Filename: Stack.sas ;
LIBNAME myLib  C:\myRfolder ;

DATA males;
  SET mydata;
  WHERE gender= m ;
  RUN;
```

```
PROC PRINT; RUN;

DATA females;
  SET mydata;
  WHERE gender= f ;
  RUN;
PROC PRINT; RUN;

DATA both;
  SET males females;
  RUN;
PROC PRINT; RUN;
```

SPSS Program for Stacking/Concatenating/Adding Data Sets

```
* Filename: Stack.sps .
CD  C:\myRfolder .

GET FILE= mydata.sav .
SELECT IF(gender = "f").
LIST.
SAVE OUTFILE= females.sav .
EXECUTE .

GET FILE= mydata.sav .
SELECT IF(gender = "m").
LIST.
SAVE OUTFILE= males.sav .
EXECUTE .

GET FILE= females.sav .
ADD FILES /FILE=*
 /FILE= males.sav .
LIST.
EXECUTE .
```

R Program for Stacking/Concatenating/Adding Data Sets

```
# Filename: Stack.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
mydata
attach(mydata)
```

```
# Create female data frame.
females <- mydata[ which(gender == "f"), ]
females

# Create male data frame.
males <- mydata[ which(gender == "m"), ]
males

#Bind their rows together with the rbind function.
both <- rbind(females, males)
both

# Drop q2 to see what happens.
males$q2 <- NULL
males

# See that row bind will not work.
both <- rbind(females, males)

# Use plyr rbind.fill.
library("plyr")
both <- rbind.fill(females, males)
both

# Add a q2 variable to males.
males <- data.frame(males, q2 = NA)
males

# Now rbind can handle it.
both <- rbind(females, males)
both
```

10.11 Joining/Merging Data Sets

One of the most frequently used data manipulation methods is joining or merging two data sets, each of which contains variables that the other lacks. SAS does this with its MERGE statement, and SPSS uses its ADD VARIABLES command.

If you have a one-to-many join, it will create a row for every possible match. A common example is a short data frame containing household-level information such as family income joined to a longer data set of individual family member variables. A complete record of each family member along with his or her household income will result. Duplicates in more than one data frame are possible, but you should study them carefully for errors.

So that we will have an ID variable to work with, let us read our practice data without the `row.names` argument. That will keep our ID variable as is and fill in row names with 1, 2, 3, etc.

```
> mydata <- read.table("mydata.csv",
+   header = TRUE, sep = ",", na.strings = " ")

> mydata

  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
3  3        1      f  2  2  4  3
4  4        2   <NA>  3  1 NA  3
5  5        1      m  4  5  2  4
6  6        2      m  5  4  5  5
7  7        1      m  5  3  4  4
8  8        2      m  4  5  5  5
```

Now we will split the left half of the data frame into one called myleft:

```
> myleft <- mydata[ c("id", "workshop", "gender", "q1", "q2") ]

> myleft

  id workshop gender q1 q2
1  1        1      f  1  1
2  2        2      f  2  1
3  3        1      f  2  2
4  4        2   <NA>  3  1
5  5        1      m  4  5
6  6        2      m  5  4
7  7        1      m  5  3
8  8        2      m  4  5
```

We then do the same for the variables on the right, but we will keep id and workshop to match on later:

```
> myright <- mydata[ c("id", "workshop", "q3", "q4") ]

> myright

  id workshop q3  q4
1  1        1  5   1
2  2        2  4   1
```

```
3  3        1  4   3
4  4        2  NA  3
5  5        1  4   4
6  6        2  5   5
7  7        1  4   4
8  8        2  5   5
```

Now we can use the merge function to put the two data frames back together:

```
> both <- merge(myleft, myright, by = "id")

> both

  id workshop.x gender q1 q2 workshop.y q3 q4
1  1          1      f  1  1          1  5  1
2  2          2      f  2  1          2  4  1
3  3          1      f  2  2          1  4  3
4  4          2   <NA>  3  1          2 NA  3
5  5          1      m  4  5          1  2  4
6  6          2      m  5  4          2  5  5
7  7          1      m  5  3          1  4  4
8  8          2      m  4  5          2  5  5
```

This call to the merge function has three arguments:

1. The first data frame to merge.
2. The second data frame to merge.
3. The by argument, which has either a single variable name in quotes or a character vector of names.

If you leave out the by argument, it will match by all variables with common names! That is quite unlike SAS or SPSS, which would simply match the two row by row. That is what the R cbind function will do. It is much safer to match on some sort of ID variable(s), though. Very often, rows do not match up as well as you think they will.

Sometimes the same variable has two different names in the data frames you need to merge. For example, one may have "id" and another "subject." If you have such a situation, you can use the by.x argument to identify the first variable or set of variables and the by.y argument to identify the second. The merge function will match them up in order and do the proper merge. In this next example, I do just that. The variables have the same name, but it still works:

```
> both <- merge(myleft,    myright,
+                by.x = "id", by.y = "id")
```

```
> both

  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
3  3        1      f  2  2  4  3
4  4        2   <NA>  3  1 NA  3
5  5        1      m  4  5  2  4
6  6        2      m  5  4  5  5
7  7        1      m  5  3  4  4
8  8        2      m  4  5  5  5
```

If you have multiple variables in common, but you only want to match on a subset of them, you can use the form

```
both <- merge( myleft, myright,
               by = c("id", "workshop") )
```

If each file had variables with slightly different names, you could use the form

```
both <- merge( myleft,myright, by.x = c("id",     "workshop")
  y.y = c("subject", "shortCourse") )
```

By default, SAS and SPSS keep all records regardless of whether or not they match (a full outer join). For observations that do not have matches in the other file, the `merge` function will fill them in with missing values. R takes the opposite approach, keeping only those that have a record in both (an inner join). To get merge to keep all records, use the argument `all = TRUE`. You can also use `all.x = TRUE` to keep all of the records in the first file regardless of whether or not they have matches in the second. The `all.y = TRUE` argument does the reverse.

While SAS and SPSS can merge any number of files at once, base R can only do two at a time. To do more, you can use the `merge_all` function in the `reshape` package.

10.11.1 Example Programs for Joining/Merging Data Sets

SAS Program for Joining/Merging Data Sets

```
* Filename: Merge.sas ;
LIBNAME myLib  C:\myRfolder ;

DATA  myLib.myleft;
  SET mylib.mydata;
  KEEP id workshop gender q1 q2;
PROC SORT; BY id workshop; RUN;
```

```
DATA  myLib.myright;
  SET myLib.mydata;
  KEEP id workshop q3 q4;
PROC SORT; BY id workshop; RUN;

DATA  myLib.both;
  MERGE myLib.myleft myLib.myright;
  BY id workshop;
RUN;
```

SPSS Program for Joining/Merging Data Sets

```
* Filename: Merge.sps .
CD  C:\myRfolder .

GET FILE= mydata.sav .
DELETE VARIABLES q3 to q4.
SAVE OUTFILE= myleft.sav .

GET FILE= mydata.sav .
DELETE VARIABLES gender, q1 to q2.
SAVE OUTFILE= myright.sav .

GET FILE= myleft.sav .
MATCH FILES /FILE=*
 /FILE= myright.sav
 /BY id.
```

R Program for Joining/Merging Data Sets

```
# Filename: Merge.R
setwd("c:/myRfolder")

# Read data keeping ID as a variable.
mydata <- read.table("mydata.csv",
  header = TRUE, sep = ",", na.strings = " ")
mydata

# Create a data frame keeping the left two q variables.
myleft <- mydata[ c("id", "workshop", "gender", "q1", "q2") ]
myleft

# Create a data frame keeping the right two q variables.
myright <- mydata[ c("id", "workshop", "q3", "q4") ]
myright
```

```
# Merge the two data frames by ID.
both <- merge(myleft, myright, by = "id")
both

# Merge the two data frames by ID.
both <- merge(myleft,    myright,
            by.x = "id", by.y = "id" )

#Merge data frames by both ID and workshop.
both <- merge(myleft, myright, by = c("id","workshop"))
both

#Merge dataframes by both ID and workshop,
#while allowing them to have different names.
both <- merge(myleft,
            myright,
            by.x=c("id", "workshop"),
            by.y=c("id", "workshop") )
both
```

10.12 Creating Summarized or Aggregated Data Sets

We often have to work on data that are a summarization of other data. For example, you might work on household-level data that you aggregated from a data set that had each family member as its own observation. SAS calls this *summarization* and performs it with the SUMMARY procedure. SPSS calls this process *aggregation* and performs it using the AGGREGATE command. Database programmers call this *rolling up* data.

R has three distinct advantages over SAS and SPSS regarding aggregation.

1. It is possible to perform multilevel calculations and selections in a single step, so there is less need to create aggregated data sets.
2. R can aggregate *with every function it has and any function you write*! It is not limited to the few that SAS and SPSS build into SUMMARY and AGGREGATE.
3. R has data structures optimized to hold aggregate results. Other functions offer methods to take advantage of those structures.

10.12.1 The aggregate Function

We will use the `aggregate` function to calculate the mean of the q1 variable by gender and save it to a new (very small!) data frame.

```
> attach(mydata)

> myAgg1 <- aggregate(q1,
+   by = data.frame(gender),
+   mean, na.rm = TRUE)

> myAgg1

  gender        x
1      f 1.666667
2      m 4.500000
```

The `aggregate` function call above has four arguments.

1. The variable you wish to aggregate.
2. One or more grouping factors. Unlike SAS, the data do not have to be sorted by these factors. The factors must be in the form of a list (or data frame, which is a type of list). Recall that single subscripting of a data frame creates a list. So `mydata["gender"]` and `mydata[2]` work. Adding the comma to either one will prevent them from working. Therefore, `mydata[ ,"gender"]` or `mydata[ ,2]` will not work. If you have attached the data frame, `data.frame(gender)` will work. The function call `list(gender)` will also work, but it loses track of the grouping variable names.
3. The function that you wish to apply – in this case – the `mean` function. An important limitation of the `aggregate` function is that it can apply only functions that return a *single* value. If you need to apply a function that returns multiple values, you can use the `tapply` function.
4. Arguments to pass to the function applied. Here `na.rm = TRUE` is passed to the `mean` function to remove missing, or NA, values.

Next we will aggregate by two variables: workshop and gender. To keep our `by` factors in the form of a list (or data frame), we can use any one of the following forms:

```
mydata[ c("workshop", "gender")]
```

or

```
mydata[ c(2, 3) ]
```

or, if you have attached the data frame,

```
data.frame(workshop, gender)
```

In this example, we will use the latter form.

```
> myAgg2 <- aggregate(q1,
+   by = data.frame(workshop, gender),
+   mean, na.rm = TRUE)

> myAgg2

  workshop gender   x
1        R      f 1.5
2      SAS      f 2.0
3        R      m 4.5
4      SAS      m 4.5
```

Now let us use the `mode` and `class` functions to see the type of object the `aggregate` function creates is a data frame:

```
> mode(myAgg2)

[1] "list"

> class(myAgg2)

[1] "data.frame"
```

It is small, but ready for further analysis.

10.12.2 The `tapply` Function

In the last subsection we discussed the `aggregate` function. That function has an important limitation: you can only use it with functions that return single values. The `tapply` function works very similarly to the `aggregate` function but can perform aggregation using any R function. To gain this ability, it has to abandon the convenience of creating a data frame. Instead, its output is in the form of a matrix or an array.

Let us first duplicate the last example from the above subsection using `tapply`:

```
> myAgg2 <- tapply(q1,
+   data.frame(workshop, gender),
+   mean, na.rm = TRUE)

> myAgg2

        gender
workshop    f    m
   R      1.5  4.5
   SAS    2.0  4.5
```

The tapply function call above uses four arguments:

1. The variable to aggregate.
2. One or more grouping factors (or vectors that it will coerce into factors). Unlike SAS and SPSS, the data do not have to be sorted by these factors. This must be in the form of a list (or data frame, which is a list). Recall that single subscripting of a data frame creates a list. So mydata["gender"] and mydata[2] work. Adding the comma to either one will prevent them from working. Therefore, mydata[,"gender"] or mydata[,2] will not work. If you have attached the data frame, data.frame(gender) will work. The function call list(gender) will also work, but it loses track of the grouping variable names.
3. The function to apply – in this case, the mean function. This function can return any result, not just single values.
4. Any additional parameters to pass to the applied function. In this case, na.rm = TRUE is used by the mean function to remove NA or missing values.

The actual means are, of course, the same as we obtained before using the aggregate function. However, the result is now a numeric matrix rather than a data frame.

```
> class(myAgg2)
[1] "matrix"

> mode(myAgg2)
[1] "numeric"
```

Now let us do an example that the aggregate function could not perform. The range function returns two values: the minimum and maximum for each variable:

```
> myAgg2 <- tapply(q1,
+   data.frame(workshop,gender),
+   range, na.rm = TRUE)

> myAgg2

        gender
workshop f         m
   R     Numeric,2 Numeric,2
   SAS   Numeric,2 Numeric,2
```

This output looks quite odd! It is certainly not formatted for communicating results to others. Let us see how it is stored:

```
> mode(myAgg2)
```

```
[1] "list"

> class(myAgg2)

[1] "matrix"
```

It is a matrix, whose elements are lists. Let us look at the entry for the females who took the R workshop. That result is stored in the first row and first column:

```
> class( myAgg2[1,1] )

[1] "list"

> myAgg2[1,1]

[[1]]
[1] 1 2
```

So we see that each component in this matrix is a list that contains a single vector of minimum and maximum values. The opinions of the females who took the R workshop range from 1 to 2. While this output is not very useful for communicating results, it is very useful as input for further programming.

10.12.3 Merging Aggregates with Original Data

It is often useful to add aggregate values back to the original data frame. This allows you to perform multilevel transformations that involve both individual-level and aggregate-level values. A common example of such a calculation is a Z-score, which subtracts a variable's mean and then divides by its standard deviation (see Sect. 10.2.3 for an easier way to do that particular task).

Another important use for merging aggregates with original data is to perform multilevel selections of observations. To select individual-level observations based on aggregate-level values requires access to both at once. For example, we could create a subset of subjects who fall below their group's mean value.

This is an area in which R has a distinct advantage over SAS and SPSS. R's greater flexibility allows it to do both multilevel transformations and selections in a single step.

Now let us calculate a Z-score for variable q1 with the single following statement. Note that we are specifying the long form of the name for our new variable, mydata$Zq1, so that it will go into our data frame.

```
> mydata$Zq1 <- (q1 - mean(q1) ) / sd(q1)
> mydata

  workshop gender q1 q2 q3 q4        Zq1
1        R      f  1  1  5  1 -1.5120484
2      SAS      f  2  1  4  1 -0.8400269
3        R      f  2  2  4  3 -0.8400269
4      SAS   <NA>  3  1 NA  3 -0.1680054
5        R      m  4  5  2  4  0.5040161
6      SAS      m  5  4  5  5  1.1760376
7        R      m  5  3  4  4  1.1760376
8      SAS      m  4  5  5  5  0.5040161
```

You can also select the observations that were below average with this single statement:

```
> mySubset <- mydata[ q1 < mean(q1), ]

> mySubset
  workshop gender q1 q2 q3 q4        Zq1
1        R      f  1  1  5  1 -1.5120484
2      SAS      f  2  1  4  1 -0.8400269
3        R      f  2  2  4  3 -0.8400269
4      SAS   <NA>  3  1 NA  3 -0.1680054
```

SAS and SPSS cannot perform such calculations and selections in one step. You would have to create the aggregate-level data and then merge it back into the individual-level data set. R can use that approach too, and as the number of levels you consider increases, it becomes more reasonable to do so.

So let us now merge myAgg2, created in Sect. 10.12.1, “The `aggregate` Function,” to mydata. To do that, we will rename the mean of q1 from x to mean.q1 using the `rename` function from the `reshape` package. If you do not have that package installed, see Chap. 2, “Installing and Maintaining R.”

```
> library("reshape")

> myAgg3 <- rename(myAgg2, c(x = "mean.q1"))

> myAgg3

  workshop gender mean.q1
1        R      f     1.5
2      SAS      f     2.0
3        R      m     4.5
4      SAS      m     4.5
```

Now we merge the mean onto each of the original observations:

```
> mydata2 <- merge(mydata, myAgg3,
+   by = c("workshop", "gender") )

> mydata2

  workshop gender q1 q2 q3 q4        Zq1 mean.q1
1        R      f  1  1  5  1 -1.5120484     1.5
2        R      f  2  2  4  3 -0.8400269     1.5
3        R      m  4  5  2  4  0.5040161     4.5
4        R      m  5  3  4  4  1.1760376     4.5
5      SAS      f  2  1  4  1 -0.8400269     2.0
6      SAS      m  5  4  5  5  1.1760376     4.5
7      SAS      m  4  5  5  5  0.5040161     4.5
```

The merge function call above has only two arguments.

1. The two data frames to merge. Unlike SAS and SPSS, which can merge many data sets at once, R can only do two at a time.
2. The by argument specifies the variables to match on. In this case, they have the same name in both data frames. They can, however, have different names. See the merge help files for details. While some other functions require by variables in list form, here you provide more than one variable in the form of a character vector.

We can now perform multilevel transformations or selections on mydata2.

10.12.4 Tabular Aggregation

The aim of table creation in SAS and SPSS is to communicate results to people. You can create simple tables of frequencies and percentages using the SAS FREQ procedure and SPSS CROSSTABS. For more complex tables, SAS has PROC TABULATE, and SPSS has its CTABLES procedure. These two create complex tables with basic statistics in almost any form, as well as some basic hypothesis tests. However, no other procedures are programmed to process these tables further automatically. You can analyze them further using the SAS Output Delivery (ODS) or SPSS Output Mangement System (OMS), but not as easily as in R.

R can create tables for presentation, too, but it also creates tables and matrices that are optimized for further use by other functions. They are a different form of aggregated data set. See Chap. 17, “Statistics,” for other uses of tables.

Let us revisit simple frequencies using the table function. First, let us look at just workshop attendance (the data frame is attached, so I am using short variable names):

```
> table(workshop)

workshop
  R SAS
  4   4
```

And now gender and workshop:

```
> table(gender,workshop)
      workshop

gender R SAS
     f 2   1
     m 2   2
```

Let us save this table to an object, myCounts, and check its mode and class:

```
> myCounts <- table(gender, workshop)

> mode(myCounts)

[1] "numeric"

> class(myCounts)

[1] "table"
```

We see that the mode of myCounts is numeric and its class is *table*. Other functions that exist to work with presummarized data know what to do with table objects. In Chap. 15, "Traditional Graphics," we will see the kinds of plots we can make from tables. In Chap. 17, "Statistics," we will also work with table objects to calculate related values like row and column percents.

Other functions prefer count data in the form of a data frame. This is the type of output created by the SAS SUMMARY procedure or the similar SPSS AGGREGATE command. The `as.data.frame` function makes quick work of it:

```
> myCountsDF <- as.data.frame(myCounts)

> myCountsDF

  gender workshop Freq
1      f        R    2
2      m        R    2
3      f      SAS    1
4      m      SAS    2
```

```
> class(myCountsDF)
[1] "data.frame"
```

This approach is particularly useful for people who use analysis of variance. You can get cell counts for very complex models in a form that is very easy to read and use in further analyses.

10.12.5 The plyr and reshape2 Packages

If you perform a lot of data aggregation, you will want to learn how to use two of Wickham's packages. His `plyr` package [73] provides a useful set of apply-like functions that are more comprehensive and consistent than those built into R. His `reshape2` package [70] is also very useful for aggregation. While its main purpose is to reshape data sets, it can also aggregate them as it does so. Its use is covered in Sect. 10.17, "Reshaping Variables to Observations and Back."

10.12.6 Comparing Summarization Methods

Table 10.4. Comparison of summarization functions.See Sect. 10.17 for `reshape2`.

	Input	Functions it can apply	Output
`by`	Data frame	Any function	List with class of "by." Easier to read but not as easy to program
`aggregate`	Data frame	Only functions that return single values	Data frame. Easy to read and program
`tapply`	List or data frame	Any function	List. Easy to access components for programming. Not as nicely formatted for reading.
`table`	Factors	Does counting only	Table object. Easy to read and easy to analyze further.
`reshape2`	Data frame	Only functions that return single values	Data frame (`dcast`) or list (`lcast`). Easy to read and program, especially useful for ANOVA data.

In this section we have examined several methods of summarization. In the following section, we will see that the `by` function not only does analyses in the "by" approach used by SAS and the SPLIT FILE approach used by SPSS, but it can also create summarized data sets. Table 10.4, can help you choose which one to use.

10.12.7 Example Programs for Aggregating/Summarizing Data

SAS Program for Aggregating/Summarizing Data

```
* Filename: Aggregate.sas ;

LIBNAME myLib  C:\myRfolder ;

* Get means of q1 for each gender;
PROC SUMMARY DATA=myLib.mydata MEAN NWAY;
  CLASS GENDER;
  VAR q1;
  OUTPUT OUT=myLib.myAgg;
  RUN;
PROC PRINT; RUN;
DATA myLib.myAgg;
  SET myLib.myAgg;
  WHERE _STAT_= MEAN ŠMEANŠ;
  KEEP gender q1;
RUN;
PROC PRINT; RUN;

*Get means of q1 by workshop and gender;
PROC SUMMARY DATA=myLib.mydata MEAN NWAY;
CLASS WORKSHOP GENDER;
VAR Q1;
OUTPUT OUT=myLib.myAgg;RUN;
PROC PRINT; RUN;

*Strip out just the mean and matching variables;
DATA myLib.myAgg;
  SET myLib.myAgg;
  WHERE _STAT_= MEAN ;
  KEEP workshop gender q1;
  RENAME q1=meanQ1;
RUN;
PROC PRINT; RUN;

*Now merge aggregated data back into mydata;
PROC SORT DATA=myLib.mydata;
  BY workshop gender; RUN:
PROC SORT DATA=myLib.myAgg;
  BY workshop gender; RUN:
DATA myLib.mydata2;
  MERGE myLib.mydata myLib.myAgg;
  BY workshop gender;
PROC PRINT; RUN;
```

SPSS Program for Aggregating/Summarizing Data

```
* Filename: Aggregate.sps .

CD  C:\myRfolder .
GET FILE= mydata.sav .

AGGREGATE
  /OUTFILE= myAgg.sav
  /BREAK=gender
  /q1_mean = MEAN(q1).
GET FILE= myAgg.sav .
LIST.

* Get mean of q1 by workshop and gender.
GET FILE= mydata.sav .
AGGREGATE
  /OUTFILE= myAgg.sav .
  /BREAK=workshop gender
  /q1_mean = MEAN(q1).
GET FILE= myAgg.sav .
LIST.

* Merge aggregated data back into mydata.
* This step can be saved by using
* MODE=ADDVARIABLES in the previous step.
GET FILE= mydata.sav .
SORT CASES BY  workshop (A) gender (A) .
MATCH FILES /FILE=*
 /TABLE= \myAgg.sav
 /BY workshop gender.
SAVE OUTFILE= mydata.sav .
```

R Program for Aggregating/Summarizing Data

```
# Filename: Aggregate.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
attach(mydata)
mydata
```

```
# The aggregate Function.

# Means by gender.
myAgg1 <- aggregate(q1,
  by=data.frame(gender),
  mean, na.rm = TRUE)
myAgg1

# Now by workshop and gender.
myAgg2 <- aggregate(q1,
  by = data.frame(workshop, gender),
  mean, na.rm=TRUE)
myAgg2
mode(myAgg2)
class(myAgg2)

# Aggregation with tapply.

myAgg2 <- tapply(q1,
  data.frame(workshop, gender),
  mean, na.rm = TRUE)
myAgg2
class(myAgg2)
mode(myAgg2)

myAgg2 <- tapply(q1,
  data.frame(workshop, gender),
  range, na.rm = TRUE)
myAgg2
mode(myAgg1)
class(myAgg2)
myAgg2[[1]]

# Example multi-level transformation.

mydata$Zq1 <- (q1 - mean(q1) ) / sd(q1)
mydata

mySubset <- mydata[ q1 < mean(q1), ]
mySubset

# Rename x to be mean.q1.
library("reshape2")
myAgg3 <- rename(myAgg2, c(x = "mean.q1") )
myAgg3
```

```
# Now merge means back with mydata.
mydata2 <- merge(mydata, myAgg3,
  by=c("workshop", "gender") )
mydata2

# Tables of Counts
table(workshop)
table(gender, workshop)
myCounts <- table(gender, workshop)
mode(myCounts)
class(myCounts)

# Counts in Summary/Aggregate style.
myCountsDF <- as.data.frame(myCounts)
myCountsDF
class(myCountsDF)

# Clean up
mydata["Zq1"] <- NULL
rm(myAgg1, myAgg2, myAgg3,
  myComplete, myMeans, myCounts, myCountsDF)
```

10.13 By or Split-File Processing

When you want to repeat an analysis for every level of a categorical variable, you can use the BY statement in SAS, or the SPLIT FILE command in SPSS. SAS requires you to sort the data by the factor variable(s) first, but SPSS and R do not.

R has a by function, which repeats analysis for levels of factors. In Sect. 10.12, "Creating Summarized or Aggregated Data Sets," we did similar things while creating summary data sets. When we finish with this topic, we will compare the two approaches.

Let us look at the by function first and then discuss how it compares to similar functions. We will use the by function to apply the mean function. First, let us use the mean function by itself just for review. To get the means of our q variables, we can use

```
> mean( mydata[ c("q1","q2","q3","q4") ] ,
+        na.rm = TRUE)

    q1     q2     q3     q4
3.25003.2500 2.7500 4.1429 3.7500
```

Now let us get means for the males and females using the by function:

```
> myBYout <- by( mydata[ c("q1","q2","q3","q4") ] ,
+     mydata["gender"],
+     mean,na.rm = TRUE)

> myBYout

gender: f
      q1       q2       q3       q4
1.666667 1.333333 4.333333 1.666667
---------------------------------------------------
gender: m
  q1   q2   q3   q4
4.50 4.25 4.00 4.50
```

The by function call above has four arguments:

1. The data frame name or variables to analyze, mydata[c("q1","q2","q3","q4")].
2. One or more grouping factors. Unlike SAS, the data does not have to be sorted by these factors. The factors must be in the form of a list (or data frame, which is a type of list). Recall that single subscripting of a data frame creates a list. So mydata["gender"] and mydata[2] work. Adding the comma to either one will prevent them from working. Therefore, mydata[,"gender"] or mydata[,2] will not work. If you have attached the data frame, data.frame(gender) will work. The function call list(gender) will also work, but it loses track of the grouping variable names.
3. The function to apply – in this case, the mean function. The by function can apply functions that calculate more than one value (unlike the aggregate function).
4. Any additional arguments are ignored by the by function and simply passed on to the applied function. In this case, na.rm = TRUE is simply passed on to the mean function.

Let us check to see what the mode and class are of the output object.

```
> mode(myBYout)

[1] "list"

> class(myBYout)

[1] "by"
```

It is a list, with a class of "by." If we would like to convert that to a data frame, we can do so with the following commands. The as.table function gets the data into a form that the as.data.frame function can then turn into a data frame:

```
> myBYdata <- as.data.frame( (as.table(myBYout) ) )

> myBYdata

   gender   Freq.f Freq.m
q1      f 1.666667   4.50
q2      m 1.333333   4.25
q3      f 4.333333   4.00
q4      m 1.666667   4.50
```

Now let us break the mean down by both workshop and gender. To keep our by factors in the form of a list (or data frame), we can use any one of these forms:

```
mydata[ c("workshop", "gender")]
```

or

```
mydata[ c(2, 3) ]
```

or, if you have attached the data frame,

```
data.frame( workshop, gender)
```

This starts to look messy, so let us put both our variable list and our factor list into character vectors:

```
myVars <- c("q1", "q2", "q3", "q4")

myBys  <- mydata[ c("workshop", "gender") ]
```

By using our character vectors as arguments for the by function, it is much easier to read. This time, let us use the range function to show that the by function can apply functions that return more than one value.

```
> myBYout <- by( mydata[myVars],
+   myBys, range, na.rm = TRUE )

> myBYout

workshop: R
gender: f
[1] 1 5
---------------------------------------------------
workshop: SAS
gender: f
```

```
[1] 1 4
---------------------------------------------------
workshop: R
gender: m
[1] 2 5
---------------------------------------------------
workshop: SAS
gender: m
[1] 4 5
```

That output is quite readable. Recall that when we did this same analysis using the `tapply` function, the results were in a form that were optimized for further analysis rather than communication. However, we can save the data to a data frame if we like. The approach it takes is most interesting. Let us see what type of object we have:

```
> mode(myBYout)
[1] "list"

> class(myBYout)
[1] "by"

> names(myBYout)
NULL
```

It is a list with a class of "by" and no names. Let us look at one of its components:

```
> myBYout[[1]]
[1] 1 5
```

This is the first set of ranges from the printout above. If we wanted to create a data frame from these, we could bind them into the rows of a matrix and then convert that to a data frame with

```
> myBYdata <- data.frame(
+   rbind( myBYout[[1]], myBYout[[2]],
+          myBYout[[3]], myBYout[[4]] )
+ )

> myBYdata

  X1 X2
1  1  5
2  1  4
3  2  5
4  4  5
```

That approach is easy to understand but not much fun to use if we had many more factor levels! Luckily the `do.call` function can call a function you choose once, on all of the components of a list, just as if you had entered them individually. That is quite different from the `lapply` function, which applies the function you choose repeatedly on each separate component. All we have to do is give it the function to feed the components into, `rbind` in this case, and the list name, byBYout:

```
> myBYdata <- data.frame( do.call(rbind, myBYout) )

> myBYdata

  X1 X2
1  1  5
2  1  4
3  2  5
4  4  5
```

10.13.1 Example Programs for By or Split-File Processing

SAS Program for By or Split-File processing

```
* Filename: By.sas ;
LIBNAME myLib  C:\myRfolder ;

PROC MEANS DATA=myLib.mydata;
  RUN;

PROC SORT DATA=myLib.mydata;
  BY gender;
  RUN;
PROC MEANS DATA=myLib.mydata;
  BY gender;
  RUN;

PROC SORT DATA=myLib.mydata;
  BY workshop gender;
  RUN;
PROC MEANS DATA=myLib.mydata;
  BY workshop gender;
  RUN;
```

SPSS Program for By or Split-File processing

```
* Filename: By.sps .
```

```
CD  C:\myRfolder .
GET FILE= mydata.sav .

DESCRIPTIVES
  VARIABLES=q1 q2 q3 q4
  /STATISTICS=MEAN STDDEV MIN MAX .

SORT CASES BY gender .
SPLIT FILE
  SEPARATE BY gender .
DESCRIPTIVES
  VARIABLES=q1 q2 q3 q4
  /STATISTICS=MEAN STDDEV MIN MAX .

SORT CASES BY workshop gender .
SPLIT FILE
  SEPARATE BY workshop gender .
DESCRIPTIVES
  VARIABLES=q1 q2 q3 q4
  /STATISTICS=MEAN STDDEV MIN MAX .
```

R Program for By or Split-File processing

```
# Filename: By.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
attach(mydata)
options(width=64)
mydata

# Get means of q variables for all observations.
mean( mydata[ c("q1", "q2", "q3", "q4") ] ,
       na.rm = TRUE)

# Now get means by gender.
myBYout <- by( mydata[ c("q1", "q2", "q3", "q4") ] ,
    mydata["gender"],
    mean,na.rm = TRUE)
myBYout
mode(myBYout)
class(myBYout)
myBYdata <- as.data.frame( (as.table(myBYout) ) )
myBYdata
```

```
# Get range by workshop and gender
myVars <- c("q1", "q2", "q3", "q4")
myBys  <- mydata[ c("workshop", "gender") ]
myBYout <- by( mydata[myVars],
  myBys, range, na.rm = TRUE )
myBYout

# Converting output to data frame.
mode(myBYout)
class(myBYout)
names(myBYout)
myBYout[[1]]

# A data frame the long way.
myBYdata <- data.frame(
  rbind(myBYout[[1]], myBYout[[2]],
        myBYout[[3]], myBYout[[4]])
)
myBYdata

# A data frame using do.call.
myBYdata <- data.frame( do.call( rbind, myBYout) )
myBYdata
mode(myBYdata)
class(myBYdata)
```

10.14 Removing Duplicate Observations

Duplicate observations frequently creep into data sets, especially those that are merged from various other data sets. One SAS approach is to use PROC SORT NODUPRECS to get rid of duplicates without examining them. The SPSS approach uses the menu choice, *Identify Duplicate Cases* to generate programming code that will identify or filter the observations. Of course SAS and SPSS are powerful enough to do either approach. We will use both the methods in R.

We will first see how to identify observations that are duplicates for every variable (NODUPRECS) and then find those who duplicate just key values (NODUPKEY).

10.14.1 Completely Duplicate Observations

First, let us create a data frame that takes the top two observations from mydata and appends them to the bottom with the `rbind` function:

```
> myDuplicates <- rbind(mydata, mydata[1:2, ])

> myDuplicates

   workshop gender q1 q2 q3 q4
1         R      f  1  1  5  1  <- We are copying
2       SAS      f  2  1  4  1  <- these two...
3         R      f  2  2  4  3
4       SAS   <NA> 3  1 NA  3
5         R      m  4  5  2  4
6       SAS      m  5  4  5  5
7         R      m  5  3  4  4
8       SAS      m  4  5  5  5
9         R      f  1  1  5  1  <- ...down here
10      SAS      f  2  1  4  1  <- as duplicates.
```

Next we will use the `unique` function to find and delete them:

```
> myNoDuplicates <- unique(myDuplicates)

> myNoDuplicates
  workshop gender q1 q2 q3 q4
1        R      f  1  1  5  1
2      SAS      f  2  1  4  1
3        R      f  2  2  4  3
4      SAS   <NA> 3  1 NA  3
5        R      m  4  5  2  4
6      SAS      m  5  4  5  5
7        R      m  5  3  4  4
8      SAS      m  4  5  5  5
```

So the `unique` function removed them but did not show them to us. In a more realistic data set, we would certainly not want to print the whole thing and examine the duplicates visually as we did above. However, knowing more about the duplicates might help us prevent them from creeping into our future analyses. Let us put the duplicates back and see what the `duplicated` function can do:

```
> myDuplicates <- rbind(mydata, mydata[1:2, ])

> myDuplicates$DupRecs <- duplicated(myDuplicates)

> myDuplicates

   workshop gender q1 q2 q3 q4 DupRecs
1         R      f  1  1  5  1   FALSE
```

```
2       SAS      f  2  1  4  1   FALSE
3         R      f  2  2  4  3   FALSE
4       SAS   <NA>  3  1 NA  3   FALSE
5         R      m  4  5  2  4   FALSE
6       SAS      m  5  4  5  5   FALSE
7         R      m  5  3  4  4   FALSE
8       SAS      m  4  5  5  5   FALSE
9         R      f  1  1  5  1    TRUE
10      SAS      f  2  1  4  1    TRUE
```

The `duplicated` function added the variable named *DupRecs* to our data frame. Its TRUE values show us that R has indeed located the *dup*licate *rec*ords. It is interesting to note that now we technically no longer have complete duplicates! The original first two records now have values of FALSE, whereas the last two, which up until now had been exact duplicates, have values of TRUE. So they have ceased to be exact duplicates! Therefore, the `unique` function would no longer identify the last two records. That is okay because now we will just get rid of those marked TRUE after we print a report of duplicate records.

```
> attach(myDuplicates)

> myDuplicates[DupRecs, ]

   workshop gender q1 q2 q3 q4 DupRecs
9         R      f  1  1  5  1    TRUE
10      SAS      f  2  1  4  1    TRUE
```

Finally, we will choose those not duplicated (i.e., !DupRecs) and drop the seventh variable, which is the TRUE/FALSE variable itself:

```
> myNoDuplicates <- myDuplicates[!DupRecs, -7 ]

> myNoDuplicates

  workshop gender q1 q2 q3 q4
1        R      f  1  1  5  1
2      SAS      f  2  1  4  1
3        R      f  2  2  4  3
4      SAS    <NA> 3  1 NA  3
5        R      m  4  5  2  4
6      SAS      m  5  4  5  5
7        R      m  5  3  4  4
8      SAS      m  4  5  5  5
```

Now our data are back to their original, duplicate-free state.

If I were doing this just for myself, I would have left the DupRecs variable as a vector outside the data frame. That would have saved me the need to attach the data frame (simplifying the selection) and later removing this variable. However, adding it to our small data frame made it more clear what it was doing.

10.14.2 Duplicate Keys

SAS also has a NODUPKEY option that eliminates records that have duplicate key values while allowing other values to differ. This approach uses the method in the section above, but applies it only to the key variables of workshop and gender.

Since we are now focusing on just workshop and gender, our original data set *already* contained duplicates, so our job now is to identify them.

I will first create a character vector containing the keys of interest:

```
> myKeys <- c("workshop", "gender")
```

Next, I will apply the `duplicated` function only to the part of the data frame that contains our keys:

```
> mydata$DupKeys <- duplicated(mydata[ ,myKeys])

> mydata

  workshop gender q1 q2 q3 q4 DupKeys
1        R      f  1  1  5  1   FALSE
2      SAS      f  2  1  4  1   FALSE
3        R      f  2  2  4  3    TRUE
4      SAS   <NA>  3  1 NA  3   FALSE
5        R      m  4  5  2  4   FALSE
6      SAS      m  5  4  5  5   FALSE
7        R      m  5  3  4  4    TRUE
8      SAS      m  4  5  5  5    TRUE
```

Now we see that only the first occurrence of each workshop–gender combination is considered a nonduplicate.

Using the DupKeys variable, you can now print the duplicated records or delete them using the same steps as we used previously for records that were complete duplicates.

10.14.3 Example Programs for Removing Duplicates

SAS Program for Removing Duplicates

```
* Filename: Duplicates.sas ;
```

```
LIBNAME myLib 'C:\myRfolder';

DATA mycopy; SET myLib.mydata;
Data lastTwo;
  SET myLib.mydata;
  IF ID GE 7;
  RUN;

DATA Duplicates;
  SET mycopy lastTwo;
  PROC PRINT; RUN;

PROC SORT NODUPREC DATA=Duplicates;
  BY id workshop gender q1-q4;
RUN;

PROC PRINT;
RUN;

PROC SORT NODUPKEY EQUALS DATA=mycopy;
  BY workshop gender;
RUN;

PROC PRINT DATA=mycopy;
RUN;
```

SPSS Program for Removing Duplicates

```
* Filename: Duplicates.sps .

CD 'C:\myRfolder'.
GET FILE='mydata.sav'.
* Identify Duplicate Cases.
SORT CASES BY workshop(A) gender(A)
  q2(A) q1(A) q3(A) q4(A) .
MATCH FILES /FILE = *
 /BY workshop gender q2 q1 q3 q4
 /FIRST = PrimaryFirst
 /LAST = PrimaryLast.
DO IF (PrimaryFirst).
+ COMPUTE MatchSequence = 1 - PrimaryLast.
ELSE.
+ COMPUTE MatchSequence = MatchSequence + 1.
END IF.
```

```
LEAVE MatchSequence.
FORMAT MatchSequence (f7).
COMPUTE InDupGrp = MatchSequence > 0.
SORT CASES InDupGrp(D).
MATCH FILES /FILE = *
 /DROP = PrimaryFirst InDupGrp MatchSequence.
VARIABLE LABELS
  PrimaryLast  Indicator of each last matching case as Primary .
VALUE LABELS PrimaryLast
  0  Duplicate ŚDuplicate Case  CaseŠ
  1  Primary ŚPrimary Case CaseŠ.
VARIABLE LEVEL PrimaryLast (ORDINAL).
FREQUENCIES VARIABLES = PrimaryLast .
```

R Program for Removing Duplicate Observations

```
# Filename: Duplicates.R

setwd("c:/myRfolder")
load("mydata.RData")
mydata

# Create some duplicates.
myDuplicates <- rbind(mydata, mydata[1:2, ])
myDuplicates

# Get rid of duplicates without seeing them.
myNoDuplicates <- unique(myDuplicates)
myNoDuplicates

# This checks for location of duplicates
# before getting rid of them.

myDuplicates <- rbind(mydata, mydata[1:2, ])
myDuplicates

myDuplicates$DupRecs <- duplicated(myDuplicates)
myDuplicates

# Print a report of just the duplicate records.
attach(myDuplicates)
myDuplicates[DupRecs, ]

# Remove duplicates and Duplicated variable.
myNoDuplicates <- myDuplicates[!DupRecs, -7 ]
```

```
myNoDuplicates

# Locate records with duplicate keys.
myKeys <- c("workshop", "gender")
mydata$DupKeys <- duplicated(mydata[ ,myKeys])
mydata
```

10.15 Selecting First or Last Observations per Group

When a data set contains groups, members within each group are often sorted in a useful order. For example, a company may have divisions divided into departments. Each department might have salary information for each person and a running total. So the last person's running total value would be the total for each department.

The SAS approach on this problem is quite flexible. Simply saying,

```
DATA mydata;
SET  mydata;
BY workshop gender;
```

creates four temporary variables, first.workshop, first.gender, last.workshop and last.gender. These all have values of 1 when true and 0 when false. These variables vanish at the end of the data step unless you assign them to regular variables, but that is usually not necessary.

SPSS uses a very similar approach in the MATCH FILES procedure. Normally, you think of MATCH FILES as requiring two files to join, but you can use it in this case with only one file. It creates only a single FIRST or LAST variable that is saved to the data set. Be careful with this approach as it subsets the main file, so you need to save it to a new name.

SPSS can also view this problem as an aggregation. Unlike SAS, its AGGREGATE procedure has FIRST and LAST functions. This works fine for just a few variables, but since it requires naming every variable you wish to save, it is not very useful for saving many variables. The example SPSS program at the end of this section demonstrates both approaches.

The R approach to this problem demonstrates R's extreme flexibility. It does not have a function aimed directly at this problem. However, it is easy to create one using several other functions. We have seen the `head` function print the top few observations of a data frame. The `tail` function does the same for the last few. We have also used the `by` function to apply a function to groups within a data frame. We can use the `by` function to apply the `head` function to get the first observation in each group or use the `tail` function to get the last. Since the `head` and `tail` functions both have an "`n=`" argument, we can not only use `n = 1` to get the single first or last, but we could also use `n = 2` to get the first two or last two observations per group, and so on.

The first record per group is often of interest because we may need to initialize values for each group, e.g., when calculating cumulative totals. The last record per group is often of interest since it contains the final values. We will look at an example how to select the last observation per group. The idea readily extends to the first record(s) per group.

First, we will read our data and create an ID variable based on the row names. We do not need ID for our first example, but it will be helpful in the second.

```
> setwd("c:/myRfolder")
> load(file = "mydata.RData")

> mydata$id <- row.names(mydata)

> mydata
  workshop gender q1 q2 q3 q4 id
1        R      f  1  1  5  1  1
2      SAS      f  2  1  4  1  2
3        R      f  2  2  4  3  3
...
```

Next, we will put our *by* variables into a data frame. By using workshop and then gender, we will soon be selecting the last male in each workshop.

```
myBys <- data.frame(mydata$workshop, mydata$gender)
```

Next, we use the `by` function to apply the `tail` function to mydata by workshop and gender. We are saving the result to mylast, which is in the form of a list:

```
> mylastList <- by(mydata, myBys, tail, n = 1)

> mylastList

mydata.workshop: R
mydata.gender: f
  workshop gender q1 q2 q3 q4 id
3        R      f  2  2  4  3  3
------------------------------------------------------------
mydata.workshop: SAS
mydata.gender: f
  workshop gender q1 q2 q3 q4 id
2      SAS      f  2  1  4  1  2
------------------------------------------------------------
mydata.workshop: R
mydata.gender: m
  workshop gender q1 q2 q3 q4 id
```

```
7        R      m  5  3  4  4  7
------------------------------------------------------------
mydata.workshop: SAS
mydata.gender: m
  workshop gender q1 q2 q3 q4 id
8      SAS      m  4  5  5  5  8
```

We would like to put this into a data frame by combining all of the vectors from that list in the form of rows. The `do.call` function does this. It essentially takes all of the elements of a list and feeds them into a single call to the function you choose. In this case, that is the `rbind` function:

```
> mylastDF <- do.call(rbind, mylastList)

> mylastDF

  workshop gender q1 q2 q3 q4 id
3        R      f  2  2  4  3  3
2      SAS      f  2  1  4  1  2
7        R      m  5  3  4  4  7
8      SAS      m  4  5  5  5  8
```

That single call to the `do.call` function is the equivalent of all this:

```
mylastDF <- rbind(as.list(mylastList)[[1]],
                  as.list(mylastList)[[2]],
                  as.list(mylastList)[[3]],
                  as.list(mylastList)[[4]])
```

If we had hundreds of groups, the `do.call` function would be a big time saver!

At this point in my work, I am usually finished. However, some people need a variable in the original data frame that indicates which record per group is last (or first). SAS would name this variable “last.gender”. SPSS similarly uses the `LAST=` option to create such a variable. You can create such a variable by simply adding a constant 1 to mylastDF, then merging mylastDF back to the original data frame. Here is how to create a constant with a value of 1:

```
> mylastDF$lastGender <- rep(1, nrow(mylastDF) )

> mylastDF
  workshop gender q1 q2 q3 q4 id lastGender
3        R      f  2  2  4  3  3          1
2      SAS      f  2  1  4  1  2          1
7        R      m  5  3  4  4  7          1
8      SAS      m  4  5  5  5  8          1
```

The `rep` function call used only two arguments, the value to *rep*eat and the number of times to repeat it. We could have specified 4 there but in a larger data frame you may not know how many rows you have so using `nrow(mylastDF)` is more general.

Next we save just the variables id and lastGender, and merge the two data frames by id. The `all = TRUE` argument tells it to save records even if the id values are not in both data frames:

```
> mylastDF2 <- mylastDF[ c("id", "lastGender") ]

> mydata2 <- merge(mydata, mylastDF2, by = "id", all = TRUE)

> mydata2

  id workshop gender q1 q2 q3 q4 lastGender
1  1        R      f  1  1  5  1         NA
2  2      SAS      f  2  1  4  1          1
3  3        R      f  2  2  4  3          1
4  4      SAS   <NA>  3  1 NA  3         NA
...
```

Notice the number 1 appears when it is the last workshop and gender combination but NA elsewhere. To match what SAS and SPSS do, we need for NA to be zero instead. That is easy to change:

```
> mydata2$lastGender[ is.na(mydata2$lastGender) ] <- 0

> mydata2

  id workshop gender q1 q2 q3 q4 lastGender
1  1        R      f  1  1  5  1          0
2  2      SAS      f  2  1  4  1          1
3  3        R      f  2  2  4  3          1
4  4      SAS   <NA>  3  1 NA  3          0
5  5        R      m  4  5  2  4          0
6  6      SAS      m  5  4  5  5          0
7  7        R      m  5  3  4  4          1
8  8      SAS      m  4  5  5  5          1
```

Recall that the form `var==NA` can never be true, so we use the form `is.na(var)` to find missing values. Now you have your lastGender variable and can use that in other calculations.

10.15.1 Example Programs for Selecting Last Observation per Group

SAS Program for Selecting Last Observation per Group

```
* Filename: FirstLastObs.sas ;
```

```
LIBNAME myLib C:\myRfolder ;
PROC SORT DATA=sasuser.mydata;
  BY workshop gender;
RUN;

DATA sasuser.mylast;
  SET  sasuser.mydata;
  BY   workshop gender;
  IF   last.gender;
RUN;

PROC PRINT; RUN;
```

SPSS Program for Selecting Last Observation per Group

```
* Filename: FirstLastObs.sps .

CD C:\myRfolder .

* Match files method.
GET FILE= mydata.sav .
SORT CASES BY workshop gender.
MATCH FILES FILE=* /By workshop gender /LAST=lastgender.
SELECT IF lastgender.
LIST.
SAVE OUTFILE= mylast.sav .

* Aggregation method.
SORT CASES BY workshop gender.
AGGREGATE /OUTFILE= C:\mylast.sav
/BREAK workshop gender
/q1 = LAST(q1)
/q2 = LAST(q2)
/q3 = LAST(q3)
/q4 = LAST(q4).

* Using LIST here would display original file.
GET FILE= mylast.sav .
DATASET NAME DataSet5 WINDOW=FRONT.
LIST.
```

R Program for Selecting Last Observation per Group

```
# Filename: FirstLastObs.R
```

```
setwd("c:/myRfolder")
load(file = "mydata.RData")
mydata$id <- row.names(mydata)
mydata

myBys <- data.frame(mydata$workshop, mydata$gender)
mylastList <- by(mydata, myBys, tail, n = 1)
mylastList

#Back into a data frame:
mylastDF <- do.call(rbind, mylastList)
mylastDF

# Another way to create the data frame:
mylastDF <- rbind(mylastList[[1]],
                  mylastList[[2]],
                  mylastList[[3]],
                  mylastList[[4]])
mylastDF

# Generating just an indicator variable
mylastDF$lastGender <- rep(1, nrow(mylastDF) )
mylastDF

mylastDF2 <- mylastDF[ c("id", "lastGender") ]
mydata2 <- merge(mydata, mylastDF2, by = "id", all = TRUE )
mydata2

mydata2$lastGender[ is.na(mydata2$lastGender) ] <- 0
mydata2
```

10.16 Transposing or Flipping Data Sets

When data arrive from a package not designed for data analysis, it is occasionally entered "sideways" with the variables in the rows and the observations or cases in the columns.

Let us start with the easy case: data that are all of one type. It would usually be all numeric data but it could be all character as well. To simulate a data set that needs to be transposed, I will simply strip off the q variables:

```
> mydata

> myQs <- c("q1", "q2", "q3", "q4")
> myQdf <- mydata[ ,myQs]

> myQdf
  q1 q2 q3 q4
1  1  1  5  1
2  2  1  4  1
3  2  2  4  3
4  3  1 NA  3
5  4  5  2  4
6  5  4  5  5
7  5  3  4  4
8  4  5  5  5
```

Now that we have an all-numeric data frame, we can use the `t` function to *t*ranspose it:

```
> myFlipped <- t(myQdf)

> myFlipped

   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
q1    1    2    2    3    4    5    5    4
q2    1    1    2    1    5    4    3    5
q3    5    4    4   NA    2    5    4    5
q4    1    1    3    3    4    5    4    5
```

Now this is the type of data we might have read in from some other source, which we might need to transpose or flip before we can use it. Before we do that, let us see what its class is:

```
> class(myFlipped)

[1] "matrix"
```

We see that the `t` function coerced the data frame into a matrix. So to flip it back into a form that is useful for data analysis, we can nest the call to the `t` function within a call to the `as.data.frame` function:

```
> myFixed <- as.data.frame( t(myFlipped) )

> myFixed

  q1 q2 q3 q4
1  1  1  5  1
2  2  1  4  1
3  2  2  4  3
```

```
4  3  1 NA  3
5  4  5  2  4
6  5  4  5  5
7  5  3  4  4
8  4  5  5  5
```

That was easy. However, eventually you are going to get a data set that contains character data. That presents a much more challenging situation. We will simulate that by flipping our whole data frame:

```
> myFlipped <- t(mydata)

> myFlipped

         [,1] [,2]  [,3] [,4]  [,5] [,6]  [,7] [,8]
workshop "R"  "SAS" "R"  "SAS" "R"  "SAS" "R"  "SAS"
gender   "f"  "f"   "f"  NA    "m"  "m"   "m"  "m"
q1       "1"  "2"   "2"  "3"   "4"  "5"   "5"  "4"
q2       "1"  "1"   "2"  "1"   "5"  "4"   "3"  "5"
q3       " 5" " 4"  " 4" NA    " 2" " 5"  " 4" " 5"
q4       "1"  "1"   "3"  "3"   "4"  "5"   "4"  "5"
```

The t functions coerces a data frame into a matrix, and all the elements of a matrix must be of the same mode: numeric or character. Therefore, myFlipped is a character matrix. That is obvious from the quotes around each of its elements. So these are the data as we might have imported them from some other source and now we need to transpose them to make them useful:

```
> myFixed <- t(myFlipped)

> myFixed

     workshop gender q1  q2  q3   q4
[1,] "R"      "f"    "1" "1" " 5" "1"
[2,] "SAS"    "f"    "2" "1" " 4" "1"
[3,] "R"      "f"    "2" "2" " 4" "3"
[4,] "SAS"    NA     "3" "1" NA   "3"
[5,] "R"      "m"    "4" "5" " 2" "4"
[6,] "SAS"    "m"    "5" "4" " 5" "5"
[7,] "R"      "m"    "5" "3" " 4" "4"
[8,] "SAS"    "m"    "4" "5" " 5" "5"
```

That gets it back into the general form we need in R, but it is still a character matrix. We can convert it to a data frame and check its structure with the str function:

```
> myFixed <- data.frame(myFixed)
> str(myFixed)

 data.frame :   8 obs. of  6 variables:
  $ workshop: Factor w/ 2 levels "R","SAS": 1 2 1 2 1 2 1 2
  $ gender  : Factor w/ 2 levels "f","m": 1 1 1 NA 2 2 2 2
  $ q1      : Factor w/ 5 levels "1","2","3","4",..: 1 2 2 3 4 5
  $ q2      : Factor w/ 5 levels "1","2","3","4",..: 1 1 2 1 5 4
  $ q3      : Factor w/ 3 levels " 2"," 4"," 5": 3 2 2 NA 1 3 2 3
  $ q4      : Factor w/ 4 levels "1","3","4","5": 1 1 2 2 3 4 3 4
```

The `data.frame` function converts all character data to factors (by default). Therefore, we need to convert the q variables to numeric by applying the `as.numeric` function:

```
> myQs <- c("q1", "q2", "q3", "q4")

> myFixed[ ,myQs] <-
+   lapply(myFixed[ ,myQs], as.numeric)

> str(myFixed)

 data.frame :   8 obs. of  6 variables:
  $ workshop: Factor w/ 2 levels "R","SAS": 1 2 1 2 1 2 1 2
  $ gender  : Factor w/ 2 levels "f","m": 1 1 1 NA 2 2 2 2
  $ q1      : num  1 2 2 3 4 5 5 4
  $ q2      : num  1 1 2 1 5 4 3 5
  $ q3      : num  3 2 2 NA 1 3 2 3
  $ q4      : num  1 1 2 2 3 4 3 4
```

Now our data frame is back in its original form and ready to analyze.

10.16.1 Example Programs for Transposing or Flipping Data Sets

SAS Program for Transposing or Flipping Data Sets

```
* Filename: Transpose.sas ;

LIBNAME myLib  C:\myRfolder ;

PROC TRANSPOSE DATA=myLib.mydata OUT=mycopy;
PROC PRINT; RUN;

PROC TRANSPOSE DATA=mycopy OUT=myFixed;
PROC PRINT; RUN;
```

SPSS Program for Transposing or Flipping Data Sets

```
* Filename: Transpose.sps.

CD  C:\myRfolder .
GET FILE= mydata.sav .
DATASET NAME DataSet1 WINDOW=FRONT.

FLIP VARIABLES=id workshop gender q1 q2 q3 q4.

FLIP VARIABLES=var001 var002 var003 var004
  var005 var006 var007 var008
  /NEWNAMES=CASE_LBL.
```

R Program for Transposing or Flipping Data Sets

```
# Filename: Transpose.R

setwd("c:/myRfolder")
load("mydata.RData")
mydata

myQs <- c("q1", "q2", "q3", "q4")
myQdf <- mydata[ ,myQs]
myQdf
myFlipped <- t(myQdf)
myFlipped
class(myFlipped)  # coerced into a matrix!
myFixed <- as.data.frame( t(myFlipped) )
myFixed

# Again, but with all the data
options(width = 60)
myFlipped <- t(mydata)
myFlipped

myFixed <- t(myFlipped)
myFixed
myFixed <- data.frame(myFixed)
str(myFixed)

myQs <- c("q1", "q2", "q3", "q4")
myFixed[ ,myQs] <-
  lapply(myFixed[ ,myQs], as.numeric)
str(myFixed)
```

10.17 Reshaping Variables to Observations and Back

A common data management problem is reshaping data from "wide" format to "long" or vice versa. If we assume our variables q1, q2, q3, and q4 are the same item measured at four different times, we will have the standard wide format for repeated-measures data.

Converting this to the long format consists of writing out four records, each of which has just one measure – often called simply Y – and a counter variable, often called time, that goes 1, 2, 3, and 4. So in the simplest case, just two variables, Y and time, could replace dozens of variables. Going from long to wide is just the reverse.

SPSS makes this process very easy to do with its *Restructure Data Wizard*. It prompts you for the exact structure of your data and your goal. Then it generates the necessary SPSS program and executes it.

SAS can do this in at least two ways, but PROC TRANSPOSE is probably the easiest to use.

In R, Wickham's `reshape2` package is quite powerful and easy to use. The "2" in its name indicates that it is version 2 of the `reshape` package and it changed in ways that make it incompatible with the older version. The original `reshape` package is still available for download from CRAN so older programs will still run.

The `reshape2` package uses the analogy of *melting* your data so that you can *cast* it into a different mold. In addition to reshaping, the package makes quick work of a wide range of aggregation problems.

To work through an example, let us change some of the variable names to indicate they are measures across time:

```
> library("reshape2")

> myChanges <- c(
+   q1 = "time1",
+   q2 = "time2",
+   q3 = "time3",
+   q4 = "time4")

> mydata <- rename(mydata, myChanges)
```

For details on changing variable names, see Sect. 10.6.

Next we will add a subject variable as a factor and look at our data so far:

```
> mydata$subject <- factor(1:8)

> mydata
  workshop gender time1 time2 time3 time4 subject
1        R      f     1     1     5     1       1
2      SAS      f     2     1     4     1       2
```

```
...
7        R      m     5     3     4     4       7
8      SAS      m     4     5     5     5       8
```

Now we will “melt” the data into the long form. By default, the `melt` function will assume all factors and character variables are ID variables and all numeric variables are the values you want to stretch into the long format. That was why I created subject as a factor. Now we can melt our data set with:

```
> mylong <- melt(mydata)

Using workshop, gender, subject as id variables

> mylong

   workshop gender subject variable value
1         R      f       1    time1     1
2       SAS      f       2    time1     2
3         R      f       3    time1     2
4       SAS   <NA>       4    time1     3
...
29        R      m       5    time4     4
30      SAS      m       6    time4     5
31        R      m       7    time4     4
32      SAS      m       8    time4     5
```

The resulting data set has 32 records. The variables workshop, gender, and subject look much as they did before, but with many repeated values. The new column named “variable” stores the original variable names and “value” stores their values.

The `melt` function printed a message stating what it considered ID variables, which works fine in our case. If instead we had numeric ID variables, we could have called the `melt` function by supplying these arguments:

```
mylong <- melt(mydata,
  id.vars      = c("subject", "workshop", "gender"),
  measure.vars = c("time1", "time2", "time3", "time4"),
  value.name   = "value")
```

With our data the result would be identical.

Now let us cast the data back into the wide format:

```
> mywide <- dcast(mylong, subject+workshop+gender ~ variable)

> mywide

  subject workshop gender time1 time2 time3 time4
```

```
1       1        R      f     1     1     5     1
2       2      SAS      f     2     1     4     1
3       3        R      f     2     2     4     3
4       4      SAS   <NA>     3     1    NA     3
5       5        R      m     4     5     2     4
6       6      SAS      m     5     4     5     5
7       7        R      m     5     3     4     4
8       8      SAS      m     4     5     5     5
```

The dcast function needs only two arguments: the data to reshape and a formula. The formula has the ID variables on the left side separated by plus signs, then a tilde, "~", and the variable that will contain the new variables' values. The "d" in dcast means that it will create a *d*ata frame. The acast function works the same but creates *a*rrays (or vectors or matrices).

Let us now do a more complicated example, one with two levels of repeats. We will assume that we have data from an experiment that used two teaching methods and two levels of each method. Let us read the data again and rename our variables accordingly:

```
> load("mydata.RData")

> mydata$subject <- factor(1:8)

> library("reshape2")

> myChanges <- c(
+   q1 = "M1_L1",
+   q2 = "M1_L2",
+   q3 = "M2_L1",
+   q4 = "M2_L2")

> mydata <- rename(mydata, myChanges)

> mydata
  workshop gender M1_L1 M1_L2 M2_L1 M2_L2 subject
1        R      f     1     1     5     1       1
2      SAS      f     2     1     4     1       2
...
7        R      m     5     3     4     4       7
8      SAS      m     4     5     5     5       8
```

Now we can melt the data. All our ID variables are factors, so I will not bother specifying any arguments:

```
> mylong2 <- melt(mydata)
```

```
Using workshop, gender as id variables

> mylong2

   workshop gender subject variable value
1         R      f       1    M1_L1     1
2       SAS      f       2    M1_L1     2
3         R      f       3    M1_L1     2
4       SAS   <NA>       4    M1_L1     3
...
29        R      m       5    M2_L2     4
30      SAS      m       6    M2_L2     5
31        R      m       7    M2_L2     4
32      SAS      m       8    M2_L2     5
```

If we had started with data in this long form, we would have had two variables specifying the method and level. So before casting the data back into wide form, let us create those variables using the `rep` function that is described in detail in Sect. 12.3:

```
> mylong2$method <- rep( c("M1", "M2"), each=16, times=1)

> mylong2$level  <- rep( c("L1", "L2"), each=8,  times=2)

> mylong2

   workshop gender subject variable value method level
1         R      f       1    M1_L1     1     M1    L1
2       SAS      f       2    M1_L1     2     M1    L1
...
9         R      f       1    M1_L2     1     M1    L2
10      SAS      f       2    M1_L2     1     M1    L2
...
17        R      f       1    M2_L1     5     M2    L1
18      SAS      f       2    M2_L1     4     M2    L1
...
31        R      m       7    M2_L2     4     M2    L2
32      SAS      m       8    M2_L2     5     M2    L2
```

In SAS or SPSS you would create such group ID variables using DO loops. Just as it is critical to get your I and J counters correct there, you must also be very careful using tools like `rep` to create variables. It is easy to make errors using the `each` and `times` arguments. Only after careful checking should you take the next step of dropping the variable named “variable”:

```
mylong2$variable <- NULL
```

Now we are ready to cast the data into wide form. This uses dcast as before but this time we have two variables on the right side of the formula:

```
mywide2 <- dcast(mylong2,
  gender + workshop + subject ~ method + level)
```

We can see what the result looks like by simply printing it.

```
> mywide2

  gender workshop subject M1_L1 M1_L2 M2_L1 M2_L2
1      f        R       1     1     1     5     1
2      f        R       3     2     2     4     3
3      f      SAS       2     2     1     4     1
4      m        R       5     4     5     2     4
5      m        R       7     5     3     4     4
6      m      SAS       6     5     4     5     5
7      m      SAS       8     4     5     5     5
8   <NA>      SAS       4     3     1    NA     3
```

The only difference between the data shown here and their original form is that the rows are no longer in order by subject. We can easily fix that with:

```
mywide2 <- mywide2[ order(mywide2$subject), ]
```

See Section 10.18 for details on sorting data frames.

10.17.1 Summarizing/Aggregating Data Using reshape2

Once a data set is in long form, you can easily use reshape to get summary statistics. We just saw that we could cast the data from long into wide format with dcast. But what if the dcast formula was missing a variable or two? What would dcast do then? It would have more than one number per cell to cast, so it would have to call a function on them. The default function it uses is length. The reshape2 package can aggregate by any function that returns a single value.

Let us cast these data using the mean function. I will not bother to save these data to a data frame, though it would be easy to do so. First, let us drop the subject variable from our formula to see what the means across all subjects look like:

```
> dcast(mylong2, gender + workshop ~ method + level,
+   mean, na.rm = TRUE)

  gender workshop M1_L1 M1_L2 M2_L1 M2_L2
1      f        R   1.5   1.5   4.5     2
2      f      SAS   2.0   1.0   4.0     1
```

```
3       m         R   4.5   4.0   3.0     4
4       m       SAS   4.5   4.5   5.0     5
5    <NA>       SAS   3.0   1.0   NaN     3
```

Note that I simply appended `mean` onto the end of the function's arguments. The actual name of that argument position is `fun.aggregate`, but people generally do not bother to type it. Next let us drop the level variable and see what happens:

```
> dcast(mylong2, gender + workshop ~ method,
+   mean, na.rm = TRUE)

  gender workshop   M1   M2
1      f        R 1.50 3.25
2      f      SAS 1.50 2.50
3      m        R 4.25 3.50
4      m      SAS 4.50 5.00
5   <NA>      SAS 2.00 3.00
```

Next, I will drop the method variable from the formula. When you have no variables to put on one side of the formula or the other, you use the "`.`" symbol:

```
> dcast(mylong2, gender + workshop ~ . ,
+   mean, na.rm = TRUE)

  gender workshop       NA
1      f        R 2.375000
2      f      SAS 2.000000
3      m        R 3.875000
4      m      SAS 4.750000
5   <NA>      SAS 2.333333
```

Note that `dcast` can no longer come up with a good variable name for the mean. Finally, let us remove gender, leaving only workshop:

```
> dcast(mylong2, workshop ~ . , mean, na.rm = TRUE)

  workshop       NA
1        R 3.125000
2      SAS 3.533333
```

If you need to select part of a data frame before casting it, you can do so using `dcast`'s `subset` argument.

As we have seen with the `reshape2` package and others, R's ability to extend its power with packages can be very useful. However, it also occasionally leads to confusion among names. R comes with a *function* named `reshape`, and the `Hmisc` package has one named `reShape` (note the capital "S"). They both do similar tasks but are not as flexible as the reshape2 *package*.

10.17.2 Example Programs for Reshaping Variables to Observations and Back

To save space, the SAS and SPSS programs just perform the first transformation from wide to long and back.

SAS Program for Reshaping Data

```
* Filename: Reshape.sas ;

LIBNAME myLib  C:\myRfolder ;

* Wide to long;
PROC TRANSPOSE DATA=mylib.mydata
  OUT=myLib.mylong;
  VAR q1-q4;
  BY id workshop gender;
PROC PRINT;
RUN;
DATA mylib.mylong;
  SET mylib.mylong( rename=(COL1=value) );
  time=INPUT( SUBSTR( _NAME_, 2) , 1.);
  DROP _NAME_;
RUN;
PROC PRINT;
RUN;

* Long to wide;
PROC TRANSPOSE DATA=mylib.mylong
  OUT=myLib.mywide PREFIX=q;
  BY id workshop gender;
  ID time;
  VAR value;
RUN;
DATA mylib.mywide;
  SET mylib.mywide(DROP=_NAME_);
RUN;
PROC PRINT;
RUN;
```

SPSS Program for Reshaping Data

```
* Filename: Reshape.sps .

CD  C:\myRfolder .
```

```
GET FILE= mydata.sav .
* Wide to long.
VARSTOCASES  /MAKE Y FROM q1 q2 q3 q4
 /INDEX = Question(4)
 /KEEP =  id workshop gender
 /NULL = KEEP.
LIST.
SAVE OUTFILE= mywide.sav .
* Long to wide.
GET FILE= mywide.sav .
CASESTOVARS
 /ID = id workshop gender
 /INDEX = Question
 /GROUPBY = VARIABLE.
LIST.
SAVE OUTFILE= mylong.sav .
```

R Program for Reshaping Data

```
# Filename: Reshape.R

setwd("c:/myRfolder")
load("mydata.RData")

library("reshape2")
myChanges <- c(
  q1 = "time1",
  q2 = "time2",
  q3 = "time3",
  q4 = "time4")
mydata <- rename(mydata, myChanges)
mydata$subject <- factor(1:8)
mydata

# Reshaping from wide to long
library("reshape2")  # Just a reminder
mylong <- melt(mydata)
mylong

# Again, specifying arguments
mylong <- melt(mydata,
  id.vars      = c("subject", "workshop", "gender"),
  measure.vars = c("time1", "time2", "time3", "time4"),
  value.name   = "variable")
mylong
```

```
# Reshaping from long to wide
mywide <- dcast(mylong,
  subject + workshop + gender ~ variable)
mywide

# ---Two Time Variables---

load("mydata.RData")
mydata$subject <- factor(1:8)
library("reshape2")
myChanges <- c(
  q1 = "M1_L1",
  q2 = "M1_L2",
  q3 = "M2_L1",
  q4 = "M2_L2")
mydata <- rename(mydata, myChanges)
mydata

library("reshape2")  # Just a reminder
mylong2 <- melt(mydata)
mylong2

# Same thing with arguments specified
mylong2 <- melt(mydata,
  id.vars      = c("subject", "workshop", "gender"),
  measure.vars = c("M1_L1", "M1_L2", "M2_L1", "M2_L2"),
  value.name   = "value")
mylong2

mylong2$method <- rep( c("M1", "M2"), each=16, times=1)
mylong2$level  <- rep( c("L1", "L2"), each=8,  times=2)
mylong2
mylong2$variable <- NULL

# Reshape to wide
mywide2 <- dcast(mylong2,
  gender + workshop + subject ~ method + level)
mywide2
mywide2[ order(mywide2$subject), ]

# Aggregation via reshape

dcast(mylong2, gender + workshop ~ method + level,
  mean, na.rm = TRUE)
```

```
dcast(mylong2, gender + workshop ~ method,
  mean, na.rm = TRUE)
dcast(mylong2, workshop ~ .,
  mean, na.rm = TRUE)
```

10.18 Sorting Data Frames

Sorting is one of the areas in which R differs most from SAS and SPSS. In SAS and SPSS, sorting is a critical prerequisite for three frequent tasks:

1. Doing the same analysis repeatedly for different groups. In SAS this is called BY processing. SPSS calls it SPLIT–FILE processing.
2. Calculating summary statistics for each group in the SAS SUMMARY procedure (the similar SPSS AGGREGATE command does not require sorted data).
3. Merging files matched on the sorted variables such as ID.

As we have seen, R does not need for data to be sorted for any of these tasks. Still, sorting is useful in a variety of contexts.

R has a function named `sort`, but it sorts vectors or factors individually. While that can be occasionally useful, it is more often disastrous. For variables in a data frame, it breaks the relationship between row and observation, essentially destroying the data frame!

The function that R uses to sort data frames is named `order`. It first determines the order that the rows would be in if sorted and then applies them to do the sort.

Consider the names Ann, Eve, Carla, Dave, and Bob. They are almost sorted in ascending order. Since the number of names is small, it is easy to determine the order that the names would require to sort them. We need the first name, Ann, followed by the fifth name, Bob, followed by the third name, Carla, the fourth name, Dave, and, finally, the second name, Eve. The `order` function would get those index values for us: 1, 5, 3, 4, 2.

To understand how these index values will help us sort, let us review briefly how data frame subscripts work. One way to select rows from a data frame is to use the form `mydata[rows, columns]`. If you leave them all out, as in `mydata[ , ]`, then you will get all rows and all columns. You can select the first four records with

```
> mydata[ c(1, 2, 3, 4), ]

  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
2  2        2      f  2  1  4  1
3  3        1      f  2  2  4  3
4  4        2   <NA>  3  1 NA  3
```

We can select them in reverse order with

```
> mydata[ c(4, 3, 2, 1), ]

  id workshop gender q1 q2 q3 q4
4  4        2   <NA>  3  1 NA  3
3  3        1      f  2  2  4  3
2  2        2      f  2  1  4  1
1  1        1      f  1  1  5  1
```

Now let us create a variable to store the order of the observations if sorted by workshop:

```
> myW <- order( mydata$workshop )

> myW
[1] 1 3 5 7 2 4 6 8
```

We can use this variable as the row subscript to mydata to see it sorted by workshop:

```
> mydata[myW, ]

  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
3  3        1      f  2  2  4  3
5  5        1      m  4  5  2  4
7  7        1      m  5  3  4  4
2  2        2      f  2  1  4  1
4  4        2   <NA>  3  1 NA  3
6  6        2      m  5  4  5  5
8  8        2      m  4  5  5  5
```

You can reverse the order of a sort calling the `rev` function:

```
> mydata[ rev(myW), ]
  workshop gender q1 q2 q3 q4
8        2      m  4  5  5  5
6        2      m  5  4  5  5
4        2   <NA>  3  1 NA  3
2        2      f  2  1  4  1
7        1      m  5  3  4  4
5        1      m  4  5  2  4
3        1      f  2  2  4  3
1        1      f  1  1  5  1
```

The `order` function is one of the few R functions that allow you to specify multiple variables without combining them in some way, like into a vector with the c function. So we can create an order variable to sort the data by gender and then workshop within gender with the following function call (GW stands for Gender then Workshop):

```
> myGW <- order( mydata$gender, mydata$workshop )

> mydata[myGW, ]

  id workshop gender q1 q2 q3 q4
1  1        1      f  1  1  5  1
3  3        1      f  2  2  4  3
2  2        2      f  2  1  4  1
5  5        1      m  4  5  2  4
7  7        1      m  5  3  4  4
6  6        2      m  5  4  5  5
8  8        2      m  4  5  5  5
4  4        2   <NA>  3  1 NA  3
```

The default order is ascending (small to large). To reverse this, place the minus sign before any variable. However, this only works with numeric variables, and we have been sorting by factors. We can use the as.numeric function to extract the numeric values that are a behind-the-scenes part of any factor:

```
> myDGW <- order(
+   -as.numeric(mydata$gender),
+    mydata$workshop
+ )

> mydata[ myDGW, ]

  workshop gender q1 q2 q3 q4
5        1      m  4  5  2  4
7        1      m  5  3  4  4
6        2      m  5  4  5  5
8        2      m  4  5  5  5
1        1      f  1  1  5  1
3        1      f  2  2  4  3
2        2      f  2  1  4  1
4        2   <NA>  3  1 NA  3
```

This time we see males sorted before females. The “D” in myDGW stands for Descending.

The as.numeric function is also helpful when sorting by numbers that are stored as character data. By default, the row names of a data frame are numbers stored as characters. The sort order of character values of: “1”, “2”, “3”, “100”, “1000” are 1, 100, 1000, 2, 3! Sorting by

```
as.numeric( row.names(mydata) )
```

will sort in the usual numeric order.

While SAS and SPSS view missing values as the smallest values to sort, R simply places them last. Recall from our discussion of missing values that R does not view NAs as large or small. You can use the argument `na.last = FALSE` to cause R to place NAs first. You can also remove records with missing values by setting `na.last=NA`.

Since it is so easy to create variables in which to store your various order indices, you do not need to store the whole data frame in sorted form to have easy access to it. However, if you want to, you can save the data frame in sorted form by using

```
mydataSorted <- mydata[ myDGW, ]
```

10.18.1 Example Programs for Sorting Data Sets

SAS Program for Sorting Data

```
* Filename: Sort.sas ;

LIBNAME myLib  \myRfolder ;

PROC SORT  DATA = myLib.mydata;
  BY workshop;
RUN;
PROC PRINT DATA = myLib.mydata;
RUN;

PROC SORT  DATA = myLib.mydata;
  BY gender workshop;
RUN;
PROC PRINT DATA = myLib.mydata;
RUN;

PROC SORT  DATA = myLib.mydata;
  BY descending gender workshop ;
RUN;
PROC PRINT DATA = myLib.mydata;
RUN;
```

SPSS Program for Sorting Data

```
* Filename: Sort.sps .
```

```
CD  \myRfolder .
GET FILE =  mydata.sav .

SORT CASES BY workshop (A).
LIST.

SORT CASES BY gender (A) workshop (A).
LIST.

SORT CASES BY  gender (D) workshop (A).
LIST.
```

R Program for Sorting Data

```
# Filename: Sort.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
mydata

# Show first four observations in order.
mydata[ c(1, 2, 3, 4), ]

# Show them in reverse order.
mydata[ c(4, 3, 2, 1), ]

# Create order variable for workshop.
myW <- order( mydata$workshop )
myW
mydata[ myW, ]

# Create order variable for gender then workshop.
myGW <- order( mydata$gender, mydata$workshop )
myGW
mydata[ myGW, ]

# Create order variable for
# descending (-) workshop then gender
myWdG <- order( -mydata$workshop, mydata$gender )
myWdG

# Print data in WG order.
mydata[ myWdG, ]

# Save data in WdG order.
```

```
mydataSorted <- mydata[ myWdG,  ]
mydataSorted
```

10.19 Converting Data Structures

In SAS and SPSS, there is only one data structure – the data set. Within that, there is only one structure, the variable. It seems absurdly obvious, but we need to say it: All SAS and SPSS procedures accept variables as input. Of course, you have to learn that putting a character variable in where a numeric one is expected causes an error. Occasionally, a character variable contains numbers and we must convert them. Putting a categorical variable in where a continuous variable belongs may not yield an error message, but perhaps it should. SPSS has added variable classification methods (nominal, ordinal, or scale) to help you choose the correct analyses and graphs.

As we have seen, R has several data structures, including vectors, factors, data frames, matrices, and lists. For many functions (what SAS/SPSS call procedures), R can automatically provide output optimized for the data structure you give it. Said more formally, *generic functions apply different methods to different classes of objects.* So to control a function, you have to know several things:

1. The classes of objects the function is able to accept.
2. What the function will do with each class; that is, what its method is for each. Although many important functions in R offer multiple methods (they are generic), not all do.
3. The data structure you supply to the function – that is, what the class of your object is. As we have seen, the way you select data determines their data structure or class.
4. If necessary, how to convert from the data structure you have to one you need.

In Chap. 7, "Selecting Variables," we learned that both of these commands select our variable q1 and pass on the data in the form of a data frame:

```
mydata[3]
mydata["q1"]
```

while these, with their additional comma, also select variable q1 but instead pass on the data in the form of a vector:

```
mydata[ ,3]
mydata[ ,"q1"]
```

Many functions would work just fine on either result. However, some procedures are fussier than others and require very specific data structures. If you are having a problem figuring out which form of data you have, there are functions that will tell you. For example,

```
class(mydata)
```

will tell you its class is “data frame.” Knowing that a data frame is a type of list, you will know that functions that require either of those structures will accept them. As with the `print` function, it may produce different output, but it will accept it. There are also functions that test the status of an object, and they all begin with “`is.`” For example,

```
is.data.frame( mydata[3] )
```

will display TRUE, but

```
is.vector( mydata[3] )
```

will display FALSE.

Some of the functions you can use to convert from one structure to another are listed in Table 10.5. Let us apply one to our data. First, recall how the `print` function prints our data frame in a vertical format:

```
> print(mydata)

  workshop gender q1 q2 q3 q4
1        R      f  1  1  5  1
2      SAS      f  2  1  4  1
3        R      f  2  2  4  3
4      SAS   <NA>  3  1 NA  3
5        R      m  4  5  2  4
6      SAS      m  5  4  5  5
7        R      m  5  3  4  4
8      SAS      m  4  5  5  9
```

Now let us print it in list form by adding the `as.list` function. This causes the data frame to become a list at the moment of printing, giving us the horizontal orientation that we have seen when printing a true list:

```
> print( as.list(mydata) )

$workshop
[1] R   SAS R   SAS R   SAS R   SAS
Levels: R SAS SPSS STATA

$gender
[1] f    f    f    <NA> m    m    m    m
Levels: f m

$q1
[1] 1 2 2 3 4 5 5 4
```

Table 10.5. Data conversion functions

Conversion to perform	Example

Index vector to logical vector	`myQindices <- c(3, 4, 5, 6)`
	`myQtf <- 1:6 %in% myQindices`

Vectors to columns of a data frame	`data.frame(x,y,z)`
Vectors to rows of a data frame	`data.frame( rbind(x,y,z) )`
Vectors to columns of a matrix	`cbind(x,y,z)`
Vectors to rows of a matrix	`rbind(x,y,z)`
Vectors combined into one long one	`c(x,y,z)`
Data frame to matrix (must be same type)	`as.matrix(mydataframe)`
Matrix to data frame	`as.data.frame(mymatrix)`
A vector to an r by c matrix	`matrix(myvector,nrow=r,ncol=c)` (note this is not as.matrix!)
Matrix to one very long vector	`as.vector(mymatrix)`
List to one long vector	`unlist(mylist)`
Lists or data frames into lists	`c(list1,list2)`
List of vectors, matrices to rows of matrix	`mymatrix <- ( do.call(rbind, myList) )`
List of vectors, matrices to cols of matrix	`mymatrix <- ( do.call(cbind, myList) )`

Logical vector to index vector	`myQtf <- c(FALSE, FALSE,`
	`TRUE, TRUE, TRUE, TRUE)`
	`myQindices <- which(myQtf)`

Table to data frame	`as.data.frame(mytable)`
Remove the class	`unclass(myobject)`

```
$q2
[1] 1 1 2 1 5 4 3 5

$q3
[1]  5  4  4 NA  2  5  4  5

$q4
[1] 1 1 3 3 4 5 4 9
```

10.19.1 Converting from Logical to Numeric Index and Back

We have looked at various ways to select variables and observations using both logical and numerical indices. Now let us look at how to convert from one to the other.

The which function will examine a logical vector and tell you which of its elements are true; that is, it will tell you the index values of those that are true. We can create a logical vector that chooses our q variables in many ways. Let us use the vector that selects the last four variables in our practice data frame. For various ways to create a vector like this, see Chap. 7, "Selecting Variables," or Chap. 8, "Selecting Observations." We will just enter it manually:

```
myQtf <- c(FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)
```

We can convert that to a numeric index vector that gives us the index numbers for each occurrence of the value TRUE using the which function:

```
myQindices <- which(myQtf)
```

Now myQindices contains 3, 4, 5, 6 and we can analyze just the q variables using

```
summary( mydata[myQindices] )
```

To go in the reverse direction, we would want to know which the variable indices – 1, 2, 3, 4, 5, 6 – were, in a logical sense, in our list of 3, 4, 5, 6:

```
myQindices <- c(3, 4, 5, 6)
```

Now we will use the %in% function to create the logical selection we need using:

```
myQtf <- 1:6 %in% myQindices
```

Now myQtf has the values FALSE, FALSE, TRUE, TRUE, TRUE, TRUE, and we can analyze just the Q variables with

```
summary( mydata[myQtf] )
```

Why are there two methods? The logical method is the most direct, since calling the `which` function is often an additional step. However, if you have 20,000 variables, as researchers in genetics often have, the logical vector will contain 20,000 values. The numeric index vector will have only as many values as there are variables in your subset. The `which` function also has a critical advantage. Since it looks only to see which values are TRUE, the NA missing values do not affect it. Using a logical vector, R will look at all values, TRUE, FALSE, even the missing values of NA. However, the selection `mydata[NA , ]` is undefined, causing problems. See Chap. 8, "Selecting Observations," for details.

10.20 Character String Manipulations

As we have seen, you can create character variables using the `c` function:

```
> gender <-c("m","f","m",NA,"m","f","m","f")

> gender

[1] "m" "f" "m" NA  "m" "f" "m" "f"
```

You must enclose the individual elements (values) of the vector in quotes, except for missing values, NA. If you actually needed the string "NA" to represent something like North America, you would then enclose it in quotes. It would work, but you would be asking for confusion in the long run!

R has two character variables built in, that contain the letters of the alphabet in lower- and upper-case:

```
> letters

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
[14] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

> LETTERS

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
[14] "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

We can use these letters to create variable names. Previously we used the `paste` function to create names like q1, q2, and q3. That is a good function to know because it is widely used. However, now that we are focusing on string variables and ways to manipulate them, we will use the `str_c` function. It comes from Wickham's `stringr` package [72], which contains a set of character string functions that are much more useful than those that are built into R. In particular, they can manipulate both character variables and factors.

Let us create some variables:

```
> library("stringr")

> myVars <- str_c( "Var", LETTERS[1:6] )

> myVars

[1] "VarA" "VarB" "VarC" "VarD" "VarE" "VarF"
```

In the above example, I loaded the `stringr` package and used `str_c` to concatenate (c=concatenate) the string "Var" to the first five letters of the uppercase alphabet. If we had mydata loaded into our workspace, we could switch to these names with:

```
names(mydata) <- myVars

summary( mydata[myVars] )
```

In our case, those names are worse than the original ones, but sometimes you get data sets from other sources that have similar names.

Let us now read a data set that contains the names of some famous statisticians. To make the matching SAS and SPSS programs easy, we will use a fixed-width format file. The following is the R code to read it:

```
> setwd("c:/myRfolder")

> giants <- read.fwf(
+    file  = "giants.txt",
+    width =  c(15, 11, 11),
+    col.names  = c("name",     "born",     "died"),
+    colClasses = c("character","character","POSIXct")
+ )

> giants
             name        born       died
1 R.A. Fisher     02/17/1890  1962-07-29
2 Carl Pearson    03/27/1857  1936-04-27
3 Gertrude Cox    01/13/1900  1978-10-17
4 John Tukey      06/16/1915  2000-07-26
5 William Gosset  06/13/1876  1937-10-16
```

That used the `read.fwf` function covered in Sect. 6.6, "Reading Fixed-Width Text Files, One Record per Case." The second and third columns we are reading are dates. We will discuss how to deal with those in Sec. 10.21.

We can check the length of each name string using the `str_length` function:

```
> str_length( giants$name )
```

```
[1] 15 15 15 15 15
```

I set the width argument to read the names from the first 15 columns of the file. I also left off the strip.white = TRUE argument, which would have saved space by trimming off the trailing blanks. Let us try two different ways to display the data for R.A. Fisher:

```
> giants[ giants$name == "R.A. Fisher", ]

[1] name born died

<0 rows> (or 0-length row.names)

> giants[ giants$name == "R.A. Fisher    ", ]

             name        born       died
1 R.A. Fisher     02/17/1890  1962-07-29
```

The first query found no results because Fisher's name is the full 15 characters including the trailing blanks. That not only wastes space, but it makes queries error prone as you try to count exactly how many blanks you need to add to each string.

In this case, the problem would have been best fixed by simply adding the strip.white = TRUE argument to the read.table function call. However, when importing data from other sources, such as a database, you may need an alternative approach to removing the trailing blanks. The str_trim function does just that:

```
> giants$name <- str_trim( giants$name )

> attach(giants)

> str_length(name)

[1] 11 12 12 10 14
```

The str_trim function would also remove any leading blanks, had their been any.

SAS users may get confused here since SAS's TRIM function removes only trailing blanks, while SAS's STRIP function removes both leading and trailing blanks.

SPSS users will find str_trim works similarly to the RTRIM function. However, while the RTRIM function will remove blanks when combining strings, it will not actually make the original variable shorter unless you are in UNICODE mode.

Another common problem when dealing with character strings is getting them into the proper case. Let us look at some examples. If you need to store strings in all uppercase or all lowercase, in SAS the functions are named UPCASE and LOWCASE. SPSS calls them UPCASE and LOWER. In R, the `toupper` function changes to uppercase:

```
> toupper(name)

[1] "R.A. FISHER"    "CARL PEARSON"   "GERTRUDE COX"
[4] "JOHN TUKEY"     "WILLIAM GOSSET"
```

While the `tolower` function does just the opposite:

```
> tolower(name)

[1] "r.a. fisher"    "carl pearson"   "gertrude cox"
[4] "john tukey"     "william gosset"
```

With proper names you typically want to have the first letter of each name capitalized, and all the following letters lowercase. The `ggplot2` package has a function that will do that:

```
> library("ggplot2")

> firstUpper( tolower(name) )

[1] "R.a. fisher"    "Carl pearson"   "Gertrude cox"
[4] "John tukey"     "William gosset"
```

You can see that in this example, the function would have been more useful if we had split the first and last names and applied it separately. The `firstUpper` function is not as helpful as the SAS PROPER function. That function would have seen the blank and capitalized the last name automatically. It would have even capitalized the "A" in "R.A." Fisher. Instead, `firstUpper` capitalizes only the first letter of a string.

Selecting pieces of strings is a very common data management task. SAS and SPSS both do this with a function named SUBSTR. The `stringr` package does it with the `str_sub` function:

```
> str_sub(name, 1, 5)

[1] "R.A. " "Carl " "Gertr" "John " "Willi"
```

Arguments 1 and 5 tell the function to begin selecting a piece of the string at the very first character, then continue on until the fifth character (i.e., 5 is the absolute location of the last character). This got some of the first names correct, but broke others off in the middle.

A better way to perform this task is to strip out all the characters wherever there is a space. All three packages solve this problem in different ways, so I will not mention the SAS and SPSS approaches. See the example programs at the end of this section to see what I used. In R, the `str_split_fixed` function makes quick work of this problem by splitting the names into a matrix:

```
> myNamesMatrix <- str_split_fixed(name, " ", 2)

> myNamesMatrix

     [,1]       [,2]
[1,] "R.A."     "Fisher"
[2,] "Carl"     "Pearson"
[3,] "Gertrude" "Cox"
[4,] "John"     "Tukey"
[5,] "William"  "Gosset"
```

The first argument to the `str_split_fixed` function is the name of the string to split and the second is the character to split at, in our case a blank. The final "2" is the number of items you would like the function to return. For this example, we need two strings returned for the first and last names. As you see, this has created a character matrix with the first name in column one and the second in column two.

You can easily extract the columns to vectors using:

```
> myFirst <- myNamesMatrix[ ,1]

> myFirst

[1] "R.A."  "Carl"  "Gertrude"  "John"  "William"

> myLast  <- myNamesMatrix[ ,2]

> myLast

[1] "Fisher"  "Pearson"  "Cox"  "Tukey"  "Gosset"
```

Finding and replacing parts of strings is often helpful when cleaning up a data set. If we wish to replace the "R.A." in Fisher's name with "Ronald A.", SAS does such replacements with its TRANWRD function, while SPSS uses REPLACE. The `stringr` package calls it `str_replace_all`:

```
> myFirst <- str_replace_all(myFirst, "R.A.", "Ronald A.")
```

A very similar function, `str_replace`, would also work in this example. It replaces only the first occurrence that it finds in each string.

Finally, let us recombine the names with the last name first. This is the type of concatenation that SAS performs using its `||` operator, while SPSS uses its CONCAT function. In `stringr` it is called the `str_c` function, with c standing for *c*oncatenate:

```
> myLastFirst <- str_c( myLast, ", ", myFirst)

> myLastFirst

[1] "Fisher, Ronald A." "Pearson, Carl"
[3] "Cox, Gertrude"     "Tukey, John"
[5] "Gosset, William"
```

You can select observations based on string comparisons or searches in R using logical comparisons, just as you would in SAS or SPSS. To list observations whose whole strings match, we can use the methods discussed in Chap. 8, "Selecting Observations,":

```
> myObs <- myLast == "Tukey"
> myObs
[1] FALSE FALSE FALSE  TRUE FALSE

> myObs <- which(myLast == "Tukey")
> myObs
[1] 4

> giants[ myObs, ]

        name       born       died
4 John Tukey 06/16/1915  2000-07-26
```

The first example creates MYOBS as a logical vector. The second adds the `which` function to find which of the logical conditions is true. Using either approach, the final selection is the same.

However, what if we want to select on just part of a name? Here we look for all the names that contain the lower-case string "key" in the people's last names:

```
> myObs <- str_detect(myLast, "key")

> myObs

[1] FALSE FALSE FALSE TRUE FALSE
```

We can use that logical result in the same way as before.

We can look for a list of people using the %in% function. First we will create a lookup table:

```
> myTable <- c("Box","Bayes","Fisher","Tukey")
```

And now we will test to see where the last names appear in the lookup table:

```
> myTable <- c("Box","Bayes","Fisher","Tukey")

> myObs <- myLast %in% myTable

> myObs

[1]  TRUE FALSE FALSE  TRUE FALSE

> name[ myObs ]

[1] "R.A. Fisher" "John Tukey"
```

As easy as that example was, it was limited to precise matches. We can use the str_detect function to search using the powerful *regular expressions.* In the following example, we are able to find Fisher and Tukey by searching for the strings "Fish" and "key", respectively:

```
> myObs <- str_detect( myLast, "Box|Bayes|Fish|key" )

> myObs

[1]  TRUE FALSE FALSE  TRUE FALSE

> name[ myObs ]

[1] "R.A. Fisher" "John Tukey"
```

The first argument to the str_detect function is the string object to search and the second is any regular expression enclosed in quotes. You can search the Internet to find many useful descriptions of regular expressions and how to use them.

The str_detect function is very useful. For example, if you wanted to find the people whose last names began with the letters "A" through "M", the regular expression that is "^[A-M]". Using that, we can find them easily with

```
> myAthruM <- str_detect(myLastFirst, "^[A-M]")

> myAthruM
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

```
> name[ myAthruM ]
[1] "R.A. Fisher"   "Gertrude Cox"   "William Gosset"
```

You can find the rest of the alphabet with the regular expression "^[N-Z]", or you can save the effort by simply asking for those not in the first selection:

```
> name[!myAthruM ]

[1] "Carl Pearson"   "John Tukey"
```

10.20.1 Example Programs for Character String Manipulation

SAS Program for Character String Manipulation

```
* Filename: CharacterStrings.sas

LIBNAME MYlIB  C:\myRfolder ;

DATA myLib.giants;
INFILE  \myRfolder\giants.txt
  MISSOVER DSD LRECL=32767;
INPUT name $char14. @16 born mmddyy10. @27 died yymmdd10.;
FORMAT born mmddyy10. died yymmdd10.;
myVarLength=length(name);
born=strip(born); *Not needed;

PROC PRINT;
  RUN;

DATA  myLib.giants;
  SET myLib.giants;
  myLower= lowcase(name);
  myUpper= upcase(name);
  myProper=propcase(name);
PROC PRINT;   RUN;

DATA  myLib.giants;
  SET myLib.giants;
  myFirst5=substr(name, 1, 5);
  * split names using substr;
  myBlank=find(name, " ");
  myFirst=strip( substr(name, 1, myBlank) );
  myLast =strip( substr(name, myBlank)    );
  PUT "Using substr... " myFirst= myLast=;
  * splip names using scan;
  myFirst=scan(name,1," ");
  myLast =scan(name,2," ");
  myFirst=tranwrd(myFirst,"R.A.","Ronald A.");
  LENGTH myLastFirst $ 17;
  myLastFirst= strip(myLast) || ", " || strip(myFirst);
  *or: CALL CATX(", ", myLastFirst, myLast, myFirst);
PROC PRINT; VAR name myFirst myLast myLastFirst;
  RUN;
DATA  tukey;
  SET myLib.giants;
  WHERE myLast="Tukey";
PROC PRINT;
  VAR name;
  RUN;
DATA  tukey;
  SET myLib.giants;
  WHERE FIND(myLast, "key");
PROC PRINT;
  VAR name;
  RUN;
DATA  mySubset;
  SET myLib.giants;
  WHERE myLast IN ("Box","Bayes","Fisher","Tukey");
  RUN;
PROC PRINT;
  VAR name;
  RUN;
DATA  FishOrKey;
  SET myLib.giants;
  IF FIND(myLast, "Box")   |
     FIND(myLast, "Bayes") |
     FIND(myLast, "Fish")  |
     FIND(myLast, "key") ;
  RUN;
PROC PRINT;
  VAR name;
  RUN;
DATA AthruM;
  SET myLib.giants;
  firstLetter=substr(myLast, 1, 1);
  IF "A" <= firstLetter <= "M";
  RUN;
PROC PRINT;
  VAR name;
  RUN;
```

SPSS Program for Character String Manipulation

```
* Filename: CharacterStrings.sps

CD  C:\myRfolder .

DATA LIST FILE= giants.txt  RECORDS=1
 /1 name 1-14 (A) born 16-26 (ADATE) died 27-37 (SDATE).

STRING myFirst5 (A5)/ myLower myUpper myLastFirst (A17)
  myFirst myLast (A9).
COMPUTE myLength1=LENGTH(name).
COMPUTE name=RTRIM(name).
COMPUTE myLength2=LENGTH(name).
COMPUTE myLower=LOWER(name).
COMPUTE myUpper=UPCASE(name).
LIST name myLower myUpper myLength1 myLength2.

COMPUTE myFirst5=SUBSTR(name, 1, 5).
COMPUTE myBlank= INDEX(name, " ").
COMPUTE myFirst=SUBSTR(name, 1, myBlank-1).
COMPUTE myFirst=REPLACE(myFirst, "R.A.", "Ronald A.").
COMPUTE myLast=SUBSTR(name, myBlank+1).
COMPUTE myLastFirst=CONCAT( RTRIM(myLast),
   ", ", RTRIM(myFirst) ).
LIST name myFirst myLast myLastFirst.

TEMPORARY.
SELECT IF (myLast EQ "Tukey").
LIST name.

TEMPORARY.
SELECT IF (CHAR.RINDEX(myLast,"Fish") GE 1
  OR CHAR.RINDEX(myLast,"key") GE 1).
LIST name.

TEMPORARY.
SELECT IF (
  myLast EQ "Box" OR
  myLast EQ "Bayes" OR
  myLast EQ "Fisher" OR
  myLast EQ "Tukey").
LIST name.
```

```
TEMPORARY.
SELECT IF (
  name EQ "Box" OR
  name EQ "Bayes" OR
  CHAR.RINDEX(myLast,"Fish") GE 1 OR
  CHAR.RINDEX(myLast,"key") GE 1).
LIST name.

TEMPORARY.
SELECT IF(SUBSTR(myLast, 1, 1) LE "M").
LIST name.
```

R Program for Character String Manipulation

```
# Filename: CharacterStrings.R

gender <-c("m", "f", "m", NA, "m", "f", "m", "f")
gender

options(width = 58)
letters
LETTERS

library("stringr")
myVars <- str_c("Var", LETTERS[1:6])
myVars

setwd("c:/myRfolder")
giants <- read.fwf(
   file  = "giants.txt",
   width = c(15, 11, 11),
   col.names  = c("name",      "born",      "died"),
   colClasses = c("character", "character", "POSIXct")
)
giants

str_length( giants$name )

giants[ giants$name == "R.A. Fisher", ]
giants[ giants$name == "R.A. Fisher    ", ]

giants$name <- str_trim(giants$name)
attach(giants)
str_length(name)
```

```
toupper(name)
tolower(name)
library("ggplot2")
firstUpper( tolower(name) )

str_sub(name, 1, 5)

myNamesMatrix <- str_split_fixed(name, " ", 2)
myNamesMatrix

myFirst <- myNamesMatrix[ ,1]
myFirst
myLast  <- myNamesMatrix[ ,2]
myLast

myFirst <- str_replace_all(myFirst, "R.A.", "Ronald A.")

myLastFirst <- str_c( myLast, ", ", myFirst)
myLastFirst

myObs <- myLast == "Tukey"
myObs
myObs <- which(myLast == "Tukey")
myObs
giants[ myObs, ]

myObs <- str_detect(myLast, "key")
myObs

myTable <- c("Box", "Bayes", "Fisher", "Tukey")
myObs <- myLast %in% myTable
myObs
name[ myObs ]

myObs <- str_detect( myLast, "Box|Bayes|Fish|key" )
myObs
name[ myObs ]

myAthruM <- str_detect(myLastFirst, "^[A-M]")
myAthruM
name[ myAthruM ]
name[!myAthruM ]
```

10.21 Dates and Times

Date and time measurements are a challenge to work with because the units of measure are a mess. Years have 365 days, except during leap years when they have 366. Months vary from 28 to 31 days in length. A day consists of 24 hours, each with 60 minutes and 3,600 seconds. To add to the excitement, we also have time zones, daylight savings time, and leap seconds to deal with.

Let us begin with a brief review of how SAS handles dates and times. It has three kinds of date–time variables: dates, datetimes, and times. SAS considers the origin, or zero point in time, to be the beginning of 1970. It stores dates as the number of days since January 1, 1970, and datetimes as the number of seconds since then. Negative numbers indicate dates or datetimes that precede 1970. Time variables are stored as the number of seconds into the current day.

SPSS uses only two types of date–time variables: dates-times and times. Both date–times and times are stored as the number of seconds from their respective origin points. The origin point for date–times is the beginning of the Gregorian Calendar, October 14, 1582. The origin for time variables is the start of the current day.

In both SAS and SPSS, the data may *appear* as dates, times, or date–times when you read them from a file, but once they have been read using the appropriate format, there is no discernible difference between the date–time variables and any others; they become simply numeric variables. SPSS hides this fact by automatically assigning a print format, but if you strip that away, you see just numbers of seconds with no clue as to what the numbers originally represented.

R has a variety of date and time objects. We will focus on the newest approach using Grolemund and Wickham's amusingly named `lubridate` package [24]. It is documented in their article, *Dates and Times Made Easy with lubridate* [25].

Let us dive in and read a file that contains the birth/death dates of some intellectual giants from the field of statistics. Here is what the file looks like:

```
R.A. Fisher    02/17/1890 1962-07-29
Carl Pearson   03/27/1857 1936-04-27
Gertrude Cox   01/13/1900 1978-10-17
John Tukey     06/16/1915 2000-07-26
William Gosset 06/13/1876 1937-10-16
```

The dates are in two different formats to demonstrate the fact that R is currently very limited in the format of dates it can read directly from a file. To make the matching SAS and SPSS programs easy, we will use a fixed-width format file. The following is the R code to read it:

```
> setwd("c:/myRfolder")

> giants <- read.fwf(
+    file        = "giants.txt",
+    width       = c(15,11,11),
+    col.names   = c("name","born","died"),
     colClasses  = c("character","character","POSIXct"),
+    row.names   = "name",
+    strip.white = TRUE,
+ )
```

We are using the `read.fwf` function covered in Sect. 6.6, “Reading Fixed-Width Text Files, One Record per Case.” The data are displayed below. The second and third columns we are reading are dates. However, in the `colClasses` argument I specified that the first two are “character” variables and the third is “POSIXct.” SAS or SPSS users would expect two consecutive date specifications in a row. SAS would have used MMDDYY10 and YYMMDD10, while SPSS would have used ADATE followed by SDATE.

However, R can only read dates-times in the YYYY-MM-DD format, which is used in the last column in our data set. For any other format, you must first read the dates (or date–times) as character variables and then convert them. Here is the data as we read them:

```
> giants
                      born       died
R.A. Fisher    02/17/1890 1962-07-29
Carl Pearson   03/27/1857 1936-04-27
Gertrude Cox   01/13/1900 1978-10-17
John Tukey     06/16/1915 2000-07-26
William Gosset 06/13/1876 1937-10-16
```

Although the dates look like dates, the `class` function shows that born is still just a character variable:

```
> class( giants$born )

[1] "character"
```

The `lubridate` package contains a variety of conversion tools. We will use the `mdy` function to convert our character variables to dates:

```
> library("lubridate")

> giants$born <- mdy( giants$born )

Using date format %m/%d/%Y.
```

The message that follows each use of the `mdy` function may seem redundant, but it is not. The function is capable of reading a wide range of dates including

```
02/17/1890
02-17-1890
02:17:1890
02171890
```

Therefore, the message is reporting which of its various formats it found. This is very similar to SAS's "any date" format, ANYDTDTE, along with its DATESTYLE option set to MDY. In other words, it is going to try to determine the date, but only within the constraints that month is first followed by day, then year. The abbreviations `%m` for month, `%d` for day, and `%Y` for the four-digit year are the same specifications that SAS uses in its picture formats for dates. For a list of these date–time format conversion specifications see Table 10.6.

Let us see how the data looks now:

```
> giants
                      born       died
R.A. Fisher     1890-02-17 1962-07-29
Carl Pearson    1857-03-27 1936-04-27
Gertrude Cox    1900-01-13 1978-10-17
John Tukey      1915-06-16 2000-07-26
William Gosset  1876-06-13 1937-10-16
```

Notice that R now presents both of the dates from largest to smallest units of year, then month, then day. Although we do not see them, there are also times associated with these dates. Since we did not actually read any times, they defaulted to zero hours, zero minutes, and zero seconds, or the start of each date. Had it been otherwise, R would have printed the times automatically.

What classes of variables are these? Let us check:

```
> class(giants$born)

[1] "POSIXt"  "POSIXct"

> class(giants$died)

[1] "POSIXt"  "POSIXct"
```

The dates are stored with two classes, POSIXt (t for Time) and POSIXct (ct for Calendar Time). POSIX is a set of computer standards that include rules for dates and times.

The POSIXct class stores the number of seconds since the start of 1970. Another class not shown is POSIXlt (lt=Local Time). That class stores date–times in a list of date elements (e.g., year, month, day). I rarely use the POSIXlt class.

Finally, POSIXt is a class that applies to both POSIXct and POSIXlt and indicates that either class can contain date–times that we can use for math, like calculating differences.

The `print` function displays our date variables using the method that its designer chose for that class of object. We can see what they really contain by temporarily stripping away their class. Let us do that for born:

```
> attach(giants)

> unclass( born )
[1] -2520460800 -3558556800 -2207952000 -1721347200
[5] -2952201600

attr(,"tzone")
[1] "UTC"
```

Now we see that just as in SAS and SPSS, the date–times are stored as a number of seconds. In this case they are the number of seconds between the dates and the start of 1970. Since the seconds are negative, they measure how many seconds *before* 1970 each person was born.

We also see that this POSIXct variable has an attribute of the time zone, set to UTC, which stands for Coordinated Universal Time. If you do precise date–time calculations with data from multiple time zones, you will want to learn more about time zones.

If we want to make this variable a simple numeric vector, we can using the `as.numeric` function:

```
> as.numeric( born )

[1] -2520460800 -3558556800 -2207952000 -1721347200
[5] -2952201600
```

You could go in the reverse direction, converting a vector of seconds into a date using the `as.POSIXct` function:

```
> as.POSIXct(
+   c(-2520460800,-3558556800,-2207952000,
+     -1721347200, -2952201600),
+   origin="1970-01-01", tz="UTC" )

[1] "1890-02-16 19:00:00 EST" "1857-03-26 19:00:00 EST"
[3] "1900-01-12 19:00:00 EST" "1915-06-15 19:00:00 EST"
[5] "1876-06-12 19:00:00 EST"
```

Notice that one of the arguments is the origin date. This comes in handy when reading dates stored in seconds that were written to text files from other software. If you were reading date–times written from SAS, you could fill in

1960-01-01 here. If you were reading date–times from Excel, you could use its origin 1900-01-01. For date–times originating in SPSS you would fill in 1582-10-14. This step applies only to text files since tools for importing data directly from other file formats would do the proper conversion for you.

10.21.1 Calculating Durations

We can calculate durations of time using the `difftime` function. It is very similar to SAS's DATDIF function and SPSS's DATEDIFF function. Let us use `difftime` to calculate the statisticians' ages:

```
> age <- difftime(died, born, units="secs")

> age

Time differences in secs
[1] 2286057600 2495664000 2485382400 2685916800 1935705600

attr(,"tzone")
[1] "UTC"
```

We now have age in seconds. It is a precise answer, but not one that we can relate to for durations that last years.

Let us do it again using the default setting, `units=auto`, which will automatically choose an appropriate unit. In this case, it chooses days.

```
> age <- difftime(died, born)

> age

Time differences in days
[1] 26459 28885 28766 31087 22404

attr(,"tzone")
[1] "UTC"
```

The `difftime` function can also measure durations in minutes, hours or weeks. It does not use months or years because those units of time vary too much in length.

The variable age appears to be a vector, but since it prints out the message regarding time differences and the time zone (tzone) attribute, we know it is not just a vector. Let us see what type of object it is:

```
> mode( age )
[1] "numeric"

> class( age )
[1] "difftime"
```

We see that it is a numeric object with a class of *difftime*. Using the jargon of the lubridate package, we have created a "duration," a precise measure of time. Durations can be added to or subtracted from other date–times or other durations.

Although the difftime function does not calculate in years, we have two options for converting age to years. The first approach is best when you are more interested in seeing ages in years and days. You simply call the as.period function:

```
> as.period(age)

[1] 10 years and 909 days   11 years and 780 days...
```

Periods are objects that the lubridate package uses to store times that are less precise but that match calendar time well.

The second approach to converting durations from days to years is to divide by the number of days in a typical year. When accounting for leap years, an average year contains 365.2425 days. We can use that to convert age into years:

```
> age/365.2425

Time differences in days
[1] 72.44228 79.08444 78.75863 85.11332 61.34007

attr(,"tzone")
[1] "UTC"
```

We *do* have age in years now, but difftime objects have a *units* attribute that is still set to days. We cannot simply change this with:

```
units(age) <- "years"  # Bad!
```

because difftime objects do not allow units greater than weeks. We can fix that by converting it to a numeric vector, and let us round it off to two decimal places while we are at it:

```
> giants$age <- round(
+   as.numeric( age/365.2425 ), 2 )

> giants
                     born       died   age
R.A. Fisher    1890-02-17 1962-07-29 72.44
Carl Pearson   1857-03-27 1936-04-27 79.08
Gertrude Cox   1900-01-13 1978-10-17 78.76
John Tukey     1915-06-16 2000-07-26 85.11
William Gosset 1876-06-13 1937-10-16 61.34
```

If you want to compare date–times to the current date and time, it is convenient to use a function like SAS's TODAY function or SPSS's system variable $TIME. In the lubridate package the equivalent function is aptly named now. Base R also has functions named Sys.time and date that are very similar.

Let us see how long ago these people died:

```
> difftime( now(), died ) / 365.2425

Time differences in days
[1] 48.10735 74.36114 31.88799 10.11341 72.89088

attr(,"tzone")
[1] "UTC"
```

Again, it is labeled in days when we have converted to years, but we see that the last statistician, John Tukey, died just over 10 years before I wrote this.

We have seen that the difftime function calculates time durations. In SAS or SPSS, this would have been easy to accomplish with subtraction. R can use subtraction, too, but with an odd twist. Rather than creating *durations*, subtracting dates creates *intervals* instead. Let us see how the two differ.

Although I do not recommend using subtraction to calculate durations, seeing what will happen will help you to understand the full impact that a package can have on how R operates.

First I will load the lubridate package again. It is already loaded, so I am doing this only to emphasize that it changes the behavior of subtracting two date–time instants. For a SAS or SPSS user, running something like a macro would never change the function of fundamental operators like the minus sign. However, R *is that flexible*, and the lubridate package does just that!

```
> library("lubridate")

> age <- died - born  # age in days

> age

[1] 26459 days beginning at 1890-02-17
[2] 28885 days beginning at 1857-03-27
[3] 28766 days beginning at 1900-01-13
[4] 31087 days beginning at 1915-06-16
[5] 22404 days beginning at 1876-06-13

> age / 365.2425

[1] 72.44228 days beginning at 1890-02-17
```

```
[2] 79.08444 days beginning at 1857-03-27
[3] 78.75863 days beginning at 1900-01-13
[4] 85.11332 days beginning at 1915-06-16
[5] 61.34007 days beginning at 1876-06-13
```

We get the same answers, although formatted differently. Now it includes the beginning date! Let us look closer at this new age object:

```
> mode(age)
[1] "list"

> class(age)
[1] "interval"   "data.frame"

> names(age)
[1] "start" "end"
```

We see that age is now a list with classes of both interval and data.frame. It also contains two variables: start and end. But it printed as vector with some labels added. What is going on? The designers of the interval class included a method for the `print` function that told it to print the difference between the start and end variables, but not the variables themselves! By simply typing the name "age" we are invoking the `print` function. Recall that we can override the `print` function's methods by "unclassing" an object. That allows us to really see what is inside it. However, we do not want to unclass age itself, but rather the two variables that it contains. We can use the `sapply` function to do that:

```
> sapply(age, unclass)

           start         end
[1,] -2520460800  -234403200
[2,] -3558556800 -1062892800
[3,] -2207952000   277430400
[4,] -1721347200   964569600
[5,] -2952201600 -1016496000
```

Now we see that when we create an interval object by subtracting two date–time variables, we create a data frame that contains the start and end of each person's age interval. The actual difference isn't calculated until the `print` function does it. We have seen the `print` function prints subsets of objects before, but we have never seen it do something like subtraction!

This interval object is a very flexible structure, but further use of it is beyond our scope. For date–time calculations, the `difftime` approach is often the better choice.

10.21.2 Adding Durations to Date–Time Variables

We have seen how to calculate a duration from two date–time variables. But what if we already know a duration, and want to add it to or subtract it from date–time variables?

Let us take the case where we know when the people were born and their ages, but not when they died.

We can use the as.duration function to create age from a numeric vector:

```
> age <- as.duration(
+          c(2286057600,2495664000,2485382400,
+          2685916800,1935705600)
+ )

> class(age)
[1] "difftime"
```

We see from the class function that our new age variable is indeed a difftime object. We can add it to the date they were born using addition:

```
> born+age

[1] "1962-07-29 UTC" "1936-04-27 UTC" "1978-10-17 UTC"
[4] "2000-07-26 UTC" "1937-10-16 UTC"
```

How did we do? Let us compare that to the variable died:

```
> died

[1] "1962-07-29 UTC" "1936-04-27 UTC" "1978-10-17 UTC"
[4] "2000-07-26 UTC" "1937-10-16 UTC"
```

That is correct!

10.21.3 Accessing Date–Time Elements

It is easy to access the various elements of date–time variables. The functions have similar names to their SAS and SPSS counterparts and are similarly easy to use:

```
> year(born)

[1] 1890 1857 1900 1915 1876
```

You can extract months in numeric or character form, with abbreviated names or not:

```
> month(born)

[1] 2 3 1 6 6

> month(born, label = TRUE)

[1] Feb Mar Jan Jun Jun

12 Levels: Jan < Feb < Mar < Apr < May < Jun < ... < Dec
```

Days can be of the month or week, and days of the week can be accessed as numbers or names:

```
> day(born)  # day of month

[1] 17 27 13 16 13

> wday(born)  # day of week

[1] 2 6 7 4 3

> wday(born, label = TRUE, abbr = FALSE)

[1] Monday    Friday    Saturday  Wednesday Tuesday

7 Levels: Sunday < Monday < Tuesday < ... < Saturday
```

Finally, you can see which day of the year the birthdays appeared on:

```
> yday(born)

[1]  48  86  13 167 165
```

One of the many nice things that the `lubridate` package does for us is display weekdays with values from 1 to 7 and months from 1 to 12. The base R functions begin at zero for both! After years of thinking of months starting at 1, thank goodness we do not have to make that mental adjustment.

10.21.4 Creating Date–Time Variables from Elements

If your data set contains separate variables for the elements of a date (e.g., years, months, days), it is easy to combine them into a date–time variable.

To demonstrate this, let us split our died variable into its elements:

```
myYear  <- year(died)
myMonth <- month(died)
myDay   <- day(died)
```

Now we can recreate the variable by combining the elements using the paste function. Since our original data had dates in mdy format, let us choose a hyphenated ymd format just to see that R can handle it:

```
> myDateString <- paste( myYear, myMonth, myDay, sep = "-" )

> myDateString

[1] "1962-7-29"  "1936-4-27"  "1978-10-17" "2000-7-26"
[5] "1937-10-16"
```

Finally, we can use the ymd function to convert the string to a date–time variable:

```
> died2 <- ymd(myDateString)

Using date format %Y-%m-%d.

> died2

[1] "1962-07-29 UTC" "1936-04-27 UTC" "1978-10-17 UTC"
[4] "2000-07-26 UTC" "1937-10-16 UTC"
```

The powerful string handling functions discussed in Sect. 10.20 can help you read date–time variables in almost any form.

10.21.5 Logical Comparisons with Date–Time Variables

As with SAS and SPSS, logical comparisons between date–times are done just as they are between numbers. For example, if we wanted to list the people who were born after the start of 1900, we could use the following selection:

```
> giants[ born > mdy("1/1/1900") , ]

                   born       died
Gertrude Cox 1900-01-13 1978-10-17
John Tukey   1915-06-16 2000-07-26
```

If any dates had been missing, we could have avoided listing them by nesting the logic within the which function.

10.21.6 Formatting Date–Time Output

We have seen that R uses the year–month–day format of printing date–times. You can use the format function to control how R prints values, including date–times. The format function and SAS picture formats both use the same ISO conversion specifications to control the style of the output. Let us look at an example:

```
> myDateText <- format(born, "%B %d, %Y is day %j of %Y")

> myDateText

[1] "February 17, 1890 is day 048 of 1890"
[2] "March 27, 1857 is day 086 of 1857"
[3] "January 13, 1900 is day 013 of 1900"
[4] "June 16, 1915 is day 167 of 1915"
[5] "June 13, 1876 is day 165 of 1876"
```

I am telling `format` to print the born variable using `%B` to represent the unabbreviated month. The code `%Y` represents the four-digit year, and so on. See Table 10.6 for all the specifications.

We can get a bit fancier by using the `cat` function (short for con*cat*enate). It concatenates strings and values and then displays them, just like the SAS PUT or SPSS PRINT commands. The following is an example:

```
> myDateText <- format(born,
+   "was born on the %jth day of %Y")

> for (i in 1:5) cat(rownames(giants)[i],
+   myDateText[i],"\n")

R.A. Fisher was born on the 048th day of 1890
Carl Pearson was born on the 086th day of 1857
Gertrude Cox was born on the 013th day of 1900
John Tukey was born on the 167th day of 1915
William Gosset was born on the 165th day of 1876
```

The `for` function repeats the `cat` function five times, once for each row in our data frame. The `\n` at the end of the `cat` function call tells R to go to a new line.

10.21.7 Two-Digit Years

It is a good idea to record years using four digits, like 2011. Even if you do, you will probably still need to read data sets with years recorded using just two digits. If you read a year of “11,” does it represent 1911 or 2011? At some point you have to decide whether to add 1900 or 2000 to any two-digit year you read.

In SAS, the YEARCUTOFF option controls this decision, and it has a default value of 1920. SPSS calls this the *Century Range for 2-Digit Years* and controls it through a dialog box, *Edit> Options> Data*.

R follows a POSIX standard that currently assumes years 69 to 99 are in the 1900s and years 00 to 68 are in the 2000s. That is like setting YEARCUTOFF=1969 in SAS. Unlike SAS and SPSS, R does not have an option to change this. If this does not match your data, then you will have to add or subtract the proper number of years to correct the problem.

Let us look at an example. We will read my birthday in 1969:

```
> my1969 <- mdy("08/31/69")
Multiple format matches with 1 successes: %m/%d/%y, %m/%d/%Y.

> my1969

[1] "1969-08-31 UTC"
```

That looks fine. Let us do it again, this time for 1968:

```
> my1968 <- mdy("08/31/68")
Multiple format matches with 1 successes: %m/%d/%y, %m/%d/%Y.

> my1968

[1] "2068-08-31 UTC"
```

Oops! We crossed the century cutoff and ended up in 2068 instead. I hope I make it that far! Since we know that time-dates are stored in seconds, we can subtract 100 years' worth of seconds:

```
> my1968 <- my1968 - as.duration(100 * 365.2425 * 24 * 60 * 60)

> my1968

[1] "1968-08-31 18:00:00 UTC"
```

Problem fixed! Keep in mind that over the years the POSIX standard will change, and then so will R.

10.21.8 Date–Time Conclusion

While we have covered the basics of date–time calculations, R offers an extensive array of additional features should you ever need them.

10.21.9 Example Programs for Dates and Times

SAS Program for Dates and Times

```
* Filename: DateTime.sas

LIBNAME MyLib  C:\myRfolder ;

DATA myLib.giants;
```

Table 10.6. Date-time format conversion specifications.

`%a`	Weekday name, abbreviated
`%A`	Weekday name, full
`%b`	Month name, abbreviated
`%B`	Month name, full
`%c`	Date and time
`%d`	Day of month (1–31)
`%H`	Hour (0–23)
`%I`	Hour (1–12)
`%j`	Day of year (1–365)
`%m`	Month (1–12)
`%M`	Minute (0–59)
`%p`	AM or PM
`%S`	Second (0–61, <=2 leap seconds)
`%U`	Week of year (0–53, Sunday as first day of week
`%w`	Day of week (0–6, 0=Sunday)
`%W`	Week of year (0–53), Monday as first day of week
`%x`	Date, `%y/%m/%d` on input, locale-specific on output
`%X`	Time, `%H/%M/%S` on input, locale-specific on output
`%y`	Year, two-digit (00–99)
`%Y`	Year, four-digit
`%z`	Offset in hours/minutes from UTC
`%Z`	Time zone (output only)
`%%`	Percent character (%)

```
INFILE \myRfolder\giants.txt
  MISSOVER DSD LRECL=32767;
INPUT name $char14. @16 born mmddyy10. @27 died mmddyy10.;

PROC PRINT;
  RUN;

PROC PRINT;
  FORMAT died born mmddyy10.;
  RUN;

* Caculating Durations.;
DATA myLib.giants;
SET  myLib.giants;
  age  = (died-born)/365.2425;
  longAgo = ( today()-died )/365.2425;
  RUN;

PROC PRINT;
FORMAT died born mmddyy10. age longAgo 5.2 ;
RUN;

* Adding Durations to Date-Times.
DATA myLib.giants;
SET myLib.giants;
  died=born+age;
  RUN;
PROC PRINT;
FORMAT died born mmddyy10. age 5.2;
RUN;

* Accessing Date-Time Elements;
DATA myLib.giants;
  SET myLib.giants;
  myYear=YEAR(born);
  myMonth=MONTH(born);
  myDay  =DAY(born);
PROC PRINT;
  FORMAT died born mmddyy10. age 5.2;
RUN;

* Creating Date-Time Variables from Elements;
DATA myLib.giants;
  set myLib.giants;
  born=MDY(myMonth, myDay, myYear);
  PROC PRINT;
  FORMAT died born mmddyy10. age 5.2;
RUN;

* Logical Comparisons with Date-Times;
DATA Born1900s;
  set myLib.giants;
  if born > "01jan1900"d ;
PROC PRINT;
  FORMAT died born mmddyy10. age 5.2;
  RUN;

* Formatting Date-Time Output;
PROC FORMAT;
  PICTURE myFormatI
  LOW - HIGH = '%B %d, %Y is day %j of %Y'
    (DATATYPE=DATE);
  RUN;
PROC PRINT DATA=myLib.giants;
  VAR born;
  FORMAT born myFormatI40.;
  RUN;

PROC FORMAT;
  PICTURE myFormatII
  LOW - HIGH = ' was born on the %jth day of %Y'
    (DATATYPE=DATE);
  RUN;
DATA _NULL_;
  SET myLib.giants;
  PUT name $char14. born myFormatII34.;
  run;
```

SPSS Program for Dates and Times

```
* Filename: DateTime.sps

CD 'C:\myRfolder'.

DATA LIST FILE='giants.txt' RECORDS=1
 /1 name 1-14 (A) born 16-26 (ADATE) died 27-37 (SDATE).

* Calculating Durations.
COMPUTE age=died-born.
LIST.
```

```
COMPUTE age=(died-born) / (365.2425*24*60*60).
LIST.
COMPUTE longAgo=($TIME-died) /(365.2425*24*60*60) .
LIST.

* Adding Durations to Date-Times.
COMPUTE died=born+age.
LIST.

* Accessing Date-Time Elements.
COMPUTE myYear=XDATE.YEAR(born).
COMPUTE myMonth=XDATE.MONTH(born).
COMPUTE myDay=XDATE.MDAY(born).
LIST name born myYear myMonth myDay.

* Creating Date-Time Variables from Elements.
COMPUTE born=DATE.MDY(myMonth, myDay, myYear).
LIST name born.

* Logical Comparisons with Date-Times.
TEMPORARY.
SELECT IF born GE date.mdy(1,1,1900).
LIST name born.

* Formatting Date-Time Output.
PRINT /born (adate)  is the  myDay (F3.0)
  th day of  myYear (F4.0).
EXECUTE.

PRINT /name  was born on the  myDay (F3.0)
  th day of  myYear (F4.0).
EXECUTE.
```

R Program for Dates and Times

```
# Filename: DateTime.R

setwd("c:/myRfolder")
giants <- read.fwf(
   file        = "giants.txt",
   width       = c(15,11,11),
   col.names   = c("name",      "born",      "died"),
   colClasses  = c("character", "character", "POSIXct"),
   row.names   = "name",
   strip.white = TRUE,
)
giants
class(giants$born)  # A character vector.

library("lubridate")
giants$born <- mdy(giants$born)
giants

class(giants$born)
class(giants$died)

giants  # They display in yyyy-mm-dd by default.
attach(giants)

unclass( born )
as.numeric( born )
as.POSIXct(
  c(-2520460800,-3558556800,-2207952000,
    -1721347200, -2952201600),
  origin="1960-01-01", tz="UTC" )

#---Calculating Durations---

age <- difftime(died, born, units="secs")
age
age <- difftime(died, born)
age # now we have age in days

mode( age )
class( age )  # it s a difftime object

as.period(age)

age/365.2425  # age in years
giants$age <- round(
  as.numeric( age/365.2425 ), 2 )
giants

now()  # Current date-time.
difftime( now(), died ) / 365.2425

# Again, using subtraction.
age <- died - born  # age in days
age
age / 365.2425   # Age in years, mislabeled.
```

```
mode(age)
class( age )     # it s an interval object.
names( age )
sapply( age, unclass )
# Not a helpful age to store in our data frame!


#---Adding Durations to Date-Times---

age <- as.duration(
         c(2286057600,2495664000,2485382400,
         2685916800,1935705600)
)
class(age)
born+age
died


#---Accessing Date-Time Elements---

year(born)
month(born)
month(born, label = TRUE)
day(born)   # day of month
wday(born)  # day of week
wday(born, label = TRUE, abbr = FALSE)
yday(born)

#---Creating Date-Times from Elements---

myYear  <- year(died)
myMonth <- month(died)
myDay   <- day(died)

myDateString <- paste(myYear, myMonth, myDay, sep="/")
myDateString
died2 <- ymd(myDateString)
died2

#---Logical Comparisons with Date-Times---

giants[ born > mdy("1/1/1900") , ]

#---SAS Picture Format Example---
```

```
myDateText <- format(born, "%B %d, %Y is day %j of %Y")
myDateText

myDateText <- format(born,
  "was born on the %jth day of %Y")
for (i in 1:5) cat(rownames(giants)[i],
  myDateText[i],"\n")

# Two-Digit Years

my1969 <- mdy("08/31/69")
my1969
my1968 <- mdy("08/31/68")
my1968
my1968 <- my1968 - as.duration(100 * 365.2425 * 24 * 60 * 60)
my1968

as.POSIXlt("08/31/68", format="%m/%d/%y")
as.POSIXlt("08/31/69", format="%m/%d/%y")

as.POSIXlt("08/31/69", format="%m/%d/%y")
```

11

Enhancing Your Output

As we have seen, compared to SAS or SPSS, R output is quite sparse and not nicely formatted for word processing. You can improve R's output by adding value and variable labels. You can also format the output to make beautiful tables to use with word processors, Web pages, and document preparation systems.

11.1 Value Labels or Formats (and Measurement Level)

This chapter blends two topics because in R they are inseparable. In both SAS and SPSS, assigning labels to values is independent of the variable's measurement level. In R, you can assign value labels only to variables whose measurement level is factor. To be more precise, only objects whose class is factor can have label attributes.

In SAS, a variable's measurement level of nominal, ordinal, or interval is not stored. Instead, you list the variable on a specific statement, such as CLASS or BY, to tell SAS that you wish to view it as categorical.

SAS's use of value labels is a two-step process. First, PROC FORMAT creates a *format* for every unique set of labels. Then the FORMAT statement assigns a format to each variable or set of variables (see example program Sect. 11.1.6). The formats are stored outside the data set in a format library.

In SPSS, the VARIABLE LEVEL command sets the measurement level, but this is merely a convenience to help its GUI work well. Newer SPSS procedures take advantage of this information and do not show you nominal variables in a dialog box that it considers inappropriate. However, if you enter them into the same procedure using the programming language, it will accept them. As with SAS, special commands tell SPSS how you want to view the scale of the data. These include GROUPS and BY.

Independently, the VALUE LABEL command sets labels for each level, and the labels are stored within the data set itself as an attribute.

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_11, © Springer Science+Business Media, LLC 2011

R has a measurement level of factor for nominal data, ordered factor for ordinal data, and numeric for interval or scale data. You set these in advance and then the statistical and graphical procedures use them in an appropriate way automatically. When creating a factor, assigning labels is optional. If you do not use labels, a variable's original values are stored as character labels. R stores value labels in the factor itself.

11.1.1 Character Factors

Let us review how the read.table function deals with character variables. If we do not tell the function what to do, it will convert all character data to factors.

```
> mydata <- read.table("mydata.tab")

> mydata

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
2        2      f  2  1  4  1
3        1      f  2  2  4  3
4        2   <NA>  3  1 NA  3
5        1      m  4  5  2  4
6        2      m  5  4  5  5
7        1      m  5  3  4  4
8        2      m  4  5  5  5
```

You cannot tell what gender is by looking at it, but the class function can tell us that gender is a factor.

```
> class( mydata[ ,"gender"] )

[1] "factor"
```

In our case, this is helpful. However, there are times when you want to leave character data as simple characters. When reading people's names or addresses from a database, for example, you do not want to store them as factors. The argument stringsAsFactors = FALSE will tell the read.table function to leave such variables as characters.

```
> mydata2 <- read.table("mydata.tab",
+   stringsAsFactors = FALSE)

> mydata2

  workshop gender q1 q2 q3 q4
1        1      f  1  1  5  1
```

```
2         2      f  2  1  4  1
3         1      f  2  2  4  3
4         2   <NA>  3  1 NA  3
5         1      m  4  5  2  4
6         2      m  5  4  5  5
7         1      m  5  3  4  4
8         2      m  4  5  5  5
```

This determines how all of the character variables are read, so if some of them do need to be factors, you can convert those afterward. The data look the same, but the `class` function can verify that gender is indeed now a character variable.

```
> class( mydata2[ ,"gender"] )

[1] "character"
```

Many functions will not do what you expect with character data. For example, we have seen the `summary` function count factor levels. However, it will not count them in character class (i.e. as string variables). If these were names or addresses, there would be little use in counting them to see that they were virtually all unique. Instead, `summary` simply gives you the grand total (length) and the variable's class and mode:

```
> summary( mydata2$gender )

   Length     Class      Mode
        8 character character
```

We will focus on the first data frame that has gender as a factor. As `read.table` scans the data, it assigns the numeric values to the character values in alphabetical order. R always uses the numbers 1, 2, ... However, if you use the factor in an analysis, say as a regression predictor, it will do a proper coding for you automatically. For character data, these defaults are often sufficient. If you supply the `levels` argument in the `factor` function, you can use it to specify the order of label assignment:

```
mydata$genderF <- factor(
  mydata$gender,
  levels = c("m",   "f"     ),
  labels = c("Male","Female") )
```

The `factor` function call above has three arguments:

1. The name of the factor.
2. The `levels` argument with the levels *in order*. Since "m" appears first, it will be associated with 1 and "f" with 2. This is the order the values will appear in for frequency tables and bar plots. If used in modeling, the first

level that appears will be the comparison level. For example, in a linear regression gender as coded here would show how females are different from males. For details, see Sect. 10.8. If the variable is actually ordinal rather than nominal or categorical, then use the similar `ordered` rather than `factor`

3. The `labels` argument provides labels in the same order as the `levels` argument. This example sets "m" as 1 and "f" as 2 and uses the fully written out labels. If you leave the labels out, R will use the levels themselves as labels. In this case, the labels would be simply "m" and "f." That is how we will leave our practice data set, as it keeps the examples shorter.

There is a danger in setting value labels with character data that does not appear at all in SAS or SPSS. In the function call above, if we instead set

```
levels = c("m", "F")
```

R would set the values for all females to missing (NA) because the actual values are lowercase. There are no capital Fs in the data! This danger applies, of course, to other, more obvious, misspellings.

11.1.2 Numeric Factors

Workshop is a categorical measure, but initially R assumes it is numeric because it is entered as 1 and 2. If we do any analysis on workshop, it will be as an integer variable:

```
> class( mydata$workshop )

[1] "integer"

> summary( mydata$workshop )

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     1.0     1.5     1.5     2.0     2.0
```

We can nest a call to the `as.factor` function into any analysis to overcome this problem:

```
> summary( as.factor(mydata$workshop) )

1 2
4 4
```

So we see four people took workshops 1 and 2. The values 1 and 2 are merely labels at this point.

We can use the `factor` function to convert it to a factor and optionally assign labels. Factor labels in R are stored in the factor itself.

```
mydata$workshop <- factor(
  mydata$workshop,
  levels = c( 1,   2,     3,      4     ),
  labels = c("R", "SAS", "SPSS", "Stata")
)
```

Notice that we have assigned four labels to four values even though our data only contain the values 1 and 2. This can ensure consistent value label assignments regardless of the data we currently have on hand. If we collected another data set in which people only took SPSS and Stata workshops, we would not want 1=R in one data set and 1=SPSS in another!

Since R appears first, for modeling purposes it is the comparison level. A linear regression would then include effects for SAS, SPSS, and Stata, all in comparison to R. For details, see Sect. 10.8.

To simplify our other examples, we will use only the labels that appear in this small data set:

```
mydata$workshop <- factor(
  mydata$workshop,
  levels = c( 1,   2   ),
  labels = c("R", "SAS")
)
```

Now let us convert our q variables to factors. Since we will need to specify the same levels repeatedly, let us put them in a variable.

```
> myQlevels <- c(1, 2, 3, 4, 5)

> myQlevels

[1] 1 2 3 4 5
```

Now we will do the same for the labels:

```
> myQlabels <- c("Strongly Disagree",
+                "Disagree",
+                "Neutral",
+                "Agree",
+                "Strongly Agree")

> myQlabels

[1] "Strongly Disagree"  "Disagree"  "Neutral"
[4] "Agree"  "Strongly Agree"
```

Finally, we will use the `ordered` function to complete the process. It works just like the `factor` function but tells R that the data values have order. In statistical terms, it sets the variable's measurement level to ordinal. Ordered factors allow you to perform logical comparisons of greater than or less than. R's statistical modeling functions also try to treat ordinal data appropriately. However, some methods are influenced by the assumption that the ordered values are equally spaced, which may not be the case in your data.

We will put the factors into new variables with an "f" for *f*actor in their names, like qf1 for q1. The f is just there to help us remember; it has no meaning to R. We will keep both sets of variables because people who do survey research often want to view this type of variable as numeric for some analyses and as categorical for others. It is not necessary to do this if you prefer converting back and forth on the fly:

```
> mydata$qf1 <- ordered(mydata$q1, myQlevels, myQlabels)
> mydata$qf2 <- ordered(mydata$q2, myQlevels, myQlabels)
> mydata$qf3 <- ordered(mydata$q3, myQlevels, myQlabels)
> mydata$qf4 <- ordered(mydata$q4, myQlevels, myQlabels)
```

Now we can use the `summary` function to get frequency tables on them, complete with value labels:

```
> summary( mydata[ c("qf1", "qf2", "qf3", "qf4") ] )

               qf1                      qf2                      qf3
 Strongly Disagree:1   Strongly Disagree:3   Strongly Disagree:0
 Disagree         :2   Disagree         :1   Disagree         :1
 Neutral          :1   Neutral          :1   Neutral          :0
 Agree            :2   Agree            :1   Agree            :3
 Strongly Agree   :2   Strongly Agree   :2   Strongly Agree   :3
                                             NA s            :1
               qf4
 Strongly Disagree:3
 Disagree         :0
 Neutral          :2
 Agree            :2
 Strongly Agree   :1
```

11.1.3 Making Factors of Many Variables

The approach used above works fine for small numbers of variables. However, if you have hundreds, it is needlessly tedious. We do the same thing again, this time in a form that would handle any number of variables whose names follow the format string1 to stringN. Our practice data only have q1 to q4, but the same number of commands would handle q1 to q4000.

First, we will generate variable names to use. We will use qf to represent the q variables in factor form. This is an optional step, needed only if you want to keep the variables in both forms:

```
> myQnames <- paste("q",  1:4, sep = "")

> myQnames

[1] "q1" "q2" "q3" "q4"

> myQFnames <- paste("qf", 1:4, sep = "")

> myQFnames

[1] "qf1" "qf2" "qf3" "qf4"
```

Now we will use the myQnames character vector as column names to select from our data frame. We will store those in a separate data frame:

```
> myQFvars <- mydata[ ,myQnames]

> myQFvars

  q1 q2 q3 q4
1  1  1  5  1
2  2  1  4  1
3  2  2  4  3
4  3  1 NA  3
5  4  5  2  4
6  5  4  5  5
7  5  3  4  4
8  4  5  5  5
```

Next, we will use the myQFnames character vector to rename these variables:

```
> names(myQFvars) <- myQFnames

> myQFvars

  qf1 qf2 qf3 qf4
1   1   1   5   1
2   2   1   4   1
3   2   2   4   3
4   3   1  NA   3
5   4   5   2   4
6   5   4   5   5
7   5   3   4   4
8   4   5   5   5
```

Now we need to make up a function to apply to the variables:

```
myLabeler <- function(x) {
  ordered(x, myQlevels, myQlabels)
}
```

The myLabeler function will apply myQlevels and myQlabels (defined in Sect. 11.1.2) to any variable, x, that we supply to it. Let us try it on a single variable:

```
> summary( myLabeler( myQFvars[ ,"qf1"] ) )

Strongly Disagree     Disagree      Neutral        Agree
                1            2            1            2
   Strongly Agree
                2
```

It is important to understand that the myLabeler function will work only on vectors, since the ordered function requires them. Removing the comma in the above command would select a data frame containing only q1f, instead of a vector, and this would not work.

```
> summary( myLabeler(myQFvars["qf1"]) )  # Does not work!

Strongly Disagree     Disagree      Neutral        Agree
                0            0            0            0
   Strongly Agree         NA s
                0            1
```

Now we will use the sapply function to apply myLabeler to myQFvars.

```
myQFvars <- data.frame( sapply( myQFvars, myLabeler ) )
```

The sapply function simplified the result to a matrix and the data.frame function converted that to a data frame. Now the summary function will count the values and display their labels:

```
> summary(myQFvars)

               qf1                  qf2                qf3
 Agree            :2  Agree            :1  Agree         :3
 Disagree         :2  Disagree         :1  Disagree      :1
 Neutral          :1  Neutral          :1  Strongly Agree:3
 Strongly Agree   :2  Strongly Agree   :2  NA s          :1
 Strongly Disagree:1  Strongly Disagree:3
               qf4
 Agree            :2
 Neutral          :2
 Strongly Agree   :2
 Strongly Disagree:2
```

If you care to, you can bind myQFvars to our original data frame:

```
mydata <- cbind(mydata, myQFvars)
```

11.1.4 Converting Factors to Numeric or Character Variables

R has functions for converting factors to numeric or character vectors (variables). To extract the numeric values from a factor like gender, we can use the `as.numeric` function:

```
> mydata$genderNums <- as.numeric( mydata$gender )

> mydata$genderNums

[1]  1  1  1 NA  2  2  2  2
```

If we want to extract the labels themselves to use in a character vector, we can do so with the `as.character` function.

```
> mydata$genderChars <- as.character( mydata$gender)

> mydata$genderChars

[1] "f" "f" "f" NA  "m" "m" "m" "m"
```

We can apply the same two functions to variable qf1. Since we were careful to set all of the levels, even for those that did not appear in the data, this works fine. First, we will do it using `as.numeric`:

```
> mydata$qf1Nums <- as.numeric(mydata$qf1)

> mydata$qf1Nums

[1] 1 2 2 3 4 5 5 4
```

Now let us do it again using the `as.character` function:

```
> mydata$qf1Chars <- as.character(mydata$qf1)

> mydata$qf1Chars

[1] "Strongly Disagree" "Disagree" "Disagree"
[4] "Neutral"           "Agree"    "Strongly Agree"
[7] "Strongly Agree"    "Agree"
```

Where you can run into trouble is when you do not specify the original numeric values, and they are not simply 1, 2, 3, etc. For example, let us create a variable whose original values are 10, 20, 30:

```
> x <- c(10, 20, 30)

> x

[1] 10 20 30

> xf <- as.factor(x)

> xf

[1] 10 20 30
Levels: 10 20 30
```

So far, the factor xf looks fine. However, when we try to extract the original values with the `as.numeric` function, we get, instead, the levels 1, 2, 3!

```
> as.numeric(xf)

[1] 1 2 3
```

If we use `as.character` to get the values, there are no nice value labels, so we get character versions of the original values:

```
> as.character(xf)

[1] "10" "20" "30"
```

If we want those original values in a numeric vector like the one we began with, we can use `as.numeric` to convert them. To extract the original values and store them in a variable x10, we can use:

```
> x10 <- as.numeric( as.character(xf) )

> x10

[1] 10 20 30
```

The original values were automatically stored as value labels. If you had specified value labels of your own, like low, medium, and high, the original values would have been lost. You would then have to recode the values to get them back. For details, see Sect. 10.7, "Recoding Variables."

11.1.5 Dropping Factor Levels

Earlier in this chapter, we created labels for factor levels that did not exist in our data. While this is not at all necessary, it would be helpful if you were to enter more data or merge your data frame with others that have a full set of values. However, when using such variables in analysis, it is often helpful to get rid of such empty levels. To get rid of the unused levels, append `[ , drop = TRUE]` to the variable reference. For example, if you want to include the empty levels, skip the `drop` argument:

```
> summary(workshop)

    R   SAS  SPSS Stata
    4     4     0     0
```

However, when you need to get rid of empty levels, add the `drop` argument:

```
> summary( workshop[ , drop = TRUE] )

  R SAS
  4   4
```

11.1.6 Example Programs for Value Labels

SAS Program to Assign Value Labels

```
* ValueLabels.sas ;

LIBNAME myLib  C:\myRfolder ;

PROC FORMAT;
  VALUES workshop_f 1="Control" 2="Treatment"
  VALUES $gender_f "m"="Male" "f"="Female";
  VALUES agreement
    1= Strongly Disagree
    2= Disagree
    3= Neutral
    4= Agree
    5= Strongly Agree .;
RUN;

DATA  myLib.mydata;
  SET myLib.mydata;
  FORMAT workshop workshop_f. gender gender_f.
  q1-q4 agreement.;
RUN;
```

SPSS program to Assign Value Labels

```
* ValueLabels.sps
```

```
CD C:\myRfolder .
GET FILE= mydata.sav .

VARIABLE LEVEL workshop (NOMINAL)
 /q1 TO q4 (SCALE).

VALUE LABELS  workshop 1  Control   2  Treatment
 /q1 TO q4
 1  Strongly Disagree
 2  Disagree
 3  Neutral
 4  Agree
 5  Strongly Agree .

SAVE OUTfile = "mydata.sav".
```

11.1.7 R Program to Assign Value Labels and Factor Status

```
# Filename: ValueLabels.R

setwd("c:/myRfolder")

# Character Factors

# Read gender as factor.
mydata <- read.table("mydata.tab")
mydata
class( mydata[ , "gender"] )

# Read gender as character.
mydata2 <- read.table("mydata.tab", as.is = TRUE)
mydata2
class( mydata2[ , "gender"] )
summary( mydata2$gender )
rm(mydata2)

# Numeric Factors

class( mydata$workshop )
summary( mydata$workshop )

summary( as.factor(mydata$workshop) )

# Now change workshop into a factor:
```

```
mydata$workshop <- factor( mydata$workshop,
  levels = c( 1,   2,     3,      4),
  labels = c("R", "SAS", "SPSS", "Stata") )
mydata

# Now see that summary only counts workshop attendance.
summary(mydata$workshop)

# Making the Q Variables Factors

# Store levels to use repeatedly.
myQlevels <- c(1, 2, 3, 4, 5)
myQlevels

# Store labels to use repeatedly.
myQlabels <- c("Strongly Disagree",
               "Disagree",
               "Neutral",
               "Agree",
               "Strongly Agree")
myQlabels

# Now create a new set of variables as factors.
mydata$qf1 <- ordered(mydata$q1, myQlevels, myQlabels)
mydata$qf2 <- ordered(mydata$q2, myQlevels, myQlabels)
mydata$qf3 <- ordered(mydata$q3, myQlevels, myQlabels)
mydata$qf4 <- ordered(mydata$q4, myQlevels, myQlabels)

# Get summary and see that workshops are now counted.
summary( mydata[ c("qf1", "qf2", "qf3", "qf4") ] )

# Making Factors of Many Variables

# Generate two sets of var names to use.
myQnames  <- paste( "q",  1:4, sep = "")
myQnames
myQFnames <- paste( "qf", 1:4, sep = "")
myQFnames

# Extract the q variables to a separate data frame.
myQFvars <- mydata[ ,myQnames]
myQFvars

# Rename all of the variables with F for Factor.
names(myQFvars) <- myQFnames
```

```
myQFvars

# Create a function to apply the labels to lots of variables.
myLabeler <- function(x) {
  ordered(x, myQlevels, myQlabels)
}

# Here is how to use the function on one variable.

summary( myLabeler(myQFvars[ ,"qf1"]) )
summary( myLabeler(myQFvars["qf1"]) )  # Does not work!

# Apply it to all of the variables.
myQFvars <- data.frame( sapply( myQFvars, myLabeler ) )

# Get summary again, this time with labels.
summary(myQFvars)

# You can even join the new variables to mydata.
mydata <- cbind(mydata, myQFvars)
mydata

#---Converting Factors into Character or Numeric Variables

# Converting the gender factor, first with as.numeric.

mydata$genderNums <- as.numeric(mydata$gender)
mydata$genderNums

# Again with as.character.

mydata$genderChars <- as.character(mydata$gender)
mydata$genderChars

# Converting the qf1 factor.

mydata$qf1Nums <- as.numeric(mydata$qf1)
mydata$qf1Nums

mydata$qf1Chars <- as.character(mydata$qf1)
mydata$qf1Chars

# Example with bigger values.
x <- c(10, 20, 30)
x
```

```
xf <- factor(x)
xf

as.numeric(xf)
as.character(xf)
x10 <- as.numeric( as.character(xf) )
x10
```

11.2 Variable Labels

Perhaps the most fundamental feature missing from the main R distribution is support for variable labels. It does have a *comment attribute* that you can apply to each variable, but only the `comment` function itself will display it.

In SAS or SPSS, you might name a variable BP and want your publication-ready output to display "Systolic Blood Pressure" instead. SAS does this using the LABEL statement and SPSS does it using the very similar VARIABLE LABELS command.

Survey researchers in particular rely on variable labels. They often name their variables Q1, Q2, and so on and assign labels as the full text of the survey items. R is the only statistics package that I am aware of that lacks such a feature.

It is a testament to R's openness and flexibility that a user can add such a fundamental feature. Frank Harrell did just that in his `Hmisc` package [32]. It adds a *label attribute* to the data frame and stores the labels there, even converting them from SAS data sets automatically (but not SPSS data sets). As amazing as this addition is, the fact that variable labels were not included in the main distribution means that most procedures do not take advantage of what `Hmisc` adds. The many wonderful functions in the `Hmisc` package do, of course. The `Hmisc` package creates variable labels using its `label` function:

```
library("Hmisc")

label(mydata$q1) <- "The instructor was well prepared."
label(mydata$q2) <- "The instructor communicated well."
label(mydata$q3) <- "The course materials were helpful."
label(mydata$q4) <- "Overall, I found this workshop useful."
```

Now the `Hmisc describe` function will take advantage of the labels:

```
> describe( mydata[ ,3:6] )

mydata[ ,3:6]

 4  Variables      8  Observations
-------------------------------------------------------
```

```
q1 : The instructor was well prepared.
      n missing  unique    Mean
      8       0       5    3.25
           1  2  3  4  5
Frequency  1  2  1  2  2
%         12 25 12 25 25
-------------------------------------------------------
q2 : The instructor communicated well.
      n missing  unique    Mean
      8       0       5    2.75
           1  2  3  4  5
Frequency  3  1  1  1  2
%         38 12 12 12 25
-------------------------------------------------------
q3 : The course materials were helpful.
      n missing  unique    Mean
      7       1       3   4.143
2 (1, 14%), 4 (3, 43%), 5 (3, 43%)
-------------------------------------------------------
q4 : Overall, I found this workshop useful.
      n missing  unique    Mean
      8       0       4    3.25
1 (2, 25%), 3 (2, 25%), 4 (2, 25%), 5 (2, 25%)
-------------------------------------------------------
```

Unfortunately, built-in functions such as `summary` ignore the labels.

```
> summary( mydata[ ,3:6] )

       q1              q2              q3              q4
 Min.   :1.00   Min.   :1.00   Min.   :2.00   Min.   :1.00
 1st Qu.:2.00   1st Qu.:1.00   1st Qu.:4.00   1st Qu.:2.50
 Median :3.50   Median :2.50   Median :4.00   Median :3.50
 Mean   :3.25   Mean   :2.75   Mean   :4.14   Mean   :3.25
 3rd Qu.:4.25   3rd Qu.:4.25   3rd Qu.:5.00   3rd Qu.:4.25
 Max.   :5.00   Max.   :5.00   Max.   :5.00   Max.   :5.00
                               NA s   :1.00
```

A second approach to variable labels is to store them as character variables. That is the approach used by Jim Lemon and Philippe Grosjean's `prettyR` package. Its use is beyond our scope.

Finally, you can create variable names of any length by enclosing them in quotes. This has the advantage of working with most R functions:

```
> names(mydata) <- c("Workshop","Gender",
+   "The instructor was well prepared.",
+   "The instructor communicated well.",
+   "The course materials were helpful.",
+   "Overall, I found this workshop useful.")
```

Notice here that names like q1, q2, etc. do not exist now. The labels *are* the variable names:

```
> names(mydata)

[1] "Workshop"
[2] "Gender"
[3] "The instructor was well prepared."
[4] "The instructor communicated well."
[5] "The course materials were helpful."
[6] "Overall, I found this workshop useful."
```

Now many R functions, even those built in, will use the labels. Here we select the variables by their numeric index values rather than their names:

```
> summary( mydata[ ,3:6] )
 The instructor was well prepared.
 Min.   :1.00
 1st Qu.:2.00
 Median :3.50
 Mean   :3.25
 3rd Qu.:4.25
 Max.   :5.00

 The instructor communicated well.
 Min.   :1.00
 1st Qu.:1.00
 Median :2.50
 Mean   :2.75
 3rd Qu.:4.25
 Max.   :5.00
...
```

You can still select variables by their names, but now typing the whole name out is an absurd amount of work!

```
> summary( mydata["Overall, I found this workshop useful."] )

 Overall, I found this workshop useful.
 Min.   :1.00
 1st Qu.:2.50
 Median :3.50
 Mean   :3.25
 3rd Qu.:4.25
 Max.   :5.00
```

In addition to selecting variables by their index number, it is also easy to search for keywords in long variable names using the `grep` function. For details, see Sect. 7.6, "Selecting Variables by String Search." The `grep` function call below finds the two variable names containing the string "instructor" in variables 3 and 4 and then stores their locations in a numeric index vector:

```
> myvars <- grep(  instructor , names(mydata) )

> myvars

[1] 3 4
```

Now we can use those indices to analyze the selected variables:

```
> summary ( mydata[myvars] )

 The instructor was well prepared.
 Min.   :1.00
 1st Qu.:2.00
 Median :3.50
 Mean   :3.25
 3rd Qu.:4.25
 Max.   :5.00

 The instructor communicated well.
 Min.   :1.00
 1st Qu.:1.00
 Median :2.50
 Mean   :2.75
 3rd Qu.:4.25
 Max.   :5.00
```

Some important R functions, such as `data.frame`, convert the spaces in the labels to periods.

```
> newdata <- data.frame( mydata )

> names( newdata[ ,3:6] )
[1] "The.instructor.was.well.prepared."
[2] "The.instructor.communicated.well."
[3] "The.course.materials.were.helpful."
[4] "Overall..I.found.this.workshop.useful."
```

To avoid this change, you can add the `check.names = FALSE` argument to the `data.frame` function call:

```
> newdata <- data.frame(mydata, check.names = FALSE)

> names( newdata[ ,3:6] )

[1] "The instructor was well prepared."
[2] "The instructor communicated well."
[3] "The course materials were helpful."
[4] "Overall, I found this workshop useful."
```

11.2.1 Other Packages That Support Variable Labels

There are at least two other packages that support variable labels. Unfortunately, I have not had a chance to fully evaluate them. Martin Elff's memisc package [17] offers a wide range of tools for survey research. Jim Lemon and Philippe Grosjean's prettyR package [38] stores variable labels as character variables.

11.2.2 Example Programs for Variable Labels

SAS Program for Variable Labels

```
* Filename: VarLabels.sas ;

LIBNAME myLib  C:\myRfolder ;

DATA  myLib.mydata;
  SET myLib.mydata ;
  LABEL
    Q1="The instructor was well prepared"
    Q2="The instructor communicated well"
    Q3="The course materials were helpful"
    Q4="Overall, I found this workshop useful";
RUN;

PROC FREQ; TABLES q1-q4; RUN;
RUN;
```

SPSS Program for Variable Labels

```
* Filename: VarLabels.sps .

CD  C:\myRfolder .

GET FILE= mydata.sav .
VARIABLE LABELS
```

```
  Q1 "The instructor was well prepared"
  Q2 "The instructor communicated well"
  Q3 "The course materials were helpful"
  Q4 "Overall, I found this workshop useful".

FREQUENCIES  VARIABLES=q1 q2 q3 q4.
SAVE OUTFILE= mydata.sav .
```

R Program for Variable Labels

```
# Filename: VarLabels.R

setwd("c:/myRfolder")
load(file = "mydata.RData")
options(width = 63)
mydata

# Using the Hmisc label attribute.
library("Hmisc")
label(mydata$q1) <- "The instructor was well prepared."
label(mydata$q2) <- "The instructor communicated well."
label(mydata$q3) <- "The course materials were helpful."
label(mydata$q4) <-
  "Overall, I found this workshop useful."

# Hmisc describe function uses the labels.
describe( mydata[ ,3:6] )

# Buit-in summary function ignores the labels.
summary( mydata[ ,3:6] )

#Assign long variable names to act as variable labels.
names(mydata) <- c("Workshop","Gender",
  "The instructor was well prepared.",
  "The instructor communicated well.",
  "The course materials were helpful.",
  "Overall, I found this workshop useful.")

names(mydata)

# Now summary uses the long names.
summary( mydata[ ,3:6] )

# You can still select variables by name.
summary( mydata["Overall, I found this workshop useful."] )
```

```
# Searching for strings in long variable names.
myvars <- grep( instructor , names(mydata))
myvars
summary ( mydata[myvars] )

# Data.frame replaces spaces with periods.
newdata <- data.frame( mydata )
names( newdata[ , 3:6] )

# Data.frame now keeps the spaces.
newdata <- data.frame( mydata, check.names = FALSE )
names( newdata[ , 3:6] )
```

11.3 Output for Word Processing and Web Pages

Hypertext Markup Language, or *HTML*, is the format used for displaying results on Web pages. Most word processors can also easily incorporate HTML files. For technical writing, the popular LaTeX document preparation system and the LyX document processor share a common format for displaying tables of results.

In SAS or SPSS, getting tabular results into your word processor or Web page is as easy as setting the style when you install it and then saving the output directly in format you need. For get a subset of output copy and paste works fine.

In R, the output in the console is just text. It does not even have tabs between columns. You can cut and paste it into a word processor, but it would only appear in neat columns when displayed with a monospaced font like Courier. However, there are a number of packages and functions that enable you to create nicely formatted results.

Table 11.1. Our practice data set printed in LaTeX

	workshop	gender	q1	q2	q3	q4
1	R	f	1.00	1.00	5.00	1.00
2	SAS	f	2.00	1.00	4.00	1.00
3	R	f	2.00	2.00	4.00	3.00
4	SAS		3.00	1.00		3.00
5	R	m	4.00	5.00	2.00	4.00
6	SAS	m	5.00	4.00	5.00	5.00
7	R	m	5.00	3.00	4.00	4.00
8	SAS	m	4.00	5.00	5.00	5.00

11.3.1 The xtable Package

Dahl's xtable package [15] can convert your R results to both HTML and LaTeX. What follows is our practice data frame as it appears from the print function.

```
> print(mydata)

  workshop gender q1 q2 q3 q4
1        R      f  1  1  5  1
2      SAS      f  2  1  4  1
3        R      f  2  2  4  3
4      SAS   <NA>  3  1 NA  3
5        R      m  4  5  2  4
6      SAS      m  5  4  5  5
7        R      m  5  3  4  4
8      SAS      m  4  5  5  5
```

In Table 11.1, you can see mydata printed again after loading the xtable package. How did we get such nice output? The following are the function calls and output:

```
> library("xtable")

> myXtable <- xtable(mydata)

> print(myXtable, type="latex")

% latex table generated in R 2.9.2 by xtable 1.5-5 package
% Sat Oct 17 11:55:58 2009
\begin{table}[ht]
\begin{center}
\begin{tabular}{rllrrrr}
  \hline
 & workshop & gender & q1 & q2 & q3 & q4 \\
  \hline
1 & R & f & 1.00 & 1.00 & 5.00 & 1.00 \\
  2 & SAS & f & 2.00 & 1.00 & 4.00 & 1.00 \\
  3 & R & f & 2.00 & 2.00 & 4.00 & 3.00 \\
  4 & SAS &  & 3.00 & 1.00 &  & 3.00 \\
  5 & R & m & 4.00 & 5.00 & 2.00 & 4.00 \\
  6 & SAS & m & 5.00 & 4.00 & 5.00 & 5.00 \\
  7 & R & m & 5.00 & 3.00 & 4.00 & 4.00 \\
  8 & SAS & m & 4.00 & 5.00 & 5.00 & 5.00 \\
   \hline
```

```
\end{tabular}
\end{center}
\end{table}
```

Notice that the print command has a new method for handling objects with a class of *xtable*, and a new type argument. The value, type="latex" is the default, so we did not need to list it here. However, it makes it easy to guess that type="html" is the way to get HTML output.

The LaTeX output begins with the line "% latex table generated in R. . . ." If you are not accustomed to the LaTeX language, it may see quite odd, but when we pasted it into this chapter, it came out looking nice.

If we had instead used type="html", we could have used the output on Web pages or in most any word processor.

The following is another example, this time using the output from a linear model that we will examine later in Sect. 17.5, "Linear Regression."

```
> mymodel <- lm( q4~q1 + q2 + q3, data = mydata)

> myXtable <- xtable(mymodel)
```

Before printing the result, let us add to it a label and caption. If this were an HTML file, these commands would add an anchor and a title, respectively:

```
> label(myXtable) <- c("xtableOutput")

> caption(myXtable) <-
+   c("Linear model results formatted by xtable.")

> print(myXtable,type="latex")

% latex table generated in R 2.9.2 by xtable 1.5-5 package
% Sat Oct 17 12:49:59 2009
\begin{table}[ht]
\begin{center}
\begin{tabular}{rrrrr}
  \hline
 & Estimate & Std. Error & t value & Pr($>$$|$t$|$) \\
  \hline
(Intercept) & -1.3243 & 1.2877 & -1.03 & 0.3794 \\
  q1 & 0.4297 & 0.2623 & 1.64 & 0.1999 \\
  q2 & 0.6310 & 0.2503 & 2.52 & 0.0861 \\
  q3 & 0.3150 & 0.2557 & 1.23 & 0.3058 \\
   \hline
\end{tabular}
\caption{Linear model results formatted by xtable.}
\label{xtableOutput}
```

```
\end{center}
\end{table}
```

The caption and label appear near the bottom of the output. To match the style of the rest of this book, I moved the caption to the top in the tables themselves. You can see the nicely formatted result in Table 11.2. I cheated a bit on the previous table by adding the caption manually.

Table 11.2. Linear model results formatted by `xtable`

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-1.3243	1.2877	-1.03	0.3794
q1	0.4297	0.2623	1.64	0.1999
q2	0.6310	0.2503	2.52	0.0861
q3	0.3150	0.2557	1.23	0.3058

11.3.2 Other Options for Formatting Output

There several other packages that can provide nicely formatted output. Koenker's `quantreg` package [34] includes the `latex.table` function, which creates LaTeX output. Lecoutre'S `R2HTML` package [36] includes over a dozen functions for writing output in HTML format.

Perhaps the most interesting option to get nice output is the "Weave" family of software. The packages in this family allow you to weave, or blend, R commands or output into your word processing or text formatting documents. You then see your results appear in high-quality output right in the middle of your documents. Of course, if you change your data, then you had better be sure the text that describes the results actually matches the new output! You can choose to display the R commands – if, say you are teaching R – or display only their output. This not only gives you publishable results, but it also assures the reproducibility of those results.

Leisch's `Sweave` function [37] comes with the main R installation. It allows you to "weave" your R program into your LaTeX or LyX file. Similarly, Baier's SWord software [4] allows you to weave your R code into beautiful Word documents. The `odfWeave` package [35] by Kuhn and Weaston works similarly to `SWord` but using the *Open Document Format*, or *ODF*. Many word processors, including Microsoft Word, LibreOffice, and OpenOffice, can work with ODF files.

11.3.3 Example Program for Formatting Output

This section usually shows programs in SAS, SPSS, and R. However, SAS and SPSS can both display their output already formatted for Web pages or word processors. Therefore, I show only an R program here.

```
# Filename: FormattedOutput.R

options(width = 60)
setwd("c:/myRfolder")
load("mydata.RData")
attach(mydata)

library("xtable")

# Formatting a Data Frame

print(mydata)
myXtable <- xtable(mydata)
class(myXtable)

print(myXtable, type = "html")
print(myXtable, type = "latex")

# Formatting a Linear Model

mymodel <- lm( q4 ~ q1 + q2 + q3, data = mydata)
myXtable <- xtable(mymodel)

label(myXtable) <- c("xtableOutput")
caption(myXtable) <-
  c("Linear model results formatted by xtable")

print(myXtable, type = "html")
print(myXtable, type = "latex")
```

12

Generating Data

Generating data is far more important to R users than it is to SAS or SPSS users. As we have seen, many R functions are controlled by numeric, character, or logical vectors. You can generate those vectors using the methods in this chapter, making quick work of otherwise tedious tasks.

What follows are some of the ways we have used vectors as arguments to R functions.

To create variable names that follow a pattern like q1, q2, etc. using the `paste` function For an example, see Chap. 7.

To create a set of index values to select variables, as with `mydata[ ,3:6]`. Here the colon operator generates the simple values 3, 4, 5, 6. The methods I present in this chapter can generate a much wider range of patterns.

To provide sets of column widths when reading data from fixed-width text files (Sect. 6.6.1). Our example used a very small vector of value widths, but the methods in this chapter could generate long sets of patterns to read hundreds of variables with a single command.

To create variables to assist in reshaping data, see Sect. 10.17.

Generating data is also helpful in more general ways for situations similar to those in other statistics packages.

Generating data allows you to demonstrate various analytic methods, as I have done in this book. It is not easy to find one data set to use for all of the methods you might like to demonstrate.

Generating data lets you create very small examples that can help you debug an otherwise complex problem. Debugging a problem on a large data set often introduces added complexities and slows down each new attempted execution. When you can demonstrate a problem with the smallest possible set of data, it helps you focus on the exact nature of the problem. This is usually the best way to report a problem when requesting technical assistance. If possible, provide a small generated example when you ask questions to the R-help e-mail support list.

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_12, © Springer Science+Business Media, LLC 2011

> When you are designing an experiment, the levels of the experimental variables usually follow simple repetitive patterns. You can generate those and then add the measured outcome values to them later manually. With such a nice neat data set to start with, it is tempting to collect data in that order. However, it is important to collect it in random order whenever possible so that factors such as human fatigue or machine wear do not bias the results of your study.

Some of our data generation examples use R's random number generator. It will give a different result each time you use it unless you use the `set.seed` function before each function that generates random numbers.

12.1 Generating Numeric Sequences

SAS generates data using DO loops in a data step. SPSS uses input programs to generate data. You can also use SPSS's strong links to Python to generate data. R generates data using specialized functions. We have used the simplest one: the colon operator. We can generate a simple sequence with

```
> 1:10

 [1]  1  2  3  4  5  6  7  8  9 10
```

You can store the results of any of our data generation examples in a vector using the assignment operator. So we can create the ID variable we used with

```
> id <- 1:8

> id

[1] 1 2 3 4 5 6 7 8
```

The `seq` function generates sequences like this, too, and it offers more flexibility. What follows is an example:

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

 [1]  1  2  3  4  5  6  7  8  9 10
```

The `seq` function call above has three arguments:

1. The `from` argument tells it where to begin the sequence.
2. The `to` argument tells it where to stop.
3. The `by` argument tells it the increments to use between each number.

The following is an example that goes from 5 to 50 in increments of 5:

```
> seq(from = 5, to = 50, by = 5)

 [1]  5 10 15 20 25 30 35 40 45 50
```

Of course, you do not need to name the arguments if you use them in order. So you can do the above example using this form, too.

```
> seq(5, 50, 5)

 [1]  5 10 15 20 25 30 35 40 45 50
```

12.2 Generating Factors

The `gl` function *g*enerates *l*evels of factors. What follows is an example that generates the series 1, 2, 1, 2, etc.:

```
> gl(n = 2, k = 1, length = 8)

[1] 1 2 1 2 1 2 1 2

Levels: 1 2
```

The `gl` function call above has three arguments.

1. The `n` argument tells it how many levels your factor will have.
2. The `k` argument tells it how many of each level to repeat before incrementing to the next value.
3. The `length` argument is the total number of values generated. Although this would usually be divisible by n*k, it does not have to be.

To generate our gender variable, we just need to change k to be 4:

```
> gl(n = 2, k = 4, length = 8)

[1] 1 1 1 1 2 2 2 2

Levels: 1 2
```

There is also an optional `label` argument. Here we use it to generate workshop and gender, complete with value labels:

```
> workshop <- gl(n = 2, k = 1, length = 8,
+   label = c("R", "SAS") )

> workshop

[1] R   SAS R   SAS R   SAS R   SAS
```

```
Levels: R SAS

> gender <- gl(n = 2, k = 4, length = 8,
+   label = c("f", "m") )

> gender

[1] f f f f m m m m
Levels: f m
```

12.3 Generating Repetitious Patterns (Not Factors)

When you need to generate repetitious sequences of values, you are often creating levels of factors, which is covered in the previous section. However, sometimes you need similar patterns that are numeric, not factors. The `rep` function generates these:

```
> gender <- rep(1:2, each = 4, times = 1)

> gender

[1] 1 1 1 1 2 2 2 2
```

The call to the `rep` function above has three simple arguments.

1. The set of numbers to repeat. We have used the colon operator to generate the values 1 and 2. You could use the `c` function here to list any set of numbers you need. Note that you can use any vector here and it will repeat it, the numbers need not be sequential.
2. The `each` argument tells it often to repeat each number in the set. Here we need four of each number.
3. The `times` argument tells it the number of times to repeat the (`set` by `each`) combination. For this example, we only needed one.

Note that while we are generating the gender variable as an easy example, `rep` did not create gender as a factor. To make it one, you would have to use the `factor` function. You could instead generate it directly as a factor using the more appropriate `gl` function.

Next, we generate the workshop variable by repeating each number in the 1:2 sequence only one time each but repeat that set four times:

```
> workshop <- rep(1:2, each = 1, times = 4)

> workshop

[1] 1 2 1 2 1 2 1 2
```

By comparing the way we generated gender and workshop, the meaning of the rep function's arguments should become clear.

Now we come to the only example that we actually needed for the rep function in this book – to generate a constant! We use a variation of this in Chap. 15, "Traditional Graphics," to generate a set of zeros for use on a histogram:

```
> myZeros <- rep(0, each = 8)

> myZeros

[1] 0 0 0 0 0 0 0 0
```

12.4 Generating Values for Reading Fixed-Width Files

In Section6.6 we read a data file that had the values for each variable fixed in specific columns. Now that we know how to generate patterns of values, we can use that knowledge to read far more complex files.

Let us assume we are going to read a file that contains 300 variables we want to name x1, y1, z1, x2, y2, z2, ..., x100, y100, and z100. The values of x, y, and z are 8, 12, and 10 columns wide, respectively. To read such a file, our first task is to generate the prefixes and suffixes for each of the variable names:

```
> myPrefixes <- c("x", "y", "z")

> mySuffixes <- rep(1:300, each = 3, times = 1)

> head(mySuffixes, n = 20)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7
```

You can see that we have used the rep function in the same way as we discussed in the previous section. Now we want to combine these parts. We can do so using the paste function:

```
> myVariableNames <- paste(myPrefixes, mySuffixes, sep = "")

> head(myVars, n = 20)

[1] "x1" "y1" "z1" "x2" "y2" "z2" "x3" "y3" "z3" "x4" "y4" "z4"
[13] "x5" "y5" "z5" "x6" "y6" "z6" "x7" "y7"
```

You might think that you would need to have the same number of prefix and suffix pieces to match, but R will recycle a shorter vector over and over to make it match a longer vector. Now we need to get 300 values of our variable widths:

```
> myWidthSet <- c(8, 12, 10)

> myVariableWidths <- rep(myWidthSet, each = 1, times = 100)

> head(myRecord, n = 20)

 [1]  8 12  6  8 12  6  8 12  6  8 12  6  8 12  6  8 12  6  8 12
```

To make the example more realistic, let us assume there is also an ID variable in the first five columns of the file. We can add the name and the width using the `c` function:

```
myVariablenames  <- c("id", myVariableNames)
myVariableWidths <- c( 5  , myVariableWidths)
```

Now we are ready to read the file as we did back in Sect. 6.6:

```
myReallyWideData <- read.fwf(
   file  = "myReallyWideFile.txt,
   width = myVariableWidths,
   col.names  = myVariableNames,
   row.names  = "id")
```

12.5 Generating Integer Measures

R's ability to generate samples of integers is easy to use and is quite different from SAS's and SPSS's usual approach. It is similar to the SPSSINC TRANS extension command. You provide a list of possible values and then use the `sample` function to generate your data. First, we put the Likert scale values 1, 2, 3, 4, and 5 into myValues:

```
> myValues <- c(1, 2, 3, 4, 5)
```

Next, we set the random number seed using the `set.seed` function, so you can see the same result when you run it:

```
> set.seed(1234)  # Set random number seed.
```

Finally, we generate a random sample size of 1000 from myValues using the `sample` function. We are sampling with replacement so we can use the five values repeatedly:

```
> q1 <- sample(myValues, size = 1000, replace = TRUE)
```

Now we can check its mean and standard deviation:

```
> mean(q1)

[1] 3.029

> sd(q1)

[1] 1.412854
```

To generate a sample using the same numbers but with a roughly normal distribution, we can change the values to have more as we reach the center.

```
> myValues <- c(1, 2, 2, 3, 3, 3, 4, 4, 5)
> set.seed(1234)
> q2 <- sample( myValues, 1000, replace = TRUE)

> mean(q2)
[1] 3.012

> sd(q2)
[1] 1.169283
```

You can see from the bar plots in Fig. 12.1 that our first variable follows a uniform distribution (left), and our second one follows a normal distribution (right). Do not worry about how we created the plot; we will cover that in Chap. 15, "Traditional Graphics."

We could have done the latter example more precisely by generating 1000 samples from a normal distribution and then chopping it into 5 equally spaced groups. We will cover generating samples from continuous samples in the next section.

```
> boxplot( table(q1) )
> boxplot( table(q2) )
```

If you would like to generate two Likert-scale variables that have a mean difference, you can do so by providing them with different sets of values from which to sample. In the example below, I nest the call to the `c` function, to generate a vector of values, within the call to the `sample` function. Notice that the vector for q1 has no values greater than 3 and q2 has none less than 3. This difference will create the mean difference. Here I am only asking for a sample size of 8:

```
> set.seed(1234)

> q1 <- sample( c(1, 2, 2, 3), size = 8, replace = TRUE)

> mean(q1)
[1] 1.75
```

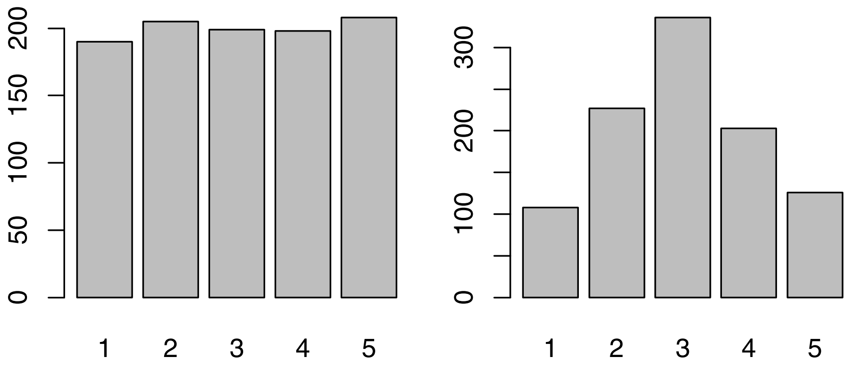


Fig. 12.1. Bar plots showing the distributions of our generated integer variables

```
> set.seed(1234)

> q2 <- sample( c(3, 4, 4, 5), size = 8, replace = TRUE)

> mean(q2)

[1] 3.75
```

12.6 Generating Continuous Measures

You can generate continuous random values from a uniform distribution using the `runif` function:

```
> set.seed(1234)

> x1 <- runif(n = 1000)

> mean(x1)
[1] 0.5072735

> sd(x1)
[1] 0.2912082
```

where the `n` argument is the number of values to generate. You can also provide `min` and `max` arguments to set the lowest and highest possible values, respectively. So you might generate 1000 pseudo test scores that range from 60 to 100 with

```
> set.seed(1234)

> x2 <- runif(n = 1000, min = 60, max = 100)

> mean(x2)
[1] 80.29094

> sd(x2)
[1] 11.64833
```

Normal distributions with a mean of 0 and standard deviation of 1 have many uses. You can use the `rnorm` function to generate 1000 values from such a distribution with

```
> set.seed(1234)

> x3 <- rnorm(n = 1000)

> mean(x3)
[1] -0.0265972

> sd(x3)
[1] 0.9973377
```

You can specify other means and standard deviations as in the following example:

```
> set.seed(1234)

> x4 <- rnorm(n = 1000, mean = 70, sd = 5)

> mean(x4)
[1] 69.86701

> sd(x4)
[1] 4.986689
```

We can use the `hist` function to see what two of these distributions look like, see Fig. 12.2:

```
> hist(x2)
> hist(x4)
```

12.7 Generating a Data Frame

Putting all of the above ideas together, we can use the following commands to create a data frame similar to our practice data set, with a couple of test scores added. I am not bothering to set the random number generator seed, so each time you run this, you will get different results.

```
> id <- 1:8

> workshop <- gl( n=2, k=1,
+   length=8, label = c("R","SAS") )

> gender <-   gl( n = 2, k = 4,
+   length=8, label=c("f", "m")   )

> q1 <- sample( c(1, 2, 2, 3), 8, replace = TRUE)
> q2 <- sample( c(3, 4, 4, 5), 8, replace = TRUE)
> q3 <- sample( c(1, 2, 2, 3), 8, replace = TRUE)
> q4 <- sample( c(3, 4, 4, 5), 8, replace = TRUE)

> pretest   <- rnorm( n = 8, mean = 70, sd = 5)
> posttest  <- rnorm( n = 8, mean = 80, sd = 5)

> myGenerated <- data.frame(id, gender, workshop,
+   q1, q2, q3, q4, pretest, posttest)

> myGenerated
  id gender workshop q1 q2 q3 q4  pretest posttest
1  1      f        R  1  5  2  3 67.77482 77.95827
2  2      f      SAS  1  4  2  5 58.28944 78.11115
3  3      f        R  2  3  2  4 68.60809 86.64183
4  4      f      SAS  2  5  2  4 64.09098 83.58218
5  5      m        R  1  5  3  4 70.16563 78.83855
6  6      m      SAS  2  3  3  3 65.81141 73.86887
```

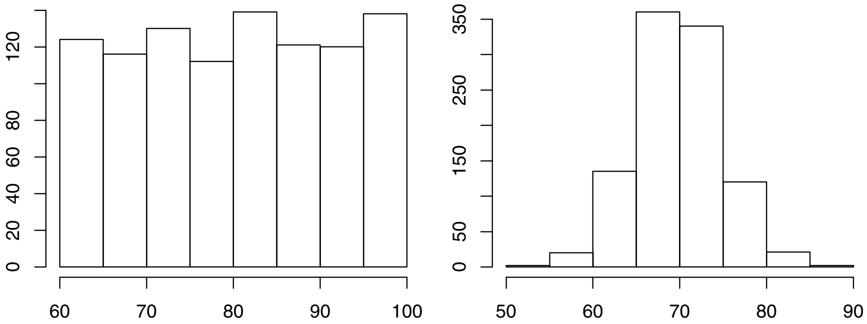


Fig. 12.2. Histograms showing the distributions of our continuous generated data

```
7  7      m        R  2  4  3  4 69.41194 85.23769
8  8      m      SAS  1  4  1  4 66.29239 72.81796
```

12.8 Example Programs for Generating Data

12.8.1 SAS Program for Generating Data

```
* GenerateData.sas ;

DATA myID;
DO id=1 TO 8;
   OUTPUT;
END;
PROC PRINT; RUN;

DATA myWorkshop;
Do i=1 to 4;
DO workshop= 1 to 2;
        OUTPUT;
END;
END;
DROP i;
RUN;
PROC PRINT; RUN;

DATA myGender;
DO i=1 to 2;
   DO j=1 to 4;
      gender=i;
      OUTPUT;
   END;
END;
DROP i j;
RUN;

PROC PRINT; RUN;

DATA myMeasures;
DO i=1 to 8;
   q1=round   ( uniform(1234) * 5 );
   q2=round ( ( uniform(1234) * 5 ) -1 );
   q3=round (uniform(1234) * 5 );
   q4=round (uniform(1234) * 5 );
   test1=normal(1234)*5  + 70;
   test2=normal(1234)*5  + 80;
   OUTPUT;
END;
DROP i;
RUN;

PROC PRINT; RUN;

PROC FORMAT;
VALUES wLabels 1= R  2= SAS ;
VALUES gLabels 1= Female  2= Male ;
RUN;

* Merge and eliminate out of range values;
DATA myGenerated;
MERGE myID myWorkshop myGender myMeasures;
FORMAT workshop wLabels. gender gLabels. ;
ARRAY q{4} q1-q4;
DO i=1 to 4;
        IF q{i} < 1 then q{i}=1;
   ELSE IF q{i} > 5 then q{i}=5;
END;
RUN;

PROC PRINT; RUN;
```

12.8.2 SPSS Program for Generating Data

```
* Filename: GenerateData.sps .

input program.
numeric id(F4.0).
string gender(a1) workshop(a4).
vector q(4).
loop #i = 1 to 8.
compute id = #i.
do if #i <= 5.
+ compute gender =  f .
else.
+ compute gender =  m .
end if.

compute #ws = mod(#i, 2).
if #ws = 0 workshop =  SAS .
if #ws = 1 workshop =  R .

do repeat #j = 1 to 4.
+ compute q(#j)=trunc(rv.uniform(1,5)).
end repeat.

compute pretest = rv.normal(70, 5).
compute posttest = rv.normal(80,5).
end case.
end loop.
end file.
end input program.

list.
```

12.8.3 R Program for Generating Data

```
# Filename: GenerateData.R

# Simple sequences.
1:10
seq(from = 1,to = 10,by = 1)
seq(from = 5,to = 50,by = 5)
seq(5, 50, 5)

# Generating our ID variable
id <- 1:8
id

# gl function Generates Levels.
gl(n = 2, k = 1, length = 8)
gl(n = 2, k = 4, length = 8)

#Adding labels.
workshop <- gl(n = 2, k = 1, length = 8,
  label = c("R", "SAS") )
workshop
gender   <- gl(n = 2, k = 4, length = 8,
  label = c("f", "m") )
gender

# Generating Repetition Patterns (Not Factors)
gender   <- rep(1:2, each = 4, times = 1)
gender
workshop <- rep(1:2, each = 1, times = 4)
```

```
workshop
myZeros  <- rep(0, each = 8)
myZeros

# Generating Values for Reading a Fixed-Width File
myPrefixes <- c("x", "y", "z")
mySuffixes <- rep(1:300, each = 3, times = 1)
head(mySuffixes, n = 20)
myVariableNames <- paste(myPrefixes, mySuffixes, sep = "")
head(myVars, n = 20)
myWidthSet <- c(8, 12, 10)
myVariableWidths <- rep(myWidthSet, each = 1, times = 100)
head(myRecord, n = 20)

myVariablenames  <- c("id", myVariableNames)
myVariableWidths <- c( 5  , myVariableWidths)

mydata <- read.fwf(
   file  = myfile,
   width = myVariableWidths,
   col.names  = myVariableNames,
   row.names  = "id")

# Generating uniformly distributed Likert data
myValues <- c(1, 2, 3, 4, 5)
set.seed(1234)
q1 <- sample( myValues, size = 1000, replace = TRUE)
mean(q1)
sd(q1)

# Generating normally distributed Likert data
myValues <- c(1, 2, 2, 3, 3, 3, 4, 4, 5)
set.seed(1234)
q2 <- sample( myValues , size = 1000, replace = TRUE)
mean(q2)
sd(q2)

# Plot details in Traditional Graphics chapter.
par( mar = c(2, 2, 2, 1)+0.1 )
par( mfrow = c(1, 2) )
barplot( table(q1) )
barplot( table(q2) )
# par( mfrow = c(1, 1) ) #Sets back to 1 plot per page.
# par( mar = c(5, 4, 4, 2) + 0.1 )
# Same two barplots routed to a file.
postscript(file = "F:/r4sas/chapter12/GeneratedIntegers.eps",
           width = 4.0, height = 2.0)
par( mar = c(2, 2, 2, 1) + 0.1 )
par( mfrow = c(1, 2) )
barplot( table(q1) )
barplot( table(q2) )
dev.off()

# Two Likert scales with mean difference
set.seed(1234)
q1 <- sample( c(1, 2, 2, 3), size = 8, replace = TRUE)
mean(q1)

set.seed(1234)
q2 <- sample( c(3, 4, 4, 5), size = 8, replace = TRUE)
mean(q2)

# Generating continuous data

# From uniform distribution.
# mean = 0.5
set.seed(1234)
x1 <- runif(n = 1000)
mean(x1)
sd(x1)

# From a uniform distribution
# between 60 and 100
set.seed(1234)
x2 <- runif(n = 1000, min = 60, max = 100)
mean(x2)
sd(x2)

# From a normal distribution.
set.seed(1234)
x3 <- rnorm(n = 1000)
mean(x3)
sd(x3)

set.seed(1234)
x4 <- rnorm(n = 1000, mean = 70, sd = 5)
mean(x4)
sd(x4)
```

```
# Plot details are in Traditional Graphics chapter.
par( mar = c(2, 2, 2, 1) + 0.1 )
par( mfrow = c(1, 2) )
hist(x2)
hist(x4)
# par( mfrow = c(1, 1) ) #Sets back to 1 plot per page.
# par( mar = c(5, 4, 4, 2) + 0.1 )

# Same pair of plots, this time sent to a file.
postscript(file = "F:/r4sas/chapter12/GeneratedContinuous.eps",
           width = 4.0, height = 2.0)

par( mar = c(2, 2, 2, 1) + 0.1 )
par( mfrow = c(1, 2) )
hist(x2, main = "")
hist(x4, main = "")
dev.off()

# par( mfrow = c(1,1) ) #Sets back to 1 plot per page.
# par( mar = c(5, 4, 4, 2) + 0.1 )

# Generating a Data Frame.
id <- 1:8
workshop <- gl( n = 2, k = 1,
  length = 8, label = c("R","SAS") )
gender <-   gl( n = 2, k = 4,
  length = 8, label = c("f","m")   )
q1 <- sample( c(1, 2, 2, 3), 8, replace = TRUE)
q2 <- sample( c(3, 4, 4, 5), 8, replace = TRUE)
q3 <- sample( c(1, 2, 2, 3), 8, replace = TRUE)
q4 <- sample( c(3, 4, 4, 5), 8, replace = TRUE)
pretest   <- rnorm(n = 8, mean = 70, sd = 5)
posttest  <- rnorm(n = 8, mean = 80, sd = 5)

myGenerated <- data.frame(id, gender, workshop,
  q1, q2, q3, q4, pretest, posttest)
myGenerated
```

13

Managing Your Files and Workspace

When using SAS and SPSS, you manage your files using the same operating system commands that you use for your other software. SAS does have a few file management procedures such as DATASETS and CATALOG, but you can get by just fine without them for most purposes.

R is quite different. It has a set of commands that replicate many operating system functions such as listing names of objects, deleting them, setting search paths, and so on. Learning how to use these commands is especially important because of the way R stores its data. You need to know how to make the most of your computer's memory.

13.1 Loading and Listing Objects

You can see what objects are in your workspace with the `ls` function. To list all objects such as data frames, vectors, and functions, use

```
ls()
```

The `objects` function does the same thing and its name is more descriptive, but `ls` is more widely used since it is the same command that UNIX, Linux, and MacOS X users can use to list the files in a particular directory or folder (without the parentheses).

When you first start R, using the `ls` function will tell you there is nothing in your workspace. How it does this is quite odd by SAS or SPSS standards. It tells you that the list of objects in memory is a character vector with zero values:

```
> ls()

character(0)
```

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_13, © Springer Science+Business Media, LLC 2011

The file *myall.RData* contains all of the objects we created in Chap. 5, "Programming Language Basics." After loading that into our workspace using the load function, ls will show us the objects that are available:

```
> load("myall.RData")

> ls()

[1] "gender"   "mydata"   "mylist"   "mymatrix" "q1"
[6] "q2"       "q3"       "q4"       "workshop"
```

You can use the pattern argument to search for any regular expression. Therefore, to get a list of all objects that begin with the string *"my,"* you can use the following:

```
> ls(pattern = "my")

[1] "mydata"   "mylist"   "mymatrix"
```

The ls function does not look inside data frames to see what they contain, and it does not even tell you when an object is a data frame. You can use many of the functions we have already covered to determine what an object is and what it contains.

To review, typing its name or using the print function will show you the whole object or at least something about it. What print shows you depends on the class of the object. The head and tail functions will show you the top or bottom few lines of vectors, matrices, tables, data frames, or functions.

The class function will tell you if an object is a data frame, list, or some other object. The names function will show you object names within objects such as data frames, lists, vectors, and matrices. The attributes function will display all of the attributes that are stored in an object such as variable names, the object's class, and any labels that it may contain.

```
> attributes(mydata)

$names

[1] "id"   "workshop" "gender"   "q1"   "q2"   "q3"   "q4"

$class

[1] "data.frame"

$row.names

[1] 1 2 3 4 5 6 7 8
```

The `str` function displays the *str*ucture of any R object in a compact form. When applied to data frames, it is the closest thing base R has to SAS's PROC CONTENTS. Equivalent SPSS commands are SYSFILE INFO and, for a file that is already open, DISPLAY DICTIONARY.

```
> str(mydata)

 data.frame :   8 obs. of  6 variables:
 $ workshop: Factor w/ 4 levels "R","SAS","SPSS",..:
      1 2 1 2 1 2 1 2

 $ gender  : Factor w/ 2 levels "f","m": 1 1 1 NA 2 2 2 2

 $ q1      : num  1 2 2 3 4 5 5 4

 $ q2      : num  1 1 2 1 5 4 3 5

 $ q3      : num  5 4 4 NA 2 5 4 5

 $ q4      : num  1 1 3 3 4 5 4 5
```

The `str` function works on functions, too. What follows is the structure it shows for the `lm` function:

```
> str( lm )

function (formula, data, subset, weights, na.action,

method = "qr", model = TRUE, x = FALSE, y = FALSE,

qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, ...)
```

The `ls.str` function applies the `str` function to every object in your workspace. It is essentially a combination of the `ls` function and the `str` function. The following is the structure of all the objects I had in my workspace as I wrote this paragraph.

```
> ls.str()

myCounts :  table  int [1:2, 1:2] 2 2 1 2

myCountsDF :  data.frame :      4 obs. of  3 variables:

 $ gender  : Factor w/ 2 levels "f","m": 1 2 1 2

 $ workshop: Factor w/ 2 levels "R","SAS": 1 1 2 2
 $ Freq    : int  2 2 1 2

mydata :  data.frame :  8 obs. of  6 variables:

 $ workshop: int  1 2 1 2 1 2 1 2

 $ gender  : Factor w/ 2 levels "f","m": 1 1 1 NA 2 2 2 2

 $ q1      : int  1 2 2 3 4 5 5 4

 $ q2      : int  1 1 2 1 5 4 3 5

 $ q3      : int  5 4 4 NA 2 5 4 5

 $ q4      : int  1 1 3 3 4 5 4 5
```

Harrell's `Hmisc` package has a `contents` function that is modeled after the SAS CONTENTS procedure. It also lists names and other attributes as shown below. However, it works only with data frames.

```
> library("Hmisc")

Attaching package: Hmisc ...

> contents(mydata)

Data frame:mydata   8 observations and 7 variables
Maximum # NAs:1

         Levels Storage NAs
id              integer   0
workshop        integer   0
gender        2 integer   1
q1              integer   0
q2              integer   0
q3              integer   1
q4              integer   0

+--------+------+
|Variable|Levels|
+--------+------+
| gender |  f,m |
+--------+------+
```

13.2 Understanding Your Search Path

Once you have data in your workspace, where exactly are they? They are in an *environment* called *.GlobalEnv*. The `search` function will show us where that resides in R's search path. Since the search path is affected by any packages or data files you load, we will start R with a clean workspace and load our practice data frame, mydata:

```
> setwd("c:/myRfolder")

> load("mydata.RData")

> ls()

[1] "mydata"
```

Now let us examine R's search path.

```
> search()

[1] ".GlobalEnv"        "package:stats" "package:graphics"
[4] "package:grDevices" "package:utils" "package:datasets"
[7] "package:methods"   "Autoloads"     "package:base"
```

Since our workspace, .GlobalEnv, is in position 1, R will search it first. By supplying no arguments to the `ls` function, we were asking for a listing of objects in the first position of the search path. Let us see what happens if we apply `ls` to different levels. We can use either the path position value, 1, 2, 3, etc. or their names.

```
> ls(1)  # This uses position number.

[1] "mydata"

> ls(".GlobalEnv")  # This uses name.

[1] "mydata"
```

The *package:stats* at level 2 contains some of R's built-in statistical functions. There are a lot of them, so let us use the `head` function to show us just the top few results:

```
> head( ls(2) )

[1] "acf"        "acf2AR"     "add.scope"  "add1"
[5] "addmargins" "aggregate"
```

```
> head( ls("package:stats") )  # Same result.

[1] "acf"          "acf2AR"       "add.scope"   "add1"
[5] "addmargins"  "aggregate"
```

13.3 Attaching Data Frames

Books on R (including this one) often warn against using the `attach` function, but then they go ahead and use it. They do this because the resulting R code is so much simpler to read when you refer to variables as `q1` rather than `mydata$q1`. However, as you will soon see, `attach` is much trickier to use than you would suspect at first.

Many R experts *never* use the `attach` function. As we discussed in Sect. 7.8, you can avoid using `attach` by instead using `with` or by specifying the `data` argument on modeling functions. Understanding the search path is essential to understanding what the `attach` function really does, and why experts avoid its use. We will attach mydata and see what happens:

```
> attach(mydata)

> search()

 [1] ".GlobalEnv"        "mydata"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"

> ls(2)

[1] "gender"   "id"       "q1"       "q2"       "q3"
[6] "q4"       "workshop"
```

You can see that `attach` has made virtual copies of the variables stored in mydata and placed them in search position 2. When we refer to just "gender" rather than "mydata$gender," R looks for it in position 1 first. It does not find anything with just that short name, even though mydata$gender is in that position. R then goes on to position 2 and finds it. (I am talking about function calls in general, not modeling function calls that specify, e.g., `data = mydata`.) This is the process that makes it so easy to refer to variables by their short names. It also makes them very confusing to work with if you create new variables! Let us say we want to take the square root of q4:

```
> q4 <- sqrt(q4)
> q4

[1] 1.000000 1.000000 1.732051 1.732051 2.000000 2.236068
[7] 2.000000 2.236068
```

This looks like it worked fine. However, let us list the contents of search positions 1 and 2 to see what really happened:

```
> ls(1)

[1] "mydata" "q4"

> ls(2)

[1] "gender"   "id"       "q1"       "q2"       "q3"
[6] "q4"       "workshop"
```

R created the new version of q4 as a separate vector in our main workspace. The copy of q4 that the `attach` function put in position 2 was never changed! Since search position 1 dominates, asking for q4 will cause R to show us the one in our workspace. Asking for mydata$q4 will cause R to go inside the data frame and show us the original, untransformed values.

There are two important lessons to learn from this:

1. If you want to create a new variable inside a data frame, do so using a method that clearly specifies the name of the data frame. One approach is specifying `mydata$varname` or `mydata[ ,"varname"]`. Another approach is to use the `tranform` function described in Sect. 10.1, "Transforming Variables."
2. When two objects have the same name, R will always choose the object higher in the search path. There is one important exception to this: when you specify a data frame using the `data` argument on a modeling function call. That will cause R to go inside the data frame you specify and get any vectors that exist there.

When the `attach` function places objects in position 2 of the search path (a position you can change but rarely need to), those objects will block, or *mask*, any others of the same name in lower positions (i.e., further toward the end of the search list). In the following example, I started with a fresh launch of R, loaded mydata, and attached it twice to see what would happen:

```
> attach(mydata)
> attach(mydata)  # Again, to see what happens.

The following object(s) are masked from mydata (position 3):
       gender id q1 q2 q3 q4 workshop
```

```
> search()

 [1] ".GlobalEnv"    "mydata"             "mydata"
 [4] "package:stats" "package:graphics"  "package:grDevices"
 [7] "package:utils" "package:datasets"  "package:methods"
[10] "Autoloads"     "package:base"
```

Note that mydata is now in search positions: 2 and 3. If you refer to any variable or object, R has to decide which one you mean (they do not need to be identical, as in this example.) The message about masked objects in position 3 tells us that the second `attach` brought variables with those names into position 2. The variables from the first `attach` were then moved to position 3, and those with common names were masked (all of them in this example). Therefore, we can no longer refer to them by their simple names; those names are already in use somewhere higher in the search path. In this case, the variables from the second attach went to position 2, and they will be used. Rarely would anyone actually try to attach a data frame twice. But if you run a program twice in a row that contains an `attach` function call, that is what will happen. However, it is simple to avoid this mess by including the `detach` function in any program that attaches a data frame:

```
attach(mydata)
  [do your work with short variable names]
detach(mydata)
```

When we first learned about vectors, we created q1, q2, q3, and q4 as vectors and then formed them into a data frame. If we had left them as separate vectors in our main workspace, even the first use of the `attach` function would have been in trouble. The vectors in position 1 would have blocked those with the same names in positions 2 and 3.

13.4 Loading Packages

We have seen throughout the book that R packages are loaded from its library using the `library` function. But how are the packages made available to R? The `library` function attaches packages to your search path.

In Sect. 2.2, "Loading an Add-on Package," we saw that when two packages share a function name, the most recent one loaded from the library is the one R will use. That is an example of the same masking effect described in the previous section. This is rarely a source of trouble since you usually want the functions from the latest package you load. However, you can avoid any potential conflicts by detaching a package when you are finished using it. In this example, I am using the `describe` function from the `prettyR` package:

```
> library("prettyR")

> describe(mydata)

Description of mydata

Numeric
          mean    median       var        sd   valid.n
q1        3.25       3.5     2.214     1.488         8
q2        2.75       2.5     3.071     1.753         8
q3       4.143         4     1.143     1.069         7
q4        3.25       3.5       2.5     1.581         8

Factor

workshop
Value                       Count Percent
R                           4       50
SAS                         4       50
mode = >1 mode  Valid n = 8

gender
Value                       Count Percent
m                           4       50
f                           3     37.5
NA                          1     12.5
mode = m  Valid n = 7

> detach("package:prettyR")
```

Next I will get very similar results using the Hmisc package:

```
> library("Hmisc")

Loading required package: survival
Loading required package: splines

Attaching package:  Hmisc

The following object(s) are masked from  package:survival :
    untangle.specials

The following object(s) are masked from  package:base :
    format.pval, round.POSIXt, trunc.POSIXt, units

> describe(mydata)
```

```
mydata

 6  Variables      8  Observations
----------------------------------------------
workshop
      n missing  unique
      8       0       2

R (4, 50%), SAS (4, 50%)
----------------------------------------------
gender
      n missing  unique
      7       1       2

f (3, 43%), m (4, 57%)...

> detach("package:Hmisc")
```

We see R is warning us about masked functions from the packages survival and base but *not* prettyR. Had I not detached prettyR it would have warned that after loading Hmisc from the library, I would no longer have access to the describe function in prettyR.

You can also avoid confusion about attached packages by prefixing each function call with the package's name and two colons:

```
> library("prettyR")
> library("Hmisc")
    ...The following object(s) are masked from  package:prettyR :
       describe...

> prettyR::describe(mydata)
    ---prettyR s describe output would appear---

> Hmisc::describe(mydata)
    ---Hmisc s describe output would appear---
```

13.5 Attaching Files

So far, we have only used the attach function with data frames. It can also be very useful with R data files. If you load a file, it brings all objects into your workspace. However, if you attach the file, you can bring in only what you need and then detach it.

For example, let us create a variable x and then add only the vector q4 from the file myall.RData, a file that contains the objects we created in Chapter5, "Programming Language Basics." Recall that in that chapter, we created each practice variable first as a vector and then converted them to a factor, a matrix, a data frame, and a list.

```
> x <- c(1, 2, 3, 4, 5, 6, 7, 8)

> attach("myall.RData")

> search()

 [1] ".GlobalEnv"        "file:myall.RData"  "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"

> q4 <- q4
```

The last statement looks quite odd! What is going on? The `attach` function loaded myall.RData, but put it at position 2 in the search path. R will place any variables you create in your workspace (position 1), and the attached copy allows R to find q4 in position 2. So it copies it from there to your workspace. Let us look at what we now have in both places:

```
> ls(1)  # Your workspace.

[1] "q4" "x"

> ls(2)  # The attached file.

 [1] "mystats" "gender"   "mydata"   "mylist"   "mymatrix"
 [6] "q1"       "q2"       "q3"       "q4"       "workshop"

> detach(2)
```

So we have succeeded in copying a single vector, q4, from a data frame into our workspace. The final `detach` removes `"file:myall.RData"` from the search path.

13.6 Removing Objects from Your Workspace

To delete an object from your workspace, use the `remove` function or the equivalent `rm` function as in

```
rm(mydata)
```

The rm function is one of the few functions that will accept multiple objects separated by commas; that is, the names do not have to be in a single character vector. In fact, the names *cannot* simply be placed into a single vector. We will soon see why.

Let us load myall.RData, so we will have lots of objects to remove:

```
> load(file = "myall.RData")

> ls()

 [1] "mystats" "gender"   "mydata"   "mylist"   "mymatrix"
 [6] "q1"       "q2"       "q3"       "q4"       "workshop"
```

We do not need our vectors, workshop, gender, and the q variable since they are in our data frame, mydata. To remove these extraneous variables, we can use

```
rm(workshop, gender, q1, q2, q3, q4)
```

If we had lots of variables, manually entering each name would get quite tedious. We can instead use any of the shortcuts for creating sets of variable names described in Chap. 7, "Selecting Variables." Let us use the ls function with its pattern argument to find all of the objects that begin with the letter "q."

```
> myQvars <- ls(pattern = "q")

> myQvars

[1] "q1" "q2" "q3" "q4"
```

Now let us use the c function to combine workshop and gender with myQvars:

```
> myDeleteItems <- c("workshop", "gender", myQvars)

> myDeleteItems

[1] "workshop" "gender"   "q1"       "q2"       "q3"
[6] "q4"
```

Note that myQvars is *not* enclosed in quotes in the first line. It is already a character vector that we are adding to the character values of "workshop" and "gender."

Finally, we can delete them all at once by adding the list argument to the rm function:

```
> rm(list = myDeleteItems)
>
> ls()
[1] "mydata"   "myDeleteItems" "mylist"   "mymatrix"
[5] "myQvars"  "mystats"
```

Finally, we can remove myQvars and myDeleteItems.

```
> rm(myQvars,myDeleteItems)

> ls()

[1] "mydata"   "mylist"   "mymatrix" "mystats"
```

It may appear that a good way to delete the list of objects in myDeleteItems would be to use

```
rm(myDeleteItems)  # Bad!
```

or, equivalently,

```
rm( c("workshop", "gender", "q1", "q2", "q3", "q4") )  # Bad!
```

However, that would delete only the *list of item names*, not the items themselves! That is why the `rm` function needs a `list` argument when dealing with character vectors.

Once you are happy with the objects remaining in your workspace, you can save them all with

```
save.image("myFavorites.RData")
```

If you want to delete all of the visible objects in your workspace, you can do the following. Be careful; there is no "undo" function for this radical step!

```
myDeleteItems <- ls()

rm( list = myDeleteItems )
```

Doing this in two steps makes it clear what is happening, but, of course, you can nest these two functions. This approach looks quite cryptic at first, but I hope the above steps make it much more obvious what is occurring.

```
rm( list = ls() )
```

To conserve workspace by saving only the variables you need within a data frame, see Sect. 10.9, "Keeping and Dropping Variables." The `rm` function cannot drop variables stored within a data frame.

13.7 Minimizing Your Workspace

Removing unneeded objects from your workspace is one important way to save space in your computer's main memory. You can also use the `cleanup.import` function from Harrell's `Hmisc` package. It automatically stores the variables in a data frame in their most compact form. You use it as

```
library("Hmisc")

mydata <- cleanup.import(mydata)
```

If you have not installed `Hmisc`, see Chap. 2, "Installing R and Add-on Packages," for details.

13.8 Setting Your Working Directory

Your working directory is the location R uses to retrieve or store files, if you do not otherwise specify the full path for filenames. In Windows, the default working directory is *My Documents*. In Windows XP or earlier, it is
C:\Documents and Settings\username\My Documents. In Windows Vista or later, it is C:\Users\Yourname\My Documents. In Mac OS, the default working directory is /Users/username.

The `getwd` function will tell you the current location of your working directory:

```
> getwd()

[1] "C:/Users/Muenchen/My Documents"
```

Windows users can see or change their working directory by choosing *File>Change dir*. R will then display a window that you use to browse to any folder you like.

In any operating system you can change the working directory with the `setwd` function. This is the equivalent to SPSS's CD command and somewhat similar to the SAS LIBNAME statement. Simply provide the full path between the quotes:

```
setwd("c:/myRfolder")
```

We discussed earlier that R uses the forward slash "/" even on computers running Windows. That is because within strings, R uses "`\t`," "`\n`," and "`\\`" to represent the single characters tab, newline, and backslash, respectively. In general, a backslash followed by another character may have a special meaning. So when using R in Windows, always specify the paths with either a single forward slash or two backslashes in a row. This book uses the single forward slash because that works with R in all operating systems.

You can set your working directory automatically by putting it in your .Rprofile. For details, see Appendix C, "Automating Your R Setup."

13.9 Saving Your Workspace

Throughout this book we manually save the objects we create, naming them as we do so. That is the way almost all other computer programs work. R also has options for saving your workspace automatically when you exit.

13.9.1 Saving Your Workspace Manually

To save the entire contents of your workspace, you can use the `save.image` function:

```
save.image(file = "myWorkspace.RData")
```

This will save all your objects, data, functions, everything. Therefore, it is usually good to remove unwanted objects first, using the `rm` function. See Sect. 13.6, “Removing Objects from Your Workspace,” for details.

If you are a Windows user, R does not automatically append the .RData extension, as do most Windows programs, so make sure you enter it yourself.

Later, when you start R, you can use *File>Load Workspace* to load it from the hard drive back into the computer’s memory. You can also restore them using the `load` function:

```
load(file = "myWorkspace.RData")
```

If you want to save only a subset of your workspace, the `save` function allows you to list the objects to save, separated by commas, before the file argument:

```
save(mydata, file = "mydata.RData")
```

This is one of the few functions that can accept many objects separated by commas, so it might save three as in the example below:

```
save(mydata, mylist, mymatrix, file = "myExamples.RData")
```

It also has a `list` argument that lets you specify a character vector of objects to save.

You exit R by choosing *File>Exit* or by entering the function call `quit()` or just `q()`. R will then offer to save your workspace. If you have used either the `save` or the `save.image` functions recommended above, you should click “No.”

13.9.2 Saving Your Workspace Automatically

Every time you exit R, it offers to save your workspace for you automatically. If you click “Yes,” it stores it in a file named “.RData” in your working directory (see how to set in in the Sect. 13.8). The next time you start R from the same working directory, it automatically loads that file back into memory, and you can continue working.

While this method saves a little time, it also has problems. The name .RData is an odd choice, because most operating systems hide files that begin with a period. So, initially, you cannot copy or rename your project files! That is true on Windows, Mac OS, and Linux/UNIX systems. Of course, you can tell your operating system to show you such files (shown below).

Since all your projects end up in a file with the same name, it is harder to find the one you need via search engines or backup systems. If you accidentally moved an .RData file to another folder, you would not know which project it contained without first loading it into R.

13.9.3 Getting Operating Systems to Show You .RData Files

While the default workspace file, ".RData," is hidden on most operating system, you can tell them to show you those files.

To get Windows XP to show you .RData, in Windows Explorer *uncheck* the option *Tools> Folder Options.* Then click on the View tab and uncheck the box "Hide extensions to known file types." Then, in the "Folder views" box at the top of the View tab, click "Apply to All Folders." Then click *OK*.

In Windows Vista, use the following selection: *Start>Control Panel> Appearance and Personalization> Folder Options> View> Show hidden files and folders*. Then click *OK*.

In Windows 7 or later, start File Explorer, then follow this menu path, and *uncheck* the option *Organize> Folder and search options> View> Hide extensions for known file types*. Then click *OK*.

Note that this will still not allow you to click on a filename like myProject.RData and rename it to just .RData. The Windows Rename message box will tell you "You must type a filename."

Linux/UNIX users can see files named .RData with the command "ls -a."

Macintosh users can see files named .RData by starting a terminal window with *Applications> Utilities> Terminal.* In the terminal window, enter

```
defaults write com.apple.finder AppleShowAllFiles TRUE
killall Finder
```

To revert back to normal file view, simply type the same thing, but with "FALSE" instead of "TRUE."

13.9.4 Organizing Projects with Windows Shortcuts

If you are a Windows user and like using shortcuts, there is another way to keep your various projects organized. You can create an R shortcut for each of your analysis projects. Then you right-click the shortcut, choose *Properties*, and set the *Start in folder* to a unique folder. When you use that shortcut to start R, on exit it will store the .RData file for that project. Although neatly organized into separate folders, each project workspace will still be in a file named .RData.

13.10 Saving Your Programs and Output

R users who prefer the graphical user interface can easily save programs, called scripts, and output to files in the usual way. Just click anywhere on the window you wish to save, choose *File>Save as*, and supply a name. The standard extension for R programs is ".R" and for output it is simply ".txt". You can also save bits of output to your word processor using the typical cut-and-paste steps.

In Windows, R will not automatically append ".R" to each filename. You must specify that yourself. When you forget this, and you will, later choosing *File>Open script* will not let you see the file! You will have to specify "*.*" as the filename to get the file to appear.

R users who prefer to use the command-line interface often use text editors such as Emacs, or the one in JGR, that will check their R syntax for errors. Those files are no different from any other file created in a given editor.

Windows and Macintosh users can cut and paste graphics output into their word processors or other applications. Users of any operating system can rerun graphs, directing their output to a file. See Chap. 14, "Graphics Overview," for details.

13.11 Saving Your History

R has a *history* that saves all of the commands in a given session. This is just like the SPSS *journal*. The closest thing SAS has is its *log*, but that contains messages, which are absent in R's history file. If you run your programs from R's program editor or from a text editor, you already know how to save your programs; you are unlikely to need to save your command history separately.

However, if you need to, you can save the current session's history to a file in your working directory with the `savehistory` function. To route the history to a different folder, use the `setwd` function to change it before using `savehistory`, or simply specify the file's full path in the `file` argument:

```
savehistory(file = "myHistory.Rhistory")
```

You can later recall it using the `loadhistory` function.

```
loadhistory(file = "myHistory.Rhistory")
```

Note that the filename can be anything you like, but the extension should be ".Rhistory." In fact the entire filename will be simply ".Rhistory" if you do not provide one. You can also automate loading and saving your history. For details, see Appendix C, "Automating Your R Setup."

All of the file and workspace functions we have discussed are summarized in Table 13.1.

Table 13.1. Workspace management functions

Function to perform	Example
List object names, including .First, .Last	`objects(all.names=TRUE)`
List object names, of most objects	`ls()` or `objects()`
List object attributes	`attributes(mydata)`
Load workspace	`load(file = "myWorkspace.RData")`
Remove a variable from a data frame	`mydata$myvar <- NULL`
Remove all objects (nonhidden ones)	`rm( list=ls() )`
Remove an object	`rm(mydata)`
Remove several objects	`rm(mydata, mymatrix, mylist)`
Save all objects	`save.image(file = "myWorkspace.RData")`
Save some objects	`save(x, y, z, file = "myObjects.RData)`
Show structure of all objects	`ls.str( all.names=TRUE )`
Show structure of most objects	`ls.str()`
Show structure of data frame only (requires `Hmisc`)	`contents(mydata)`
Show structure of objects by name	`str(mydata), str(lm)`
Store data efficiently (requires `Hmisc`)	`mydata <- cleanup.import(mydata)`
Working directory, getting	`getwd ()`
Working directory, setting	`setwd("/mypath/myfolder")` Even Windows uses forward slashes!

13.12 Large Data Set Considerations

All of the topics we have covered in this chapter are helpful for managing the amount of free space you have available in your workspace. This is a very important topic to R users.

For most procedures, SAS and SPSS store their data on your hard drive and pull "chunks" of data into memory to work on as needed. That allows them to analyze as much data as you can afford to store. Of course, even in SAS and SPSS, some kinds of models must have all their data in memory, but these are relatively few in number.

Unfortunately, R must store all the data in your computer's memory at once, so it cannot analyze what is commonly called *big data*. Much work is going into ways to get around the memory limitation. For example, Lumley's `biglm` package [39] processes data in "chunks" for some linear and generalized linear models. There are other packages that provide various ways to get around the memory limitation for certain kinds of problems. They are listed in the High Performance Computing Task View at CRAN.

Given the low cost of memory today, R's memory limitation is less of a problem than you might think. R can handle hundreds of thousands of records on a computer with 2 gigabytes of memory available to R. That is the memory limit for a single process or program in 32-bit operating systems. Operating systems with 64-bit memory spaces are now the norm, allowing you to analyze millions of records.

Another way around the limitation is to store your data in a relational database and use its facilities to generate a sample to analyze. A sample size of a few thousand is sufficient for many analyses. However, if you need to ensure that certain small groups (e.g., those who have a rare disease, the small proportion of borrowers who defaulted on a loan), then you may end up taking a complex sample, which complicates your analysis considerably. R has specialized packages to help analyze such samples, including `pps`, `sampfling`, `sampling`, `spsurvey`, and `survey`. See CRAN at `http://cran.r-project.org/` for details.

Another alternative is to purchase a commercial version of R from Revolution Analytics called *Revolution R Enterprise*. It solves the problem in a way similar to that used in packages such as SAS and SPSS. It can handle terabyte-sized files for certain types of problems. You can read more about it at `www.revolutionanalytics.com`.

13.13 Example R Program for Managing Files and Workspace

Most chapters in this book end with the SAS, SPSS, and R programs that summarize the topics in the chapter. However, this chapter has been very specific to R. Therefore, we present only the R program below.

```
# Filename: ManagingWorkspace.R

ls()

setwd("c:/myRfolder")
load("myall.RData")
ls()

# List objects that begin with "my".
ls(pattern = "my")

# Get attributes and structure of mydata.
attributes(mydata)
str(mydata)

# Get structure of the lm function.
str(lm)

# List all objects  structure.
ls.str()

# Use the Hmisc contents function.
install.packages("Hmisc")
library("Hmisc")
contents(mydata)

# ---Understanding Search Paths---

setwd("c:/myRfolder")
load("mydata.RData")
ls()
search()
ls(1)  # This uses position number.
ls(".GlobalEnv")  # This does the same using name.

head( ls(2) )
head( ls("package:stats") )  # Same result.

# See how attaching mydata change the path.
attach(mydata)
search()
ls(2)

# Create a new variable.
q4 <- sqrt(q4)
```

```
q4
ls(1)
ls(2)

# Attaching data frames.
detach(mydata)
attach(mydata)
attach(mydata)
search()

# Clean up for next example,
# or restart R with an empty workspace.
detach(mydata)
detach(mydata)
rm( list = ls() )

# Loading Packages
library("prettyR")
describe(mydata)
detach("package:prettyR")

library("Hmisc")
describe(mydata)
detach("package:Hmisc")

library("prettyR")
library("Hmisc")
prettyR::describe(mydata)
Hmisc::describe(mydata)

# Attaching files.
x <- c(1, 2, 3, 4, 5, 6, 7, 8)
attach("myall.RData")
search()
q4 <- q4

ls(1)  # Your workspace.
ls(2)  # The attached file.
detach(2)

# Removing objects.
rm(mydata)
load(file = "myall.RData")
ls()
# Example not run:
```

```
# rm(workshop, gender, q1, q2, q3, q4)
myQvars <- ls(pattern = "q")
myQvars

myDeleteItems <- c("workshop","gender",myQvars)
myDeleteItems

myDeleteItems
rm( list = myDeleteItems )

ls()
rm( myQvars, myDeleteItems )
ls()

# Wrong!
rm(myDeleteItems)
rm( c("workshop", "gender", "q1", "q2", "q3", "q4") )

save.image("myFavorites.RData")

# Removing all workspace items.
# The clear approach:

myDeleteItems <- ls()
myDeleteItems
rm(list = myDeleteItems)

# The usual approach:
rm( list = ls() )

# Setting your working directory.

getwd()
setwd("c:/myRfolder")

# Saving your workspace.

load(file = "myall.RData")

# Save everything.
save.image(file = "myPractice1.RData")

# Save some objects.
save(mydata,file = "myPractice2.RData")
save(mydata, mylist, mymatrix, file = "myPractice3.RData")
```

```
# Remove all objects and reload myPractice3.
rm( list = ls() )
load("myPractice3.RData")
ls()

# Save and load history.
savehistory(file = "myHistory.Rhistory")
loadhistory(file = "myHistory.Rhistory")
```

14

Graphics Overview

Graphics is perhaps the most difficult topic to compare across SAS, SPSS, and R. Each package contains at least two graphical approaches, each with dozens of options and each with entire books devoted to them. Therefore, we will focus on only two main approaches in R, and we will discuss many more examples in R than in SAS or SPSS. This chapter focuses on a broad, high-level comparison of the three. The next chapter focuses on R's traditional graphics. The one after that focuses just on the grammar of graphics approaches used in both R and SPSS.

14.1 Dynamic Visualization

Dynamic visualization allows you explore your data by interacting with plots. You can often rotate a three-dimensional scatter plot by dragging it about with your mouse. Selections you make in one graph, such as the females in a bar chart, are reflected in all graphs, allowing you to explore your data very quickly. SAS/IML Studio provides this capability in SAS. SPSS does not offer much interactivity. Although dynamic visualization is outside our scope, R has this capability through two excellent packages that we will review briefly.

The `iplots` package, by Urbanek, et al. [63], offers a wide array of interactive plots. These include histograms, bar charts, scatter plots, box plots, fluctuation diagrams, parallel coordinates plots, and spine plots. They all include interactive abilities such as linked highlighting and color brushing. `iplots` is a stand-alone package that you can also activate from within the `Deducer` GUI discussed in Sec. 3.11.1.

R can also link to a separate interactive graphics package, GGobi by Swayne, et al. [54], available for free at http://www.ggobi.org/. GGobi's plots include scatter plots, bar charts, and parallel coordinates plots. One of its most interesting features is called a *tour*. This approach displays a three-dimensional scatter plot, which it rotates as it searches patterns in the data. Each dimension is a weighted combination of other variables that lets you see

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_14, © Springer Science+Business Media, LLC 2011

into a higher dimensional space. Tours are also available in R via the tourr package by Wickham [74].

You can link to GGobi easily from Williams' rattle GUI for data mining, discussed in Sect. 3.11.3. You can also link to GGobi from within R using the function calls provided by the rggobi package by Temple Lang, et al. [60].

14.2 SAS/GRAPH

When you purchase Base SAS, its graphics procedures such as CHART and PLOT use only primitive printer characters. For example, lines are drawn using series of "-" and "|" characters, while plotting symbols are things like "*" or "+." You have to purchase a separate add-on package, SAS/GRAPH, to get real graphics.

From its release in 1980 through 2008, SAS/GRAPH offered a limited set of popular graphics displays. It used short commands, so getting a basic plot was easy. However, its default settings were poorly chosen and it took many additional commands to create a publication-quality graph. If you needed the same set of graphs in a periodic report, that was not too much of a problem. But if you tended to do frequent unique graphs, it was frustrating. If you needed a wider selection of graph styles, you had to switch to another package.

The release of SAS/GRAPH version 9.2 in 2008 finally corrected this problem. The SGPLOT, SGPANEL, and SGSCATTER procedures added modern plots with very attractive default settings. In addition, ODS Graphics gave SAS the ability to automatically provide a standard set of graphics with many statistical analyses. This makes sense as people typically do the same core set of graphs for a given analysis. As nice as these improvements are, they still leave SAS behind both SPSS and R in graphics flexibility.

14.3 SPSS Graphics

SPSS Base includes three types of graphics: one based on Graphics Production Language (GPL) and the two "legacy" systems of standard and interactive graphics. The standard legacy graphics use short commands and reasonable default settings. They can also plot data stored in either wide or long formats without having to restructure the data, which is very convenient. The interactive legacy graphics offer a very limited range of interactivity. For example, selections you make in one graph are not displayed automatically in others, as would happen in SAS/IML Studio, iplots, and GGobi. As you can tell by the "legacy" label that the company applied to them in version 15, SPSS is likely to phase them out eventually.

SPSS's third graphics approach is its GPL. We will discuss this extremely flexible approach below in Sect. 14.5, "The Grammar of Graphics." SPSS's standard legacy commands typically take one statement per graph, while GPL takes over a dozen! That demonstrates the classic tradeoff of simplicity versus power. The example SPSS programs in Chap. 15 use standard legacy graphics, while those in Chap. 16 use GPL.

14.4 R Graphics

R offers three main types of graphics: traditional graphics (also called "base graphics"), the `lattice` package [50], and `ggplot2` package [71].

R's traditional graphics functions come with the main R installation in the `graphics` package. That package is loaded from your library automatically each time you start R. Traditional graphics include high-level functions such as bar plots, histograms, and scatter plots. These functions are brief and the default settings are good.

While SAS and SPSS have you specify your complete plot in advance, R's traditional graphics allow you to plot repeatedly on the same graph. For example, you might start with a scatter plot, and then add a regression line. As you add new features, each writes on top of the previous ones, so the order of commands is important. Once written, a particular feature cannot be changed without starting the plot over from the beginning.

R's traditional graphics also includes low-level functions for drawing things like points, lines, and axes. These provide flexibility and control that people can use to invent new types of graphs. Their use has resulted in many add-on graphics packages for R. The level of control is so fine that you can even use it for artistic graphics that have nothing to do with data. See Murrell's *R Graphics* [42] for examples.

The most important function in R's traditional graphics is the `plot` function. It is a generic function that offers many different kinds of plots for different objects. Various packages often extend the `plot` function's capabilities for any new types of objects they introduce. So although the other graphics packages that R offers have their own advantages, traditional graphics still play an important role.

The main weakness of traditional graphics is that it requires a great deal of effort to repeat plots for groups in your data.

The second major graphics package added to R was `lattice`, written by Sarkar [50]. It implements Cleveland's Trellis graphics system [13]. It is part of the main R distribution, and it does an excellent job of repeating plots for different groups. A good book on that package is Sarkar's *Lattice: Multivariate Data Visualization with R* [49]. We will not examine `lattice` graphics, as the next system covers most of its abilities and offers additional advantages.

The third major package, `ggplot2` [71], written by Wickham, is based on the grammar of graphics described in the next section. It offers an excellent balance between power and ease of use. The `ggplot2` package is based on the same concepts as SPSS's GPL. While the `ggplot2` package offers full flexibility

Table 14.1. Comparison of R's three main graphics packages

	Traditional (or base)	`lattice`	`ggplot2`
Automatic output for different objects	Yes	No	No
Automatic legends	No	Sometimes	Yes
Easily repeats plots for different groups	No	Yes	Yes
Easy to use with multiple data sources	Yes	No	Yes
Allows you to build plots piece by piece	Yes	No	Yes
Allows you to replace pieces after creation	No	No	Yes
Consistent functions	No	No	Yes
Attractiveness of default settings	Good	Good	Excellent
Can do mosaic plots	Yes	Yes	No
Control extends beyond data graphics	Yes	No	No
Underlying graphics system	Traditional	Grid	Grid

that can require many statements, unlike SPSS's GPL, it offers ways to do most graphs in only a few commands.

We will cover `ggplot2` in Chap. 16. Although `ggplot` has advantages over `lattice`, most of our `ggplot2` examples can be created as well in the `lattice` package. A comparison of these three main packages is given in Table 14.1.

14.5 The Grammar of Graphics

Wilkinson's watershed work, *The Grammar of Graphics* [76], forms the foundation of for both SPSS GPL, and Wickham's `ggplot2` package for R. Wilkinson's key insight was the realization that general principles form the foundation of all data graphics. Once you design a graphics language to follow those principles, the language should then be able to do any existing data graphics as well as variations that people had not previously considered.

An example is a stacked bar chart that shows how many students took each workshop (i.e., one divided bar). This not a popular graph, but if you change its coordinate system from rectangular to circular (Cartesian to polar), it becomes a pie chart. So rather than requiring separate procedures, you can have one that includes changing coordinates. That type of generalization applies to various other graphical elements as well.

A much more interesting example is Charles Joseph Minard's famous plot of Napoleon's march to Moscow and back in 1812 (Fig. 14.1). The grey line shows the army's advance toward Moscow, and the dark grey line shows its retreat. The thickness of the line reflects the size of the army. Through a very substantial effort, you can get other graphics packages to create this graph, including R's traditional graphics. However, SPSS's GPL and R's `ggplot2` package can do it easily using the same general concepts that they use for any other plots.[1]

It might seem that people are unlikely to use this type of plot again, but minor variations of it are used to diagram computer network throughput, the transmission of disease, and the communication of product awareness in marketing. The point is that `ggplot2` gives you the broadest range of graphical options that R offers.

14.6 Other Graphics Packages

There are many other graphics packages that we will not have space to examine. You can see a comprehensive list of them at `http://cran.r-project.org/web/views/Graphics.html` A notable one is the `vcd` package [40] for *v*isualizing *c*ategorical *d*ata. The latter was written by Meyer, et al. and was inspired by Friendly's book *Visualizing Categorical Data* [22]. That book describes how to do its plots using SAS macros. The fact that plots initially designed for SAS could be added to R is testament to R's graphical flexibility.

14.7 Graphics Archives

There are comprehensive collections of example graphs and the R programs to make them at the R Graph Gallery `http://addictedtor.free.fr/graphiques/` and the Image Browser at `http://bg9.imslab.co.jp/Rhelp/`. Wickham's `ggplot2` Web site is also filled with many wonderful examples. It is at `http://had.co.nz/ggplot2/`.

14.8 Graphics Demonstrations

You can also use R's `demo` function to have it show you a sample of what it can do. If you provide it no arguments, `demo` will list all available demonstrations for packages that you currently have loaded from your library. Here I load the `lattice` package so you can see how it displays its demos along with those from the traditional `graphics` package:

[1] The code and data to reproduce this plot and several variations are available at `http://r4stats.com`.

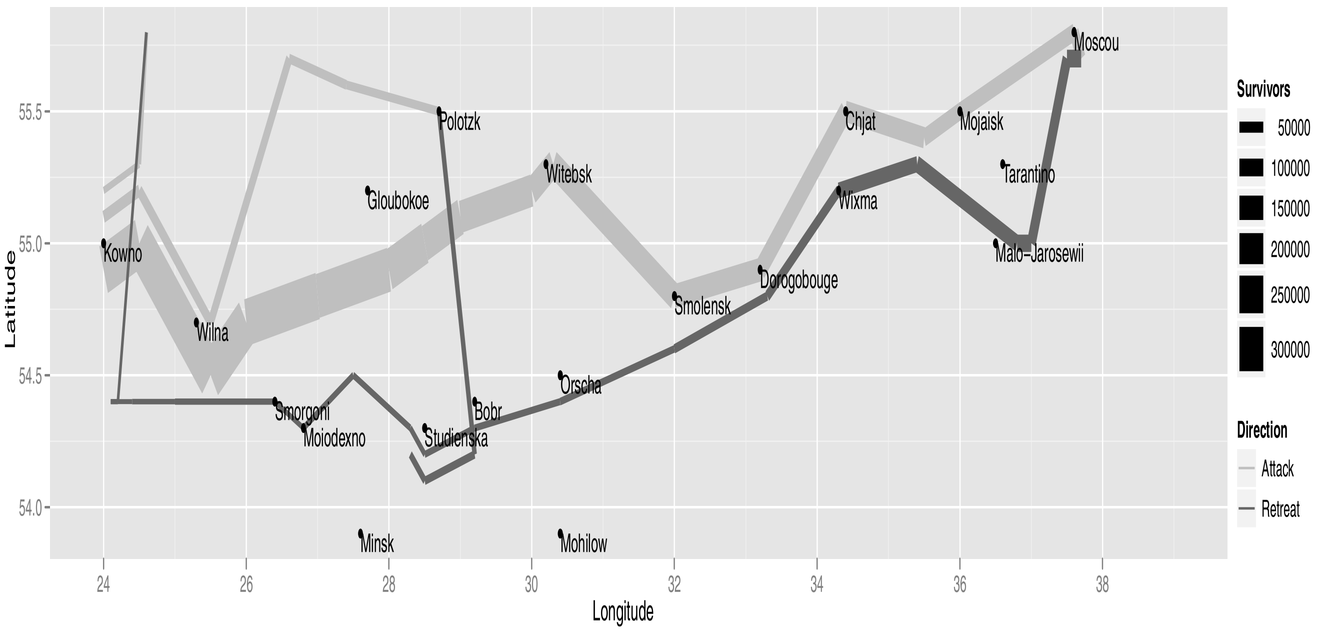


Fig. 14.1. Minard's plot of Napoleon's march created with the `ggplot2` package

```
> library("lattice")

> demo()

...
Demos in package  graphics :

Hershey           Tables of the characters in the Hershey vector
                  fonts
Japanese          Tables of the Japanese characters in the
                  Hershey vector fonts
graphics          A show of some of R s graphics capabilities
image             The image-like graphics builtins of R
persp             Extended persp() examples
plotmath          Examples of the use of mathematics annotation

Demos in package  lattice :

intervals         Confidence intervals
labels            Labels in lattice displays
lattice           Lattice Demos
panel             Custom panel functions
...
```

The `ggplot` package does not currently offer demonstration programs.

To run any of the demos, enter the `demo` function calls below in the R console, and each will generate several example plots.

```
demo("graphics")

demo("persp")

demo("lattice")
```

14.9 Graphics Procedures and Graphics Systems

In SAS/GRAPH and SPSS Base, there are high-level graphics procedures such as PROC GPLOT in SAS or GRAPH /SCATTER in SPSS. R has many add-on packages that contain similar high-level graphics functions.

At a lower level are graphics systems. This is where a command to do something like a scatter plot is turned into the precise combinations of lines and points needed. Controlling the graphics system allows you to change settings that affect all graphs, like text fonts, fill patterns, and line or point types. In SAS, you control these settings using the GOPTIONS command. In SPSS, you select the menu options *File> Options> Edit> Charts*. SPSS also allows you to control these settings when you create a template to apply to future graphs.

Graphics systems are more visible in R because there are two of them. The traditional graphics system controls the high-level traditional graphics functions such as `plot`, `barplot`, and `pie` as well as some add-on packages such as `maps`. It also controls low-level traditional graphics functions that do things like draw lines or rectangles.

The packages `lattice` and `ggplot2` instead use the `grid` graphics system. The implications of this will become much more obvious as we look at examples. The `grid` graphics system was written by Paul Murrell and is documented in his book *R Graphics* [42]. That book is a good reference for both graphics systems as well as high-level traditional graphics procedures. It also gives an introduction to `lattice` graphics.

14.10 Graphics Devices

At the lowest level of graphics control is the graphics device itself. Regardless of the graphics package or system in R you use, the way you see or save the result is the same: the graphics device.

SAS/GRAPH chooses its devices with the GOPTIONS statement. For example, to create a postscript file you might use

```
GOPTIONS DEV = PS GSFNAME = myfile GSFMODE = Replace;
```

In SPSS I usually save graphics for various devices via its *File> Export* menu. You can also use OUTPUT EXPORT or OMS to save graphics output to files.

R has both menus and functions you can use to route graphics to various locations, such as your monitor (the default) or to a file. By default, R writes to your monitor, each plot taking the place of the previous one. Windows users can record the plots for viewing later with the Page Up/Page Down keys, by choosing *History> Recording* in your plot window.

On Windows or Macintosh computers, R lets you simply copy and paste the image into your word processor. That is often an easy way to work because you can resize the image in your word processor, and you can place multiple images on a page by placing them in the cells of a word processing table. However, if you need to align the axes of multiple graphs, it may be better to create a single multiframe plot in R. Chapters 15 and 16 both show how to do that using traditional graphics and the `ggplot2` package, respectively.

When using the cut/paste approach, it is usually best to avoid *File> Copy to the clipboard> Copy as a Bitmap* since that will result in a lower-resolution image. It matches the resolution of your screen. Using *Copy as Metafile* instead will result in a high-resolution version.

You can also use menus to route what you see on your monitor to a file using the usual *File> Save as* selection. However, it does not offer as much control as using a device function.

R has functions named for the various graphics devices. So to write a plot to an encapsulated postscript file, you could use

```
postscript(
  file   = "myPlot.eps",
  paper  = "special",
  width  = 4,
  height = 3.5)          # Opens the device.

plot(pretest, posttest)  # Does the plot.

dev.off()                # Closes the device.
```

The `paper = "special"` argument lets you set the size of the plot using the `width` and `height` arguments. Measurements are in inches.
The last command, `dev.off()`, is optional, since R will close the device when you exit R. However, any additional plots you create will go to that file until you either call the `dev.off()` function or explicitly choose to use the screen device again.

Although bitmap file formats are generally lower-resolution than postscript or PDF, in some cases you may need one. For example, this book was written in a version of the LaTeX document preparation system that prefers encapsulated postscript files. Although it is a high-resolution format, it does not support transparency used by some plots. So I switched to a high-resolution Portable Network Graphics (PNG) file for the plots that used transparency. What follows is the code that I used:

```
png(
  file   = "transparencyDemo.png",
  res    = 600,
  width  = 2400,
  height = 2100)

plot(pretest, posttest)

dev.off()
```

The `png` function measures width and height in dots per inch (dpi) of resolution. So at 600 dpi, 2400 × 2100 yields a 4 × 3.5 inch plot.

Screen device functions are `windows()`, `quartz()`, and `x11()` for Microsoft Windows, Macintosh OS X, and UNIX or Linux, respectively. A helpful command for Windows users is:

```
windows(record = TRUE)
```

which tells Windows to start recording your graphs so that you can use the Page Up key to review older graphs as your work progresses. You can also activate that by displaying one plot and then on the plot window choosing *History> Recording.*

Other popular device functions include `jpeg`, `pdf`, `pictex` (for LaTeX), `png`, and `win.metafile` for Microsoft Windows. The JPG file format is popular for photographs but is best avoided for most data graphics as its compression blurs the sharp boundaries that make up lines, points, and numbers.

Multiple plots can go to the file in the above example because Postscript (and PDF) supports multiple pages. Other formats like Microsoft Windows Metafile do not.

When you send multiple plots to separate files use the following form. The part of the filename "`%03d`" tells R to append 1, 2, etc. to the end of each filename. The numbers are padded by blanks or zeros depending on the operating system:

```
win.metafile(file = "myPlot%03d.wmf")

boxplot( table(workshop) )  # 1st plot goes to myPlot  1.wmf

hist(posttest)              # 2nd plot goes to myPlot  2.wmf

plot(pretest,posttest)      # 3rd plot goes to myPlot  3.wmf

dev.off()
```

The `ggplot2` package has its own `ggsave` function to save its plots to files. You include a call to `ggsave` as the first line after your plot. It saves the type you need based on the file's extension. For details, see Chap. 16, "Graphics with `ggplot2` (GPL)."

```
ggsave(file = "myplot.pdf", width = 4, height = 3.5)
```

For much greater depth on these and many other graphics subjects, see *R Graphics* [42].

15

Traditional Graphics

In the previous chapter, we discussed the various graphics packages in R, SAS, and SPSS. Now we will delve into R's traditional, or base, graphics. Many of these examples will use the practice data set mydata100, which is described in Sect. 1.7

15.1 The plot Function

As we have seen, generic R functions examine the classes of the objects you give them and try to do "the right thing" for you. The `plot` function is generic and it provides a helpful array of plots with very little effort. In Fig. 15.1 The function call used to create each plot is displayed in the title of each plot. For the remainder of the chapter we will examine variations of these and other plots in great detail.

The `methods` function will show you the wide range of methods that `plot` offers for various objects:

```
> methods(plot)

 [1] plot.acf*           plot.data.frame*
 [3] plot.decomposed.ts* plot.default
 [5] plot.dendrogram*    plot.density
 [7] plot.ecdf           plot.factor*
 [9] plot.formula*       plot.hclust*
[11] plot.histogram*     plot.HoltWinters*
[13] plot.isoreg*        plot.lm
[15] plot.medpolish*     plot.mlm
[17] plot.path_curve*    plot.path_index*
[19] plot.ppr*           plot.prcomp*
[21] plot.princomp*      plot.profile.nls*
[23] plot.shingle*       plot.spec
```

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_15, © Springer Science+Business Media, LLC 2011

```
[25] plot.spec.coherency plot.spec.phase
[27] plot.stepfun        plot.stl*
[29] plot.table*         plot.trellis*
[31] plot.ts             plot.tskernel*
[33] plot.TukeyHSD
```

Each time you load another package from your library, this list may grow.

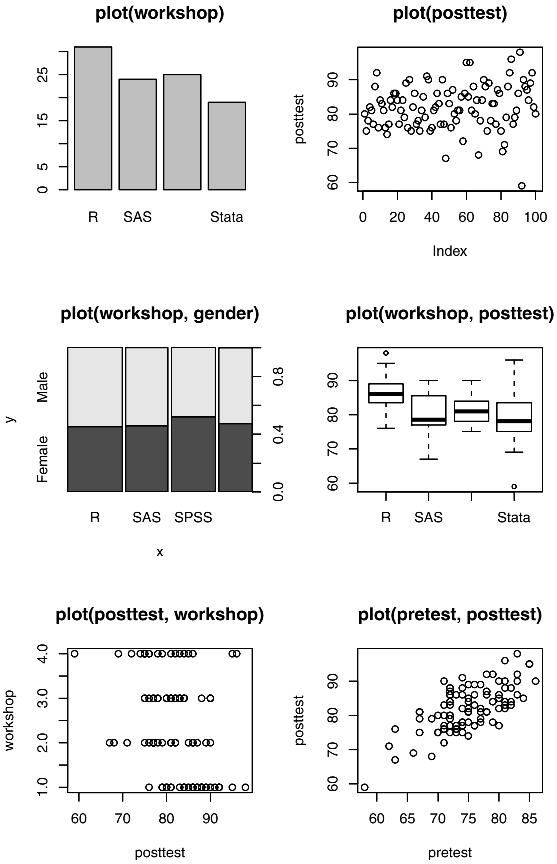


Fig. 15.1. Various plots using the `plot` function. The command that created each plot is listed above each

15.2 Bar Plots

We have seen the `plot` function create bar plots, but to create the full range of bar plots requires the `barplot` function. However, approach `barplot` uses is quite different from the approach used by SAS and SPSS (or by `plot`). While SAS and SPSS assume your data need summarizing and require options to tell them when the data are presummarized, `barplot` assumes just the opposite. While this creates extra work, it also provides additional control. We will first look at bar plots of counts, then grouped counts, and finally various types of bar plots for means.

15.2.1 Bar Plots of Counts

The `bar plot` function call below makes a chart with just two bars, one for each value (Fig. 15.2). They could be counts, means, or any other measure. The main point to see here is that we get a bar for every observation. We are ignoring options for the moment, so the plot lacks labels and its axis values are tiny:

```
barplot( c(40, 60) )
```

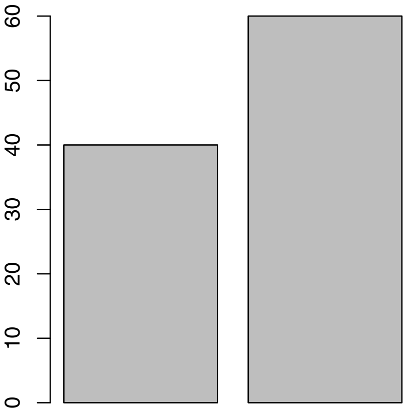


Fig. 15.2. Unlabeled bar plot using traditional graphics

If we apply the same command to variable q4, we get a bar representing each observation in the data frame (Fig. 15.3):

```
barplot(q4)  # Not good.
```

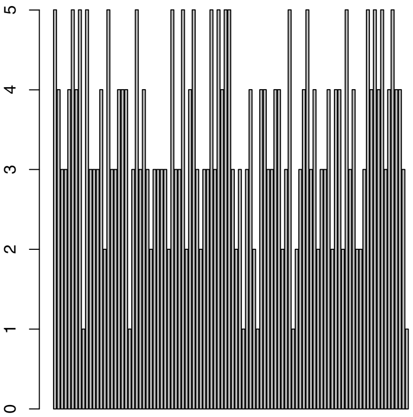


Fig. 15.3. Bar plot on unsummarized variable q4

Notice that the y-axis is labeled 1 through 5. It displays the raw values for every observation rather than summarizing them. That is not a very helpful plot!

Recall that the `table` function gets frequency counts:

```
> table(q4)

q4

 1  2  3  4  5
 6 14 34 26 20
```

Since the `table` function gets the data in the form we need for a bar plot, we can simply nest one function within the other to finally get a reasonable plot (Fig. 15.4).

```
> barplot( table(q4) )
```

When we make that same mistake with a bar plot on gender, we see a different message:

```
> barplot(gender)

Error in bar plot.default(gender) :
  height  must be a vector or a matrix
```

If gender had been coded 1 or 2, and was not stored as a factor, it would have created one bar for every subject, each with a height of 1 or 2. However, because gender is a factor, the message is telling us that `bar plot` accepts

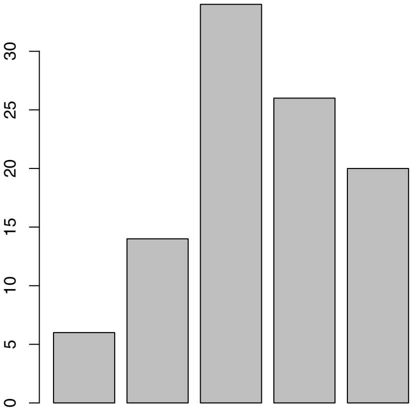


Fig. 15.4. A more reasonable bar plot for variable q4, this time summarized first

only a vector or matrix. The solution is the same as before: count the genders before plotting (Fig. 15.5):

```
barplot( table(gender) )
```

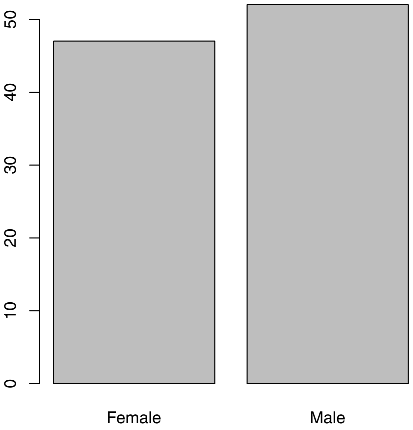


Fig. 15.5. Bar plot of gender

We can use the `horiz = TRUE` argument to turn the graph sideways (Fig. 15.6):

```
> barplot(table(workshop), horiz = TRUE)
```

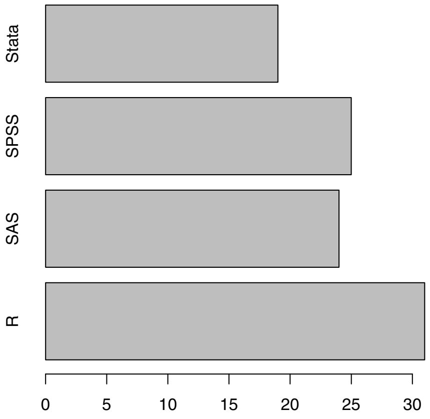


Fig. 15.6. Horizontal bar plot of workshop flipped using `horiz = TRUE`

If we are interested in viewing groups as a proportion of the total, we can stack the bars into a single one (Fig. 15.7). As we will see in the next chapter, this is essentially a pie chart in rectangular Cartesian coordinates, rather than circular polar coordinates. To do this we use the `as.matrix` function to convert the table into the form we need and use the `beside = FALSE` argument to prevent the bars from appearing beside each other.

```
barplot( as.matrix( table(workshop) ),
  beside = FALSE)
```

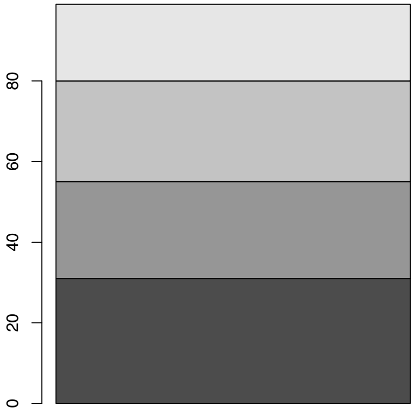


Fig. 15.7. Stacked bar plot of workshop (unlabeled)

15.2.2 Bar Plots for Subgroups of Counts

Recall that the `table` function can handle two variables. Nested within the `bar plot` function, this results in a clustered bar chart (Fig. 15.8):

```
> barplot( table(gender, workshop) )
```

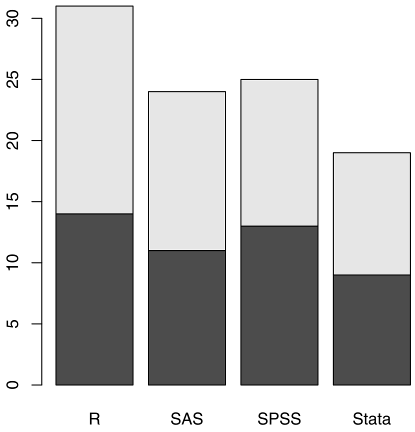


Fig. 15.8. Stacked bar plot of workshop with bars split by gender (lacking legend)

The `plot` function is generic, so it does something different depending on the variables you give it. If we give it two factors, it will create a plot similar to the one above (Fig. 15.9).

```
> plot(workshop, gender)
```

The difference is that the bars fill the plot vertically so the shading gives us proportions instead of counts. Also, the width of each bar varies, reflecting the marginal proportion of observations in each workshop. This is called a *spine plot*. Notice that we did not need to summarize the data with the `table` function. The `plot` function takes care of that for us.

The `mosaicplot` function does something similar (Fig. 15.10).

```
> mosaicplot( table(workshop, gender) )
```

Note that the `mosaicplot` function adds titles to its figures by default. You can suppress them by supplying your own blank main title (Section 15.3).

The `mosaicplot` function can handle the complexity of a third factor. We do not have one, so let us use an example from the `mosaicplot` help file:

```
> mosaicplot( ~ Sex + Age + Survived,
+   data = Titanic, color = TRUE)
```

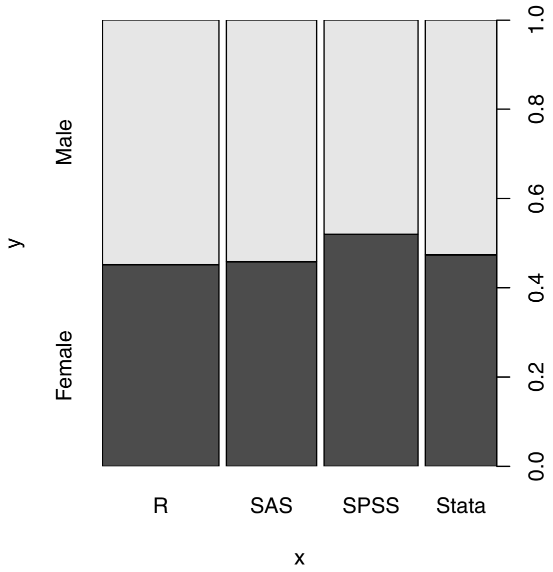


Fig. 15.9. Mosaic plot of workshop by gender, done using `plot` function. Gender is labeled automatically

In Fig. 15.11, we see that not only are the marginal proportions of sex and age reflected, but the third variable of survival is reflected as well. It is essentially four bar charts within a 2 × 2 cross-tabulation.

15.2.3 Bar Plots of Means

The `table` function counted for us. Now let us use the `mean` function, along with the `tapply` function, to get a similar table of means. To make it easier to read, we will store the table of means in myMeans and then plot it (Fig. 15.12):

```
> myMeans <- tapply(q1, gender, mean, na.rm = TRUE)

> barplot(myMeans)
```

Adding workshop to the `tapply` function is easy, but you must combine gender and workshop into a list first, using the `list` function (Fig. 15.13):

```
> myMeans <- tapply(
+   q1, list(workshop, gender), mean, na.rm = TRUE)

> barplot(myMeans, beside = TRUE)
```

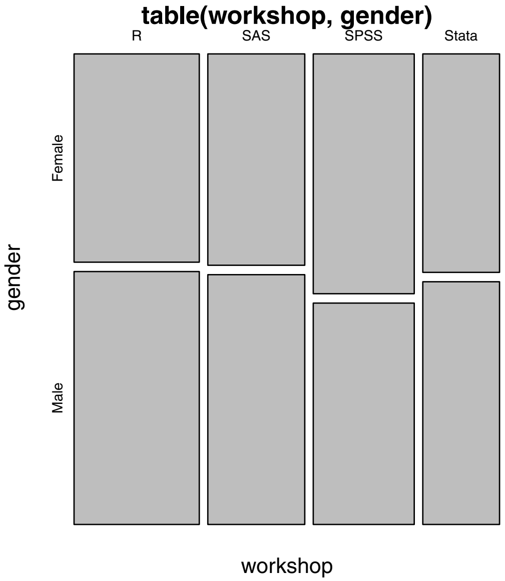


Fig. 15.10. Mosaic plot of workshop by gender, using the `mosaicplot` function. Note the default title

Many of the variations we used with bar plots of counts will work with means, of course. For example, the `horiz = TRUE` argument will flip any of the above examples on their sides.

15.3 Adding Titles, Labels, Colors, and Legends

So far our graphs have been fairly bland. Worse than that, without a legend, some of the bar charts above are essentially worthless. Let us now polish them up. Although we are using bar plots, these steps apply to many of R's traditional graphics functions (Fig. 15.14):

```
> barplot( table(gender, workshop),
+   beside = TRUE,
+   col    = c("gray90", "gray60"),
+   xlab   = "Workshop",
+   ylab   = "Count",
+   main   = "Number of males and females \nin each workshop" )
```

The `barplot` function call above has six arguments.

1. The `table(gender, workshop)` argument generates a two-way table of counts.

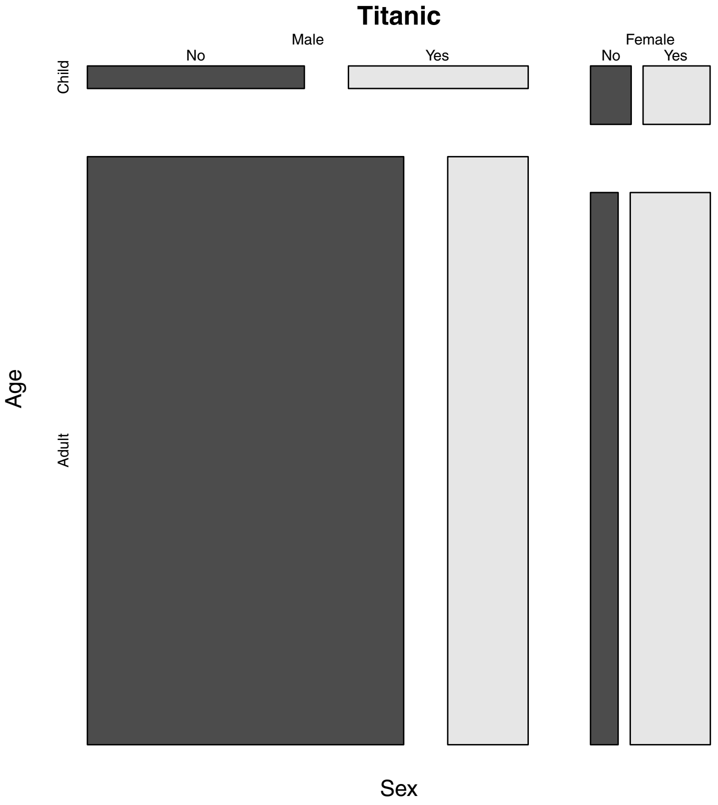


Fig. 15.11. Mosaic plot of Titanic survivors demonstrating display of three factors at once. This plot also includes a default title.

2. The `beside = TRUE` argument places the workshop bars side by side instead of stacking them on top of each other. That is better for perceiving the count per workshop. Leaving it off would result in a stacked bar chart that might be better for estimating relative total attendance of each workshop rather than absolute counts.
3. The `col = c("gray90", "gray60")` argument supplies the *col*ors (shades of gray in this case) for the bars in order of the factor levels in gender. To get a complete list of colors, enter the function `colors()`.
4. The `xlab = "Workshop"` argument specifies the x-axis *lab*el. R differentiates between upper and lower case, and I chose to name my variables all lower case. However, when it comes to labeling output that created extra work.
5. The `ylab = "Count"` argument specifies the y-axis *lab*el. It is probably obvious in this graph that the y-axis represents counts, but I add it here just as an example.

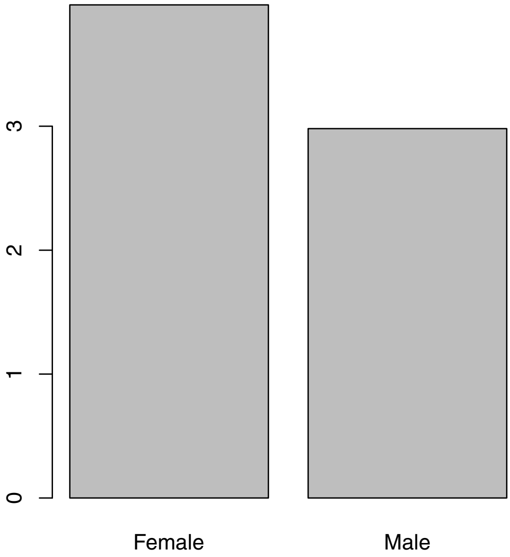


Fig. 15.12. Bar plot of means

6. The `main` argument lists the plot's title. The "`\n`" indicates where the title should skip to a *n*ew line. R uses the backslash character, "\," to introduce control characters. The "n" represents a *n*ew line. Keep in mind that many publications do not want titles on plots because the caption can contain the same information. If you do not see the `main` argument used in other plots in this chapter, any resulting titles are there by default. You can suppress default titles by adding the argument `main = ""`.

Initially, the plot in Fig. 15.14 had no legend. This code caused it to appear after the fact:

```
> legend( "topright",
+   legend = c("Female", "Male"),
+   fill   = c("gray90", "gray60") )
```

The `legend` function call above has three arguments.

1. The first argument positions the legend itself. This can be the values "topleft," "topright," "bottomleft," "bottomright," "right," and "center." It can also be a pair of x, y values such as 10, 15. The 15 is a value of the *y*-axis, but the value of 10 for the *x*-axis was determined by the `bar plot` function. You can query the settings with `par("usr")`, which will return the start and end of the *x*-axis followed by the same figures for the *y*-axis. Those figures for this graph are 0.56, 12.44, –0.17, 17.00. So we see that the *x*-axis goes from 0.56 to 12.44 and the *y*-axis from –0.17 to 17.00.

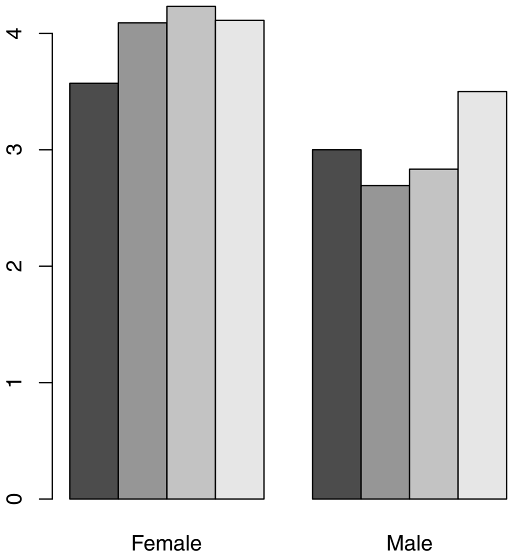


Fig. 15.13. Bar plot of q1 means by workshop (unlabeled) and gender

2. The `legend = c("Female", "Male")` argument supplies the value labels to print in the legend. It is up to you to match their order to the values of the factor gender, so be careful!
3. The `fill = c("gray90", "gray60")` argument supplies colors to match the `col` argument in the `bar plot` function. Again, it is up to you to make these match the labels as well as the graph itself! The `ggplot2` package covered in the next chapter does this for you automatically.

This might seem like an absurd amount of work for a legend, but as with SAS/GRAPH and SPSS's GPL, it trades off some ease of use for power. We are only skimming the surface of R's traditional graphics flexibility.

15.4 Graphics Parameters and Multiple Plots on a Page

R's traditional graphics make it very easy to place multiple graphs on a page. You could also use your word processor to create a table and insert graphs into the cells, but then you would have to do extra work to make them all of the proper size and position, especially if their axes need to line up for comparison. You still have to specify the axes' ranges to ensure compatibility, but then R would size and position them properly. The built-in `lattice` package and the `ggplot2` package will also standardize the axes automatically.

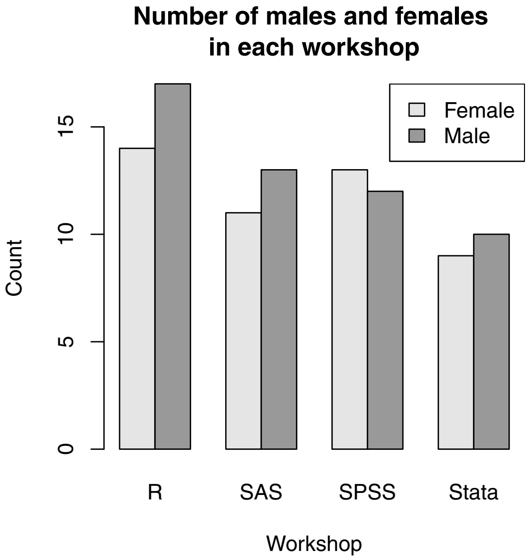


Fig. 15.14. A bar plot with a manually added title, `x`-axis label, legend and shades of gray

Another problem with using word processors to place multiple graphs on a page is that text can shrink so small as to be illegible. Using R to create multiframe plots will solve these problems.

There are three different approaches to creating multiframe plots in R. We will discuss the approach that uses the `par` function. The name `par` is short for graphics *par*ameters. It is easy to use, but is limited to equal-sized plots.

If you need to create more complex multiframe plots, see the help files for either the `layout` function or the `split.screen` function.

R also has some functions, such as `coplot` and `pairs`, that create multiframe plots themselves. Those plots cannot be one piece of an even larger multiframe plot. Even if R could do it, it would be quite a mess!

In traditional R graphics, you use the `par` function to set or query graphic parameter settings. This is the equivalent to the SAS GOPTIONS statement. Entering simply `par()` will display all 71 parameters and how they are set. That is a lot, so we will use the `head` function to print just the top few parameters. In a later section, you will see a table of all the graphics parameters and functions we use.

```
> head( par() )

$xlog
 [1] FALSE
```

```
$ylog
 [1] FALSE

$adj
 [1] 0.5

$ann
 [1] TRUE

$ask
 [1] FALSE

$bg
 [1] "transparent"
```

We can see above that the `xlog` parameter is set to FALSE, meaning the x-axis will not be scaled via the logarithm. The `ask` parameter is also FALSE telling us R will not pause and ask you to click your mouse (or press Enter) to continue. If you submit one graph at a time, this is a good setting. If you instead like to submit several graphs at a time, then you will want R to ask you when you are finished looking at each. Otherwise they will fly past so fast you can barely see them. To change this setting, enter

```
> par(ask = TRUE)
```

Setting it back to FALSE will turn it off and plots will automatically replace one another again. Notice that there is no verification of this for the moment. If you wish to query the setting of any particular parameter, you can enter it in quotes:

```
> par("mfrow")

 [1] 1 1
```

The `mfrow` parameter determines how many rows and columns of graphs will appear on one multiframe plot. The setting "1 1" that `par("mfrow")` displays means that only 1 one graph will appear (one row and one column).

One of the most common uses of the `par` function is to extract and save the original parameter settings so that you can restore them after you make a few changes. Not all the parameter settings can be changed, so you will want to save only the values of those that can be written, i.e., those that are not read-only:

```
> opar <- par(no.readonly = TRUE) #save original parameters

  [make changes, do a graph...]
```

```
> par(opar) #restore original parameters
```

The variable name “opar” is commonly used for this step. I do not use the common “my”prefix on it since these are the original settings, not anything I have set. Simply closing the graphics window or starting a different graphics device will also set the parameters back to their original settings, but it is convenient to be able to do so using programming.

Let us see how we can use the graphics parameters to display four graphs on a page. First we need to set the mfrow parameter to “2, 2” for two rows of graphs, each with two columns. We will create four different bar charts of counts to see the impact of the argument, beside = TRUE (the default value is FALSE). The graphs will appear as we read words: left to right and top to bottom (Fig. 15.15):

```
> opar <- par(no.readonly = TRUE) #save original parameters
>
> par( mfrow = c(2, 2) ) #set to 2 rows, 2 columns of graphs.
>
> barplot(table(gender, workshop))  # top left
> barplot(table(workshop, gender))  # top right
> barplot(table(gender, workshop), beside = TRUE ) # bot. left
> barplot(table(workshop, gender), beside = TRUE ) # bot. right
>
> par(opar) #reset to original parameters
```

15.5 Pie Charts

As the R help file for the pie function says, “Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data.” To prove this to yourself, look at the pie chart in Fig. 15.16 and try to determine which workshop had more attendees, SAS or SPSS. Then look at the bar plot in Fig. 15.6 to see how much easier it is to determine.

The pie function works much in the same way as the barplot function (Fig. 15.16):

```
> pie( table(workshop),
+   col  = c("white","gray90","gray60","black" ) )
```

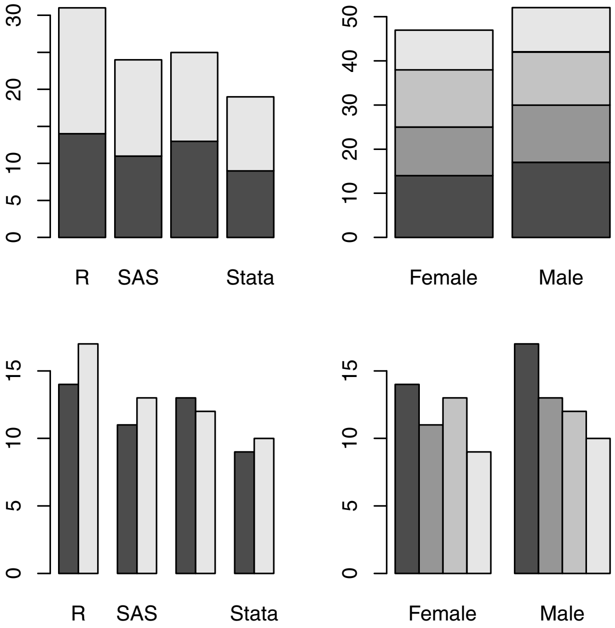


Fig. 15.15. Bar plots of counts by workshop and gender. The top two use the default argument, `beside = FALSE`; the bottom two specify `beside = TRUE`. The left two tabulated `(gender,workshop)`; the right two tabulated `(workshop,gender)`

15.6 Dot Charts

Cleveland popularized the dot chart in his book, *Visualizing Data* [13]. Based on research that showed people excel at determining the length of a line, he reduced the bar chart to just dots on lines. R makes this very easy to do (Fig. 15.17). The arguments for the `dotchart` function are essentially the same as those for the `barplot` function. The default dots in the dot chart do not show up well in the small image below, so I have added `pch = 19` to make the *p*oint *ch*aracter a solid black circle and `cex = 1.5` for *c*haracter *ex*pansion.

```
> dotchart( table(workshop,gender),
+   pch  = 19, cex = 1.5)
```

15.7 Histograms

Many statistical methods make assumptions about the distribution of your data. As long as you have enough data, say 30 or more data points, his-

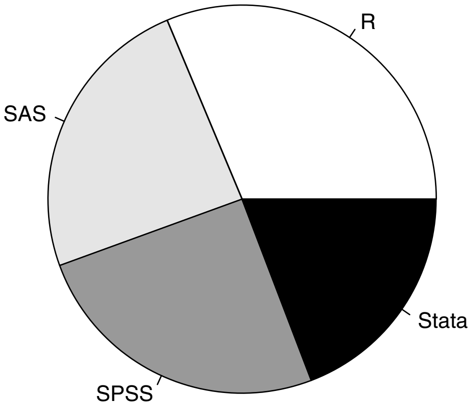


Fig. 15.16. Pie chart of workshop attendance

tograms are a good way to examine those assumptions. We will start with basic histograms, and then examine a few variations.

15.7.1 Basic Histograms

To get a histogram of our posttest score, all we need to do is enter a call to the `hist` function (Fig. 15.18):

```
> hist(posttest)
```

Figure 15.18 shows that the `hist` function prints a main title by default. You can suppress this by adding the argument `main = ""`.

One of the problems with histograms is that they break continuous data down into artificial bins. Trying a different number of bins to see how that changes the view of the distribution is a good idea.

In Fig. 15.19, we use the `breaks = 20` argument to get far more bars than we saw in the default plot. The argument `probability = TRUE` causes the y-axis to display probability instead of counts. That does not change the overall look of the histogram, but it does allow us to add a kernel density fit with a combination of the `lines` function and the `density` function:

```
> hist(posttest, breaks=20, probability = TRUE)
```

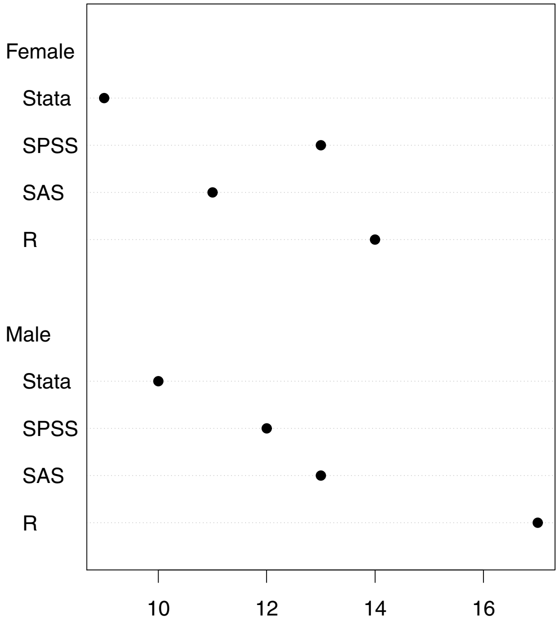


Fig. 15.17. Dot chart of workshop attendance within gender

The `lines` function draws the smooth kernel density calculated by the call to the `density` function. You can vary the amount of smoothness in this function with the `adjust` argument. See the help file for details.

```
> lines( density(posttest) )
```

Finally, we can add tick marks to the x-axis to show the exact data points. That is easy to do with the `rug` function:

```
rug(posttest)
```

Now let us get a histogram of just the males. Recall from Chap. 8, "Selecting Observations," that `posttest[gender == "Male"]` will make the selection we need. We will also add the argument `col = gray60` to give the bars a gray "color" (Fig. 15.20):

```
> hist(posttest[ which(gender == "Male") ],
  col  = "gray60",
  main = "Histogram for Males Only",
```

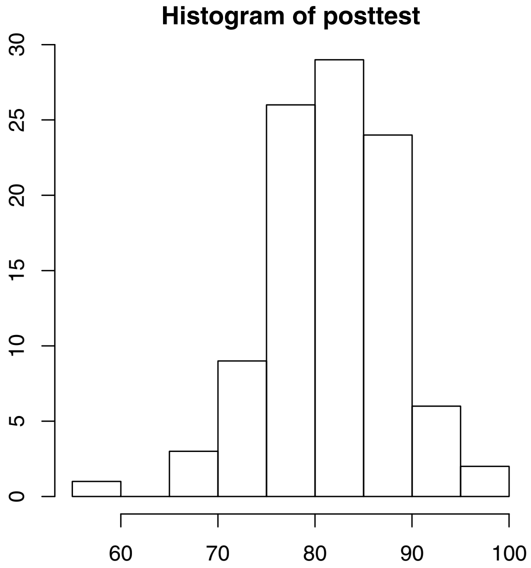


Fig. 15.18. Histogram of posttest

15.7.2 Histograms Stacked

If we want to compare two plots more directly, we can put them onto a multiframe plot (Fig. 15.21) with the graphic parameter command

```
par( mfrow = c(2, 1) )
```

To make them more comparable, we will ensure they break the data into bars at the same spots using the `breaks` argument:

```
> par( mfrow = c(2, 1) )  # Multiframe plot, 2 rows, 1 column.

> hist(posttest, col = "gray90",
+   breaks = c(50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100) )

> hist(posttest[gender == "Male"], col = "gray60",
+   breaks = c(50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100) )

> par( mfrow = c(1, 1) )  # Back to 1 graph per plot.
```

I entered all of the break points to make it clear for beginners. Once you get more used to R, it will be much easier to specify *seq*uences of numbers using the `seq` function:

```
...breaks = seq(10, 100, by=50)
```

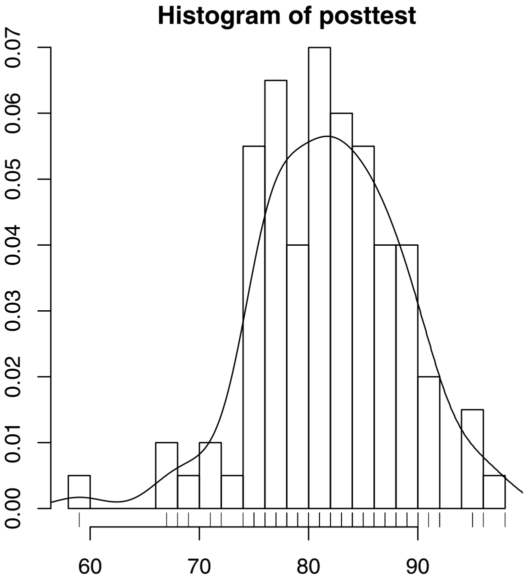


Fig. 15.19. A histogram of posttest with `breaks=20`, a kernel density curve and "rug" tick-marks for every data point on the x-axis

See Chap. 12, "Generating Data," for more ways to generate values like this.

15.7.3 Histograms Overlaid

That looks nice, but we could get a bit fancier and plot the two graphs right on top of one another. The next few examples start slow and end up rather complicated compared to similar plots in SAS or SPSS. In the next chapter, the same plot will be *much* simpler. However, this is an important example because it helps you learn the type of information held inside a graphics object.

Our entire data set contains males and females, so a histogram of both will have taller bars than a histogram for just the males. Therefore, we can overlay a histogram for males on top of one for both genders (Fig. 15.22).

The `add = TRUE` argument is what tells the `hist` function to add the second histogram on top of the first. Notice below that we also use the `seq` function to generate the numbers 50, 55,. . . ,100 without having to write them all out as we did in the previous example:

```
> hist(posttest, col = "gray90",
+   breaks = seq(from = 50, to = 100, by = 5) )

> hist(posttest[gender == "Male"], col = "gray60",
```

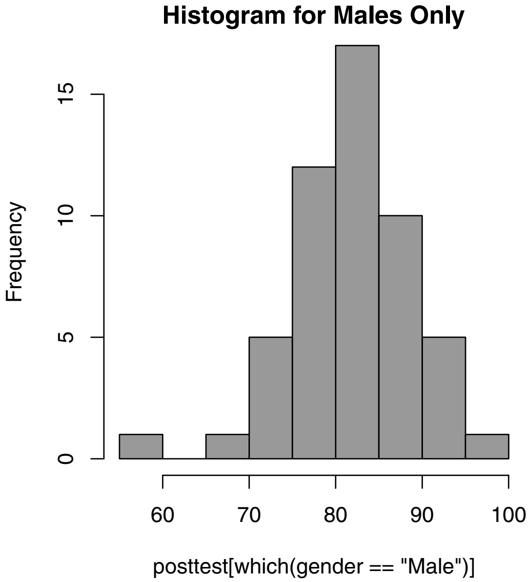


Fig. 15.20. Histogram of posttest for males only

```
+   breaks = seq(from = 50, to = 100, by = 5),
+   add    = TRUE )

> legend( "topright",
+   legend = c("Female", "Male"),
+   fill   = c("gray90", "gray60") )
```

This looks good, but we did have to manually decide what the breaks should be. In a more general-purpose program, we may want R to choose the break points in the first plot and then apply them to the second automatically. We can do that by saving the first graph to an object called, say, myHistogram.

```
> myHistogram <- hist(posttest, col = "gray90")
```

Now let us use the `names` function to see the names of its components.

```
> names(myHistogram)

 [1] "breaks"      "counts"      "intensities" "density"
 [5] "mids"        "xname"       "equidist"
```

One part of this object, myHistogram$breaks, stores the break points that we will use in the second histogram. The graph of all the data appears at this point and we can print the contents of myHistogram$breaks. Notice that R has decided that our manually selected break point of 50 was not needed:

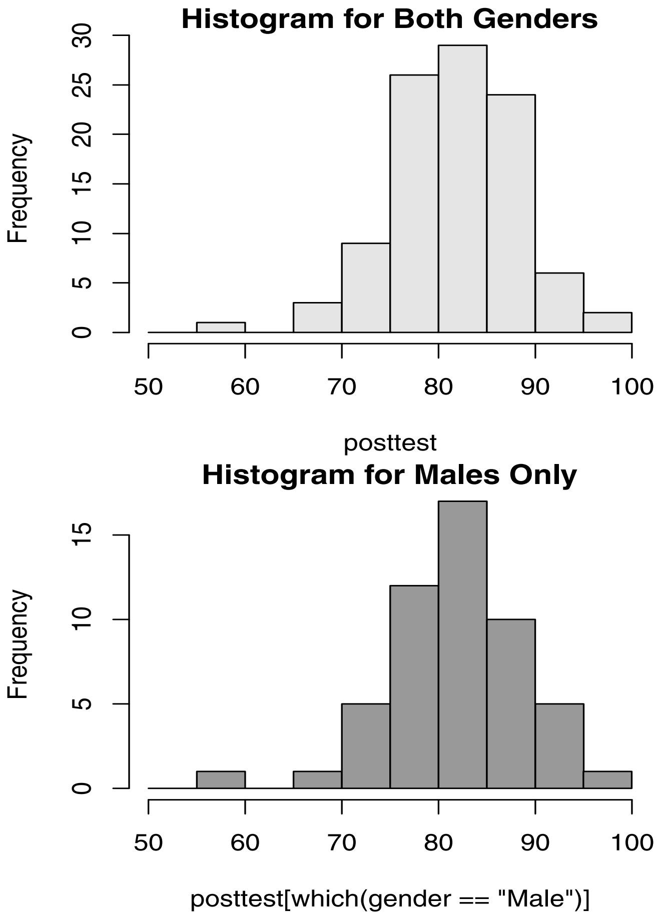


Fig. 15.21. Multiframe plot of posttest histograms for whole data set(top) and just the males (bottom)

```
> myHistogram$breaks

 [1]  55  60  65  70  75  80  85  90  95 100
```

Let us now do the histogram for males again, but this time with the argument:

```
breaks = myHistogram$breaks
```

so the break points for males will be the same as those automatically chosen for the whole sample (Fig. 15.23):

```
> hist(posttest[gender == "Male"], col =  gray60 ,
```

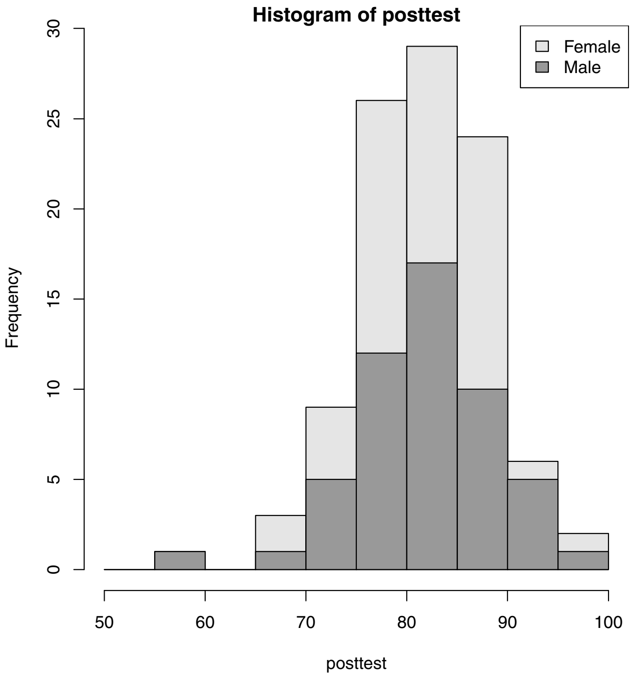


Fig. 15.22. A histogram of posttest for all data (tall bars) overlaid with just the males. The difference between the two represents the females

```
+      add = TRUE, breaks = myHistogram$breaks)

> legend( "topright",
+   legend = c("Female", "Male"),
+   fill   = c("gray90", "gray60") )
```

This is essentially the same as the previous graph, but the axis fits better by not extending all the way down to 50. Of course, we could have noticed that and fixed it manually if we had wanted. To see what else a histogram class object contains, simply enter its name. You see the breaks listed as its first element:

```
> myHistogram
$breaks
 [1]  55  60  65  70  75  80  85  90  95 100

$counts
 [1]  1  0  3  9 26 29 24  6  2

$intensities
```

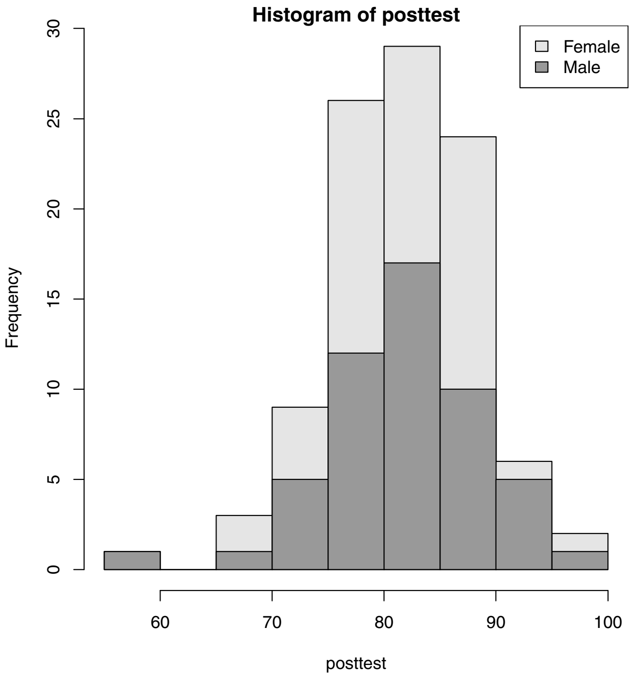


Fig. 15.23. Same histogram as previous one but now bar break points were chosen from the whole data set and then applied to males

```
 [1] 0.002 0.000 0.006 0.018 0.052 0.058 0.048 0.012 0.004

$density
 [1] 0.002 0.000 0.006 0.018 0.052 0.058 0.048 0.012 0.004

$mids
 [1] 57.5 62.5 67.5 72.5 77.5 82.5 87.5 92.5 97.5

$xname
 [1] "posttest"

$equidist
 [1] TRUE

attr(,"class")
 [1] "histogram"
```

Now that the plot is saved in myHistogram, we can display it any time with

```
plot(myHistogram)
```

You can change the plot object using standard R programming methods; you are not restricted to modifying it with the function that created it. That is a tricky way to work, but you can see how someone working to develop a new type of graph would revel in this extreme flexibility.

15.8 Normal QQ Plots

A normal QQ plot plots the quantiles of each data point against the quantiles that each point would get if the data were normally distributed. If these points fall on a straight line, they are likely to be from a normal distribution.

Histograms give you a nice view of a variable's distribution, but if you have fewer than 30 or so points, the resulting histogram is often impossible to interpret. Another problem with histograms is that they break the data down into artificial bins, so unless you fiddle with the bin size, you might miss an interesting pattern in your data. A QQ plot has neither of these limitations.

So why use histograms at all? Because they are easier to interpret. At a statistical meeting I attended, the speaker displayed a QQ plot and asked the audience, all statisticians, what the distribution looked like. It was clearly not normal and people offered quite an amusing array of responses! When the shape is not a straight line, it takes time to learn how the line's shape reflects the underlying distribution. To make matters worse, some software reverses the roles of the two axes! The plot shown in Fig. 15.24, created using the `qq.plot` function from John Fox's `car` package, has the theoretical quantiles on the x-axis, like SAS.

```
> library("car")

> qq.plot(posttest,
+   labels = row.names(mydata100),
+   col    = "black" )

> detach("package:car")
```

The call to the `qq.plot` function above has three arguments.

1. The variable to plot.
2. `labels = row.names(mydata100)` allows you to interactively identify any point you wish, and label it according to the values you request. Your cursor will become a cross-hair, and when you click on (or near) a point, its label will appear. The escape key will end this interactive mode.
3. `col = "black"` sets the color, which is red by default.

R also has a built-in function for QQ plots called `qqnorm`. However, it lacks confidence intervals. It also does not let you identify points without calling the `identify` function (graph not shown):

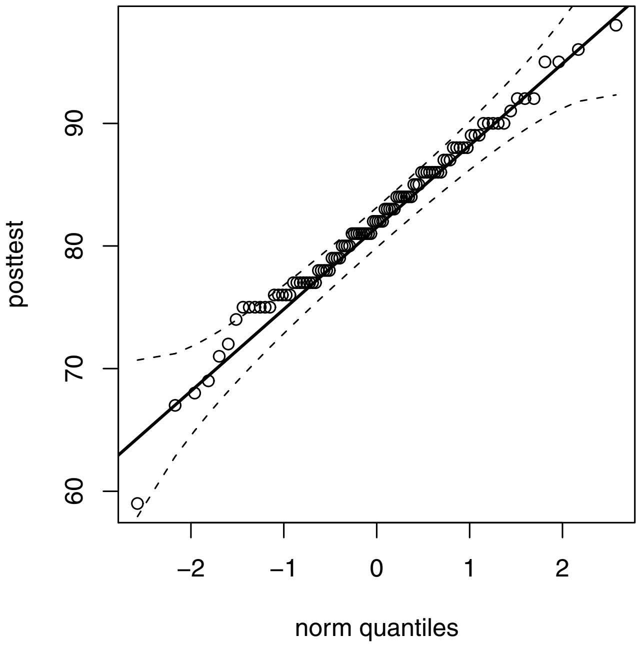


Fig. 15.24. A normal quantile plot of posttest using `qq.plot` from the `car` package

```
myQQ <- qqnorm(posttest)

identify(myQQ)
```

15.9 Strip Charts

Strip charts are scatter plots for one variable. Since they plot each data point, you might see clusters of points that would be lost in a box plot or error bar plot. The first strip chart function call below uses "jitter" or random noise added to help you see points that would otherwise be obscured by falling on top of other point(s) at the same location. The second one uses `method = "stack"` to stack the points like little histograms instead (Fig. 15.25). Here we do both types in a single multiframe plot. For details, see Sect. 15.4, "Graphics Parameters and Multiple Plots on a Page."

```
> par( mfrow = c(2, 1) )  #set up 2 columns, 1 row multiplot

> stripchart(posttest, method = "jitter",
+   main = "Stripchart with Jitter")
```

```
> stripchart(posttest, method = "stack",
+   main = "Stripchart with Stacking")

> par( mfrow = c(1, 1) )  # restore to 1 plot
```

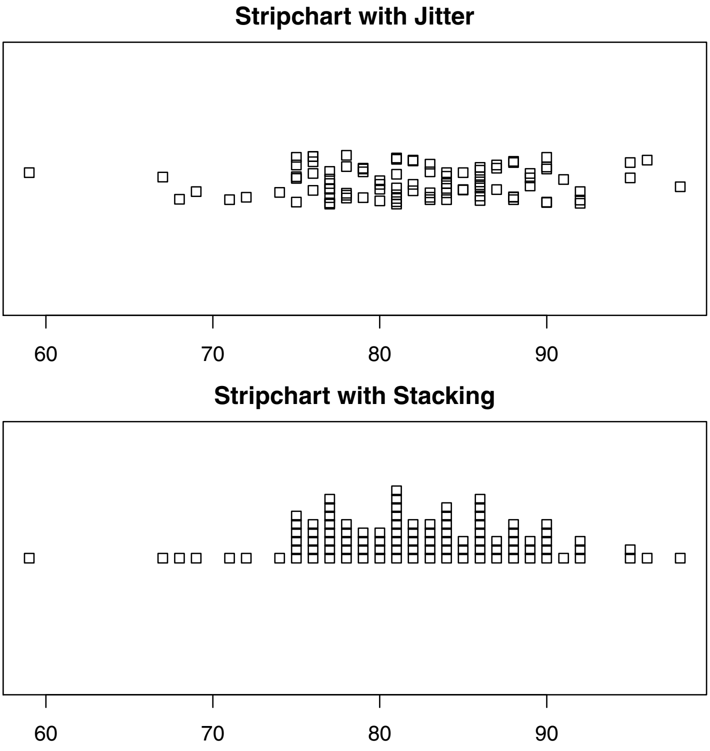


Fig. 15.25. Strip chart demonstrating the methods jitter and stack

Let us now compare groups using strip charts (Fig. 15.26). Notice the use of the formula `posttest˜workshop` to compare the workshop groups. You can reverse the order of those two variables to flip the scatter vertically, but you would lose the automated labels for the factor levels:

```
> par( las = 2, mar = c(4, 8, 4, 1) + 0.1  )

> stripchart(posttest ˜ workshop, method = "jitter")
```

The `par` function here is optional. I use it just to angle the workshop value labels so that they are easier to read. Turned that way, they need more

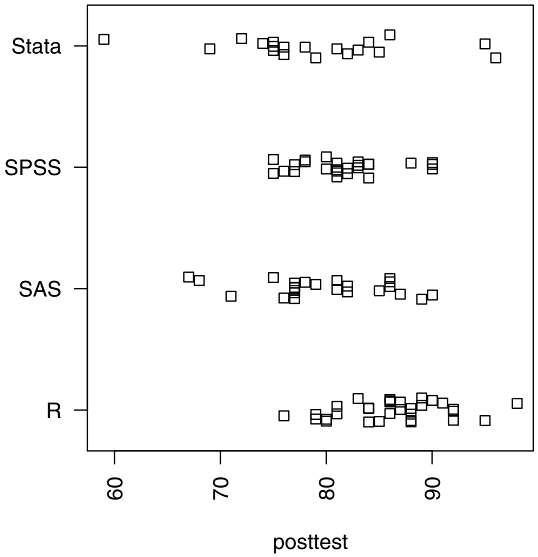


Fig. 15.26. A strip chart of posttest scores by workshop

Table 15.1. Graphics arguments for traditional high-level graphics functions

`main`	Supplies the text to the *main* title. Example: `plot(pretest, posttest, main = "My Scatter plot")`
`sub`	Supplies the text to the *sub*-title. Example: `plot(pretest, posttest...sub = "My Scatter plot")`
`xlab`	Supplies the x-axis *lab*el (variable labels from `Hmisc` package are ignored). Example: `plot(pretest, posttest...xlab = "Score Before Training")`
`xlim`	Specifies the lower and upper *lim*its of the x-axis. Example: `plot(pretest, posttest, xlim = c(50,100) )`
`ylab`	Supplies the y-axis *lab*el (variable labels from `Hmisc` package are ignored.). Example: `plot(pretest, posttest...ylab = "Score After Training")`
`ylim`	Specifies the lower and upper *lim*its of the y-axis. Example: `plot(pretest, posttest, ylim = c(50, 100) )`

space on the left-hand side of the chart. The `par` function call above uses two arguments to accomplish this.

1. The `las = 2` argument changes the *l*abel *a*ngle *s*etting of the text to be perpendicular to the y-axis. For more settings, see Table 15.1, "Graphics Parameters."
2. The `mar = c(4, 8, 4, 1) + 0.1` argument sets the size of the *mar*gins at the bottom, left, top, and right sides, respectively. The point of this was to make the left side much wider so that we could fit the labels turned on their sides. For details, see Table 15.2, "Graphics Parameters."

The `stripchart` function call above contains two arguments:

Table 15.2. Graphics parameters to set or query using only `par()`

`ask`	`par(ask = TRUE)` causes R to prompt you before showing a new graphic. The default setting of FALSE causes it to automatically replace any existing plot with a new one. If you run a program in batch mode, you should set this to FALSE!
`family`	Sets font family for text. `par(family = "sans")`, the default, requests Helvetica or Arial. `par(family = "serif")` requests a serif font like Times Roman. `par(family = "mono")` requests a monospaced font like Courier. `par(family = "symbol")` requests math and Greek symbols.
`mar`	Sets *mar*gin size in number of lines. The default setting is `par( mar = c(5, 4, 4, 2) + 0.1 )` which sets the number of margin lines in order (bottom, left, top, right). For graphs that lack labels, you may want to decrease the margins to eliminate superfluous white space. The settings 4, 4, 2, 2 work well if you do not have a title above or below your plot. An example that sets the label angle style to perpendicular and provides eight lines on the left side is `par(las = 2, mar = c(4, 8, 4, 1) + 0.1)`.
`mfrow`	Sets up a *m*ulti*f*rame plot to contain several other plots. R will plot to them left to right, top to bottom. This example yields three rows of two plots. `par( mfrow = c(3, 2) )`. This returns it to a single plot: `par( mfrow = c(1, 1) )`.
`mfcol`	Sets up a *m*ulti*f*rame plot like `mfrow`, but writes plots in *col*umns from top to bottom, left to right.
`new`	Setting `par(new = TRUE)` tells R that a new plot has already been started so that it will not erase what is there before adding to it.
`par()`	Will display all traditional graphics *par*ameters. `opar <- par(no.readonly = TRUE)` saves the writable parameters so that you can later reset them with `par(opar)`.
`ps`	Sets the *p*oint *s*ize of text. For example, to select 12-point text: `par(ps = 12)`.
`usr`	Shows you the coordinates of a plot in the form x-start, x-stop, y-start, y-stop.
`xlog`	Setting `par(xlog = TRUE)` requests a *log*arithm-transformed x-axis, including tick-marks.
`ylog`	Setting `par(ylog = TRUE)` requests a *log*arithm-transformed y-axis, including tick marks.

1. The formula `posttest ~ workshop`, which asks for a strip chart of posttest for every value of workshop.
2. The `method = "jitter"` argument that tells it to add random noise to help us see the points that would otherwise be plotted on top of others.

Table 15.3. Graphics functions to add elements to existing plots.

`abline`	A function that adds straight *line*(s) to an existing plot in the form $y = a + bx$. Example of intercept 0, slope 1: `abline(a = 0, b = 1, lty = 5)`. E.g., linear model line: `abline( lm(posttest ~ pretest), lty = 1 )`.
`arrows`	A function that draws an arrow with the arguments (from–x, from–y, to–x, to–y). The optional `length` argument sets the length of the arrowhead lines. E.g., `arrows(65, 85, 58.5, 59, length = 0.1)`.
`axis`	A function that adds an axis to an existing plot. E.g., `axis(4)` adds it to the right side (1 = bottom, 2 = left, 3 = top, 4 = right).
`box`	A function that adds a box around an existing plot. E.g., `box()`.
`grid`	A function that adds a set of vertical and horizontal lines to an existing plot. E.g., `grid()`.
`lines`	A function that adds line(s) to an existing plot (need not be straight). E.g., lowess fit: `lines( lowess(posttest ~ pretest), lty = 3 )`.
`text`	A function that adds text to an existing plot. Its arguments are the x,y position of the plot, the text, and the position of the text. Its `pos` argument *pos*itions text relative to x,y values with 1=bottom, 2=left, 3=top, 4=right. E.g., `text (65, 85, "Fit is linear", pos = 3)`.

15.10 Scatter and Line Plots

Scatter plots are helpful in many ways. They show the nature of a relationship between two variables. Is it a line? A curve? Is the variability in one variable the same at different levels of the other? Is there an outlier now visible that did not show up when checking minimum and maximum values one variable at a time? A scatter plot can answer all these questions.

The `plot` function takes advantage of R's object orientation by being generic. That is, it looks at the class of objects you provide it and takes the appropriate action.

For example, when you give it two continuous variables, it does a scatter plot (Fig. 15.27):

```
> plot(pretest, posttest)
```

Note the "92" on the lower left point. That did not appear on the graph at first. I wanted to find which observation that was, so I used the `identify` function.

```
> identify(pretest, posttest)

 [1] 92
```

The `identify` function lets you label points by clicking on them. You list the x and y variables in your plot, and optionally provide a `label =` argument to specify an ID variable, with the row names as the default. The function will label the point you click on or near. Unfortunately, it does not let you draw a box around a set of points. You must click on each one, which can get a bit tedious. When finished, users can right-click and choose "stop" (or press the *Esc* key on Windows). It will then print out the values you chose. If you assigned the result to a vector as in:

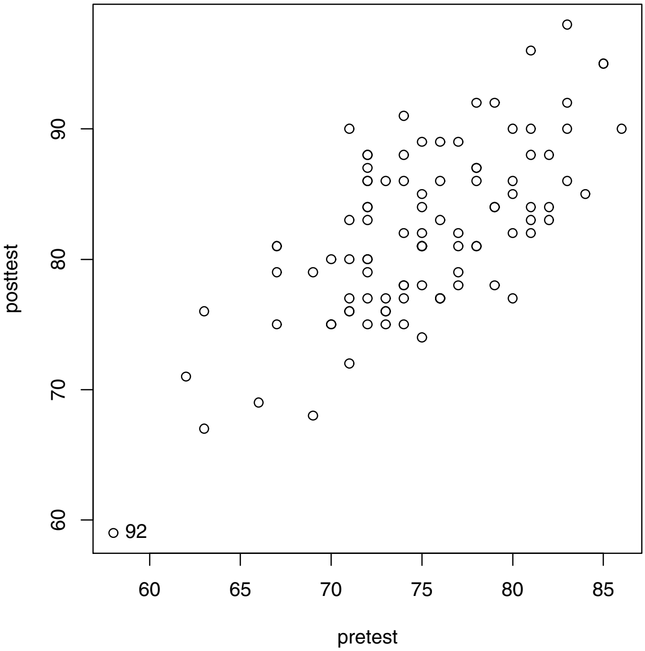


Fig. 15.27. A scatter plot of pretest and posttest. Observation 92 was identified after the plot was created

```
myPoints <- identify(pretest, posttest)
```

then that vector would contain all your selected points. You could then use logic like:

```
summary( mydata100[!myPoints, ] )
```

(not myPoints) to exclude the selected points and see how it changes your analysis. Below, I use a logical selection to verify that it is indeed observation 92 that has the low score.

```
> mydata100[pretest < 60, ]

   gender workshop q1 q2 q3 q4 pretest posttest
92   Male    Stata  3  4  4  4      58       59
```

Back at the scatter plot, you can specify the `type` argument to change how the points are displayed. The values use the letter “p” for points, the letter “l” for lines, “b” for both points and lines, and “h” for histogram-like lines that rise vertically from the x-axis. Connecting the points using either “l” or “b” makes sense only when the points are collected in a certain order, such as time series data. As you can see in Fig. 15.28, that is not the case with our data, so those appear as a jumbled nest of lines:

```
> par( mfrow = c(2, 2) )  # set up a 2x2 multiframe plot

> plot( pretest, posttest, type = "p", main = "type=p" )
> plot( pretest, posttest, type = "l", main = "type=l" )
> plot( pretest, posttest, type = "b", main = "type=b" )
> plot( pretest, posttest, type = "h", main = "type=h" )

> par( mfrow = c(1, 1) )  # set parameter back to 1 plot
```

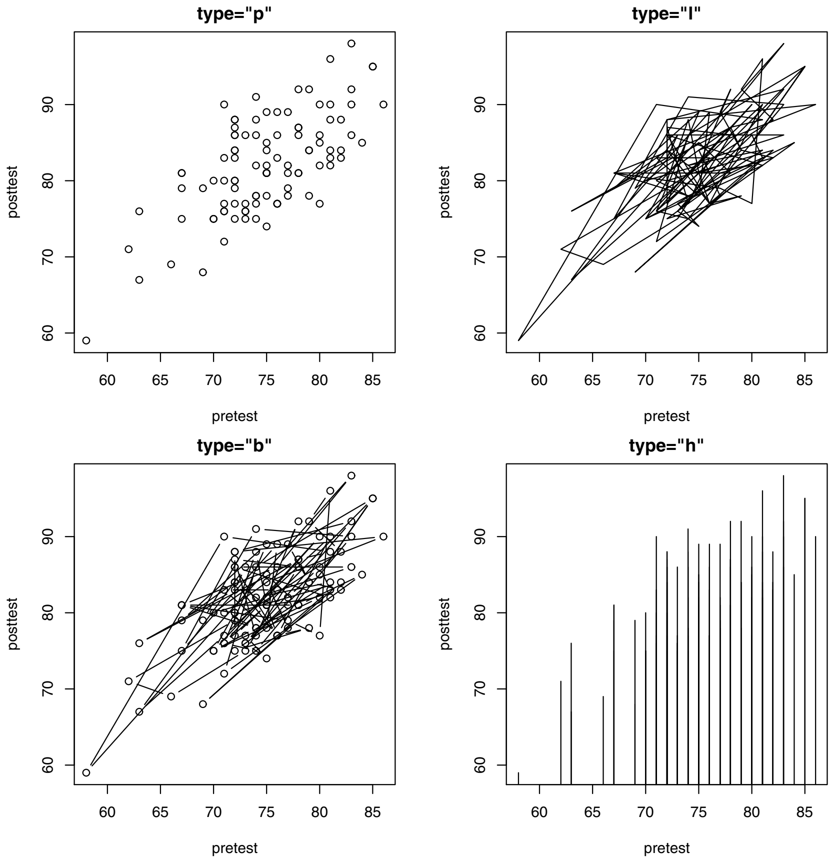


Fig. 15.28. Various scatter plots demonstrating the effect of the `type` argument

15.10.1 Scatter Plots with Jitter

The more points you have in a scatter plot, the more likely you are to have them overlap, potentially hiding the true structure of the data. This is a particularly bad problem with Likert-scale data since Likert-scales use few values. These data are typically averaged into scales that are more continuous, but we will look at an example with just two Likert measures, q1 and q4.

Jitter is simply some random variation added to the data to prevent overlap. You will see the `jitter` function in the second plot in Fig. 15.29. Its arguments are simply the variable to jitter, and a value “3” for the amount of jitter. That was derived from trial and error. The bigger the number, the greater the jitter.

```
> par( mfrow = c(1, 2) )  # set up 1x2 multiframe plot

> plot( q1, q4,
+   main = "Likert Scale Without Jitter")

> plot( jitter(q1, 3), jitter(q4, 3),
+   main = "Likert Scale With Jitter")

> par( mfrow = c(1, 1) )  # reset to single plot
```

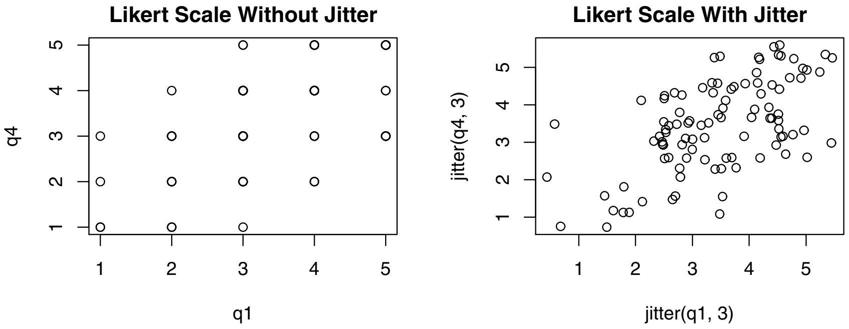


Fig. 15.29. Scatter plots demonstrating the impact of jitter on five–point Likert–scale data

15.10.2 Scatter Plots with Large Data Sets

The larger your data set, the more likely it is that points will fall on top of one another, obscuring the structure in the data. R’s traditional graphics offers several ways to deal with this problem, including decreasing point size, replacing sets of points with hexagonal bins, and using density functions to display shading instead of points.

Scatter Plots with Jitter and Small Points

The simplest way to keep points from plotting on top of one another is to use jitter as described in the previous section. It is also helpful to use the smallest *p*oint *ch*aracter with `pch = "."`. Here is an example using 5000 points (Fig. 15.30). The R code that generated pretest2 and posttest2 is included in the program at the end of this chapter.

```
> plot( jitter(pretest2,4), jitter(posttest2,4), pch = ".",
+   main = "5,000 Points Using pch =  .  \nand Jitter")
```

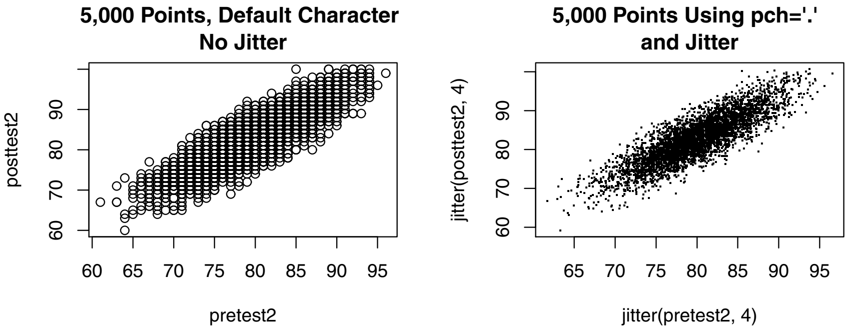


Fig. 15.30. Scatter plots showing how to handle large data sets. The plot on the left using default settings, leaves many points obscured. The one on the right uses much smaller points and jitters them, allowing us to see more points

Hexbin Plots

Another way of plotting large amounts of data is a `hexbin` plot (Fig. 15.31). This is provided via the `hexbin` package, written by Carr et al. [11]. Note that `hexbin` uses the `lattice` package, which in turn uses the `grid` graphics system. That means that you cannot put multiple graphs on a page or set any other parameters using the `par` function.

```
> library("hexbin")

Loading required package: grid
Loading required package: lattice
```

```
> plot( hexbin(pretest2, posttest2),
+   main = "5,000 Points Using Hexbin")

> detach("package:hexbin")
```

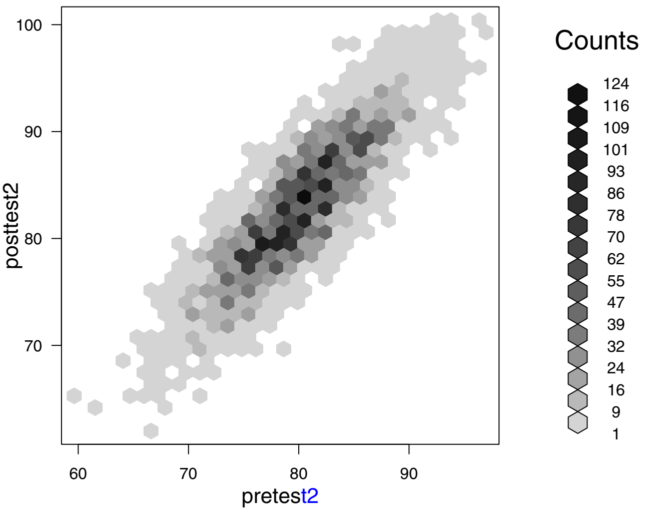


Fig. 15.31. A hexbin plot that divides large amounts of data into hexagonal bins to show structure in large data sets. Each bin can represent many original points

Scatter Plots with Density Shading

The last approach we will look at uses the smoothScatter function. It uses a density function to color or shade large amounts of data. It works very similarly to the plot function:

```
> smoothScatter( pretest2, posttest2,
+   main="5,000 Points Using smoothScatter")
```

You can see the resulting plot in Fig. 15.32.

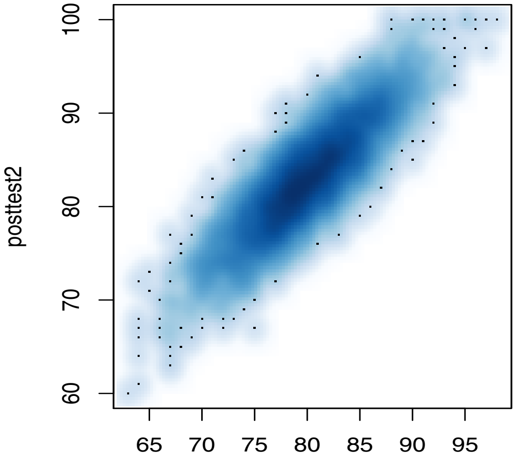


Fig. 15.32. A smoothScatter plot shades the density of the points

15.10.3 Scatter Plots with Lines

You can add straight lines to your plots with the `abline` function. It has several different types of arguments. Let us start with a scatter plot with only points on it (Fig 15.33):

```
> plot(posttest ~ pretest)
```

Now let us add a horizontal line and a vertical line at the value 75. You might do this if there were a cutoff below which students were not allowed to take the next workshop:

```
> abline( h = 75, v = 75 )
```

The `abline` function exists to add straight lines to the last plot you did, so there is no `add = TRUE` argument. Next, let us draw a diagonal line that has pretest equal to posttest. If the workshop training had no effect, the scatter would lie on this line. This line would have a y-intercept of 0 and a slope of 1. The `abline` function does formulas in the form $y = a + bx$, so we want to specify $a = 0$ and $b = 1$. We will also set the *l*ine *ty*pe to dashed with `lty = 5`:

```
> abline( a = 0, b = 1, lty = 5 )
```

Next, let us add a regression fit using the `lm` function within `abline`. The `abline` function draws straight lines, so let us use the `lines` function along with the `lowess` function to draw a smoothly fitting `lowess` curve. The `legend` function allows you to choose the order of the labels, so I have listed them as they appear from top to bottom in the upper left corner of the plot.

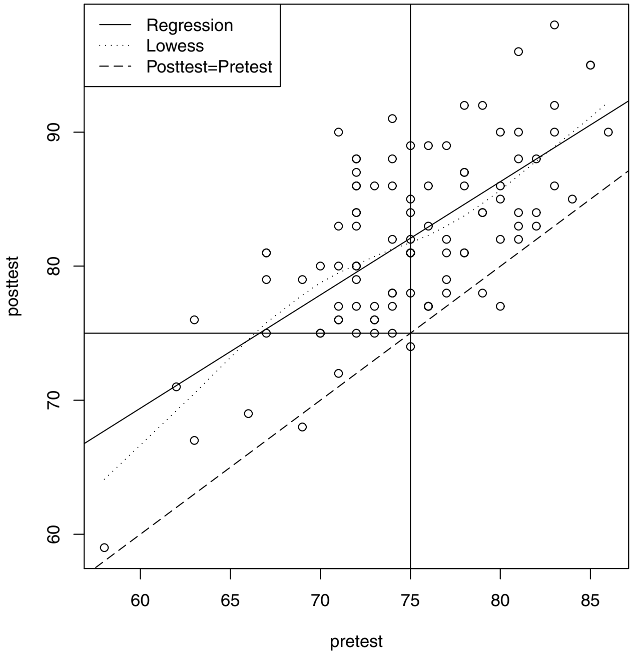


Fig. 15.33. A scatter plot demonstrating how to add various types of lines, as well as a legend and title

```
> abline( lm( posttest ~ pretest ),    lty = 1 )

> lines( lowess( posttest ~ pretest ), lty = 3 )

> legend(60, 95,
+   legend = c("Regression", "Lowess", "Posttest=Pretest"),
+   lty    = c(1, 3, 5) )
```

15.10.4 Scatter Plots with Linear Fit by Group

As we saw in the last section, it is easy to add regression lines to plots using R's traditional graphics. Let us now turn our attention to fitting a regression line separately for each group (Fig. 15.34). First, we will use the `plot` function to display the scatter, using the `pch` argument to set the *p*oint *ch*aracters based on gender. Gender is a factor that cannot be used directly to set point characters. Therefore, we are using the `as.numeric` function to convert it on the fly. That will cause it to use two symbols. If we used the `as.character` function instead, it would plot the actual characters M and F:

```
> plot(posttest ~ pretest,
+   pch = as.numeric(gender) )
```

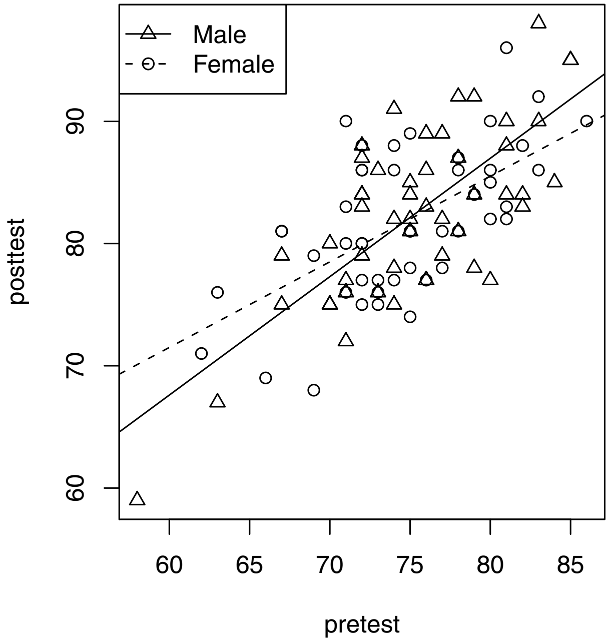


Fig. 15.34. A plot that displays different point symbols and regression lines for each gender

Next, we simply use the `abline` function as we did before, but basing our regression on the males and females separately. For details about selecting observations based on group membership, see Chap. 8, "Selecting Observations."

```
> abline( lm( posttest[ which(gender == "Male") ]
+            ~ pretest[ which(gender == "Male") ] ),
+         lty = 1 )

> abline( lm( posttest[ which(gender == "Female") ]
+            ~ pretest[ which(gender == "Female") ] ),
+         lty = 2 )

legend( "topleft",
  legend = c("Male", "Female"),
  lty    = c(1, 2),
  pch    = c(2, 1) )
```

15.10.5 Scatter Plots by Group or Level (Coplots)

Coplots are scatter plots conditioned on the levels of a third variable. For example, getting a scatter plot for each workshop is very easy. In Fig. 15.35, the box above the scatter plots indicates which plot is which. The bottom left is for R, the bottom right is for SAS, the top left is for SPSS, and the top right is for Stata. This is a rather odd layout for what could have been a simple 2 × 2 table of labels, but it makes more sense when the third variable is continuous. Both the `lattice` and `ggplot2` packages do a better job of labeling such plots. The `coplot` function is easy to use. Simply specify a formula in the form `y~x`, and list your conditioning variable after a vertical bar, "`|`":

```
> coplot( posttest ~ pretest | workshop)
```

The next plot, Fig. 15.36, is conditioned on the levels of q1. Look at the grey bars at the top labeled "Given: q1." The four grey bars are positioned left to right, bottom to top, to match the scatter plots below. The grey bar to the lower left covers the values of q1 from 0 to 3.5 (1, 2, and 3). That indicates that the scatter plot on the lower left is also shows the pretest-posttest scatter for observations that have those low q1 values. The second grey bar from the bottom and to the right covers values of q1 from 1.5 to 3.5 (2 and 3 in our case). Therefore, the scatter plot on the lower right also displays points whose observations are limited to q1 values of 2 and 3. The values of q1 in each plot overlap to prevent you from missing an important change that may occur at a cut point of q1. Here is the code that created Fig. 15.36:

```
> coplot( posttest ~ pretest | q1)
```

The functions that modify plots, apply to plots you create one at a time. That is true even when you build a multiframe plot one plot at a time. However, when you use a single R function that creates its own multiframe plot, such as `coplot`, you can no longer use those functions. See the help file for ways to modify coplots.

15.10.6 Scatter Plots with Confidence Ellipse

Confidence ellipses help visualize the strength of a correlation as well as provide a guide for identifying outliers. The `data.ellipse` function in John Fox's `car` package makes these quite easy to plot (Fig. 15.37). It works much like the `plot` function with the first two arguments being your x and y variables, respectively. The `levels` argument lets you specify one confidence limit as shown below, or you could specify a set in the usual way, for example,

```
levels = c(.25, .50, .75)
```

If you leave the `col = "black"` argument off, it will display in its default color of red:

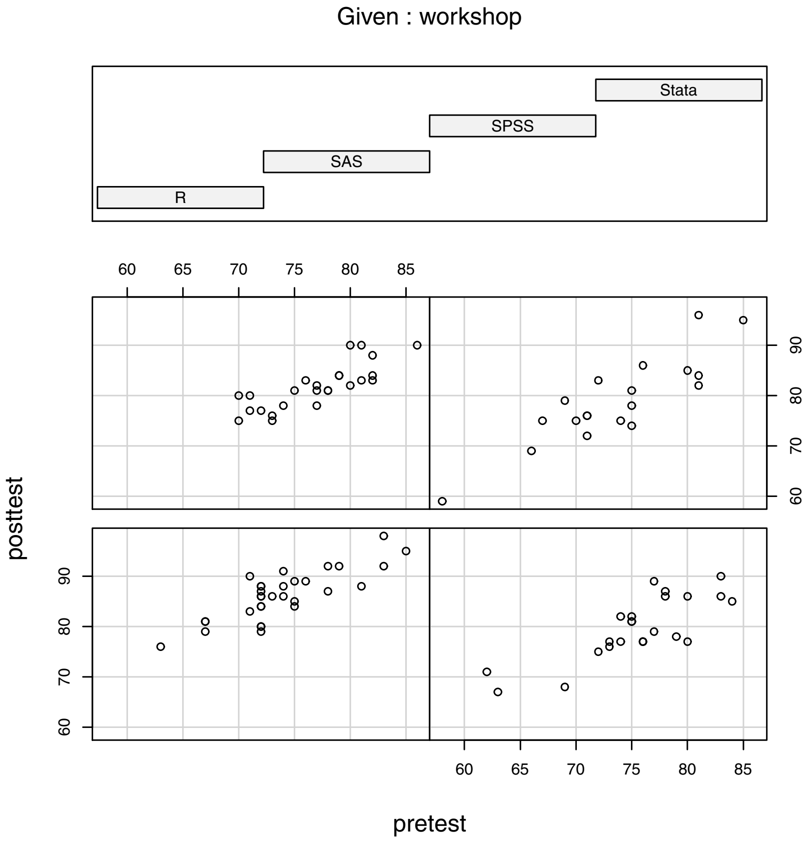


Fig. 15.35. A set of scatter plots for each workshop created using the `coplot` function. The bars in the top frame show which workshop each plot represents, from left to right, starting on the bottom row

```
> library("car")

> data.ellipse(pretest, posttest,
+   levels = .95,
+   col = "black")

> detach("package:car")
```

15.10.7 Scatter Plots with Confidence and Prediction Intervals

As we saw previously, adding a regression line to a scatter plot is easy. However, adding confidence intervals is somewhat complicated. The `ggplot2` pack-

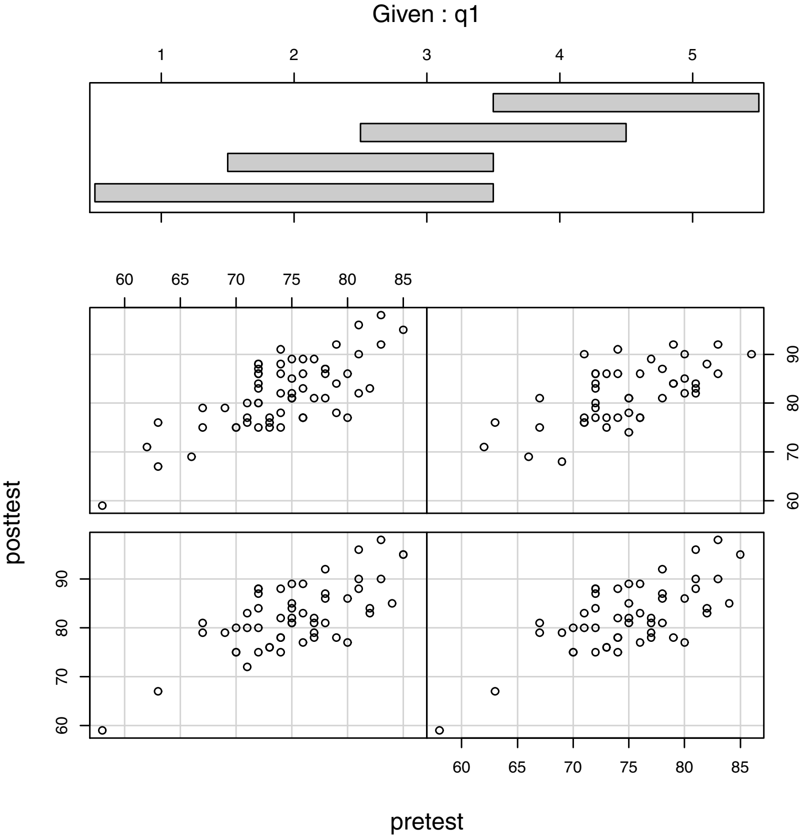


Fig. 15.36. A set of scatter plots for various levels of q1. The bars in the top frame show the values of q1 for each plot, and how they overlap. Going left to right, and from bottom to top, the values of q1 increase

age, covered in the next chapter, makes getting a line and 95% confidence band about the line easy. However, getting confidence limits about the predicted points is complicated, even with `ggplot2`.

Let us start with a simple example. We will create a vector x and three vectors y1, y2, and y3. The three ys will represent a lower confidence limit, the prediction line, and the upper confidence limit:

```
> x <- c(1, 2, 3, 4)

> y1 <- c(1, 2, 3, 4)

> y2 <- c(2, 3, 4, 5)
```

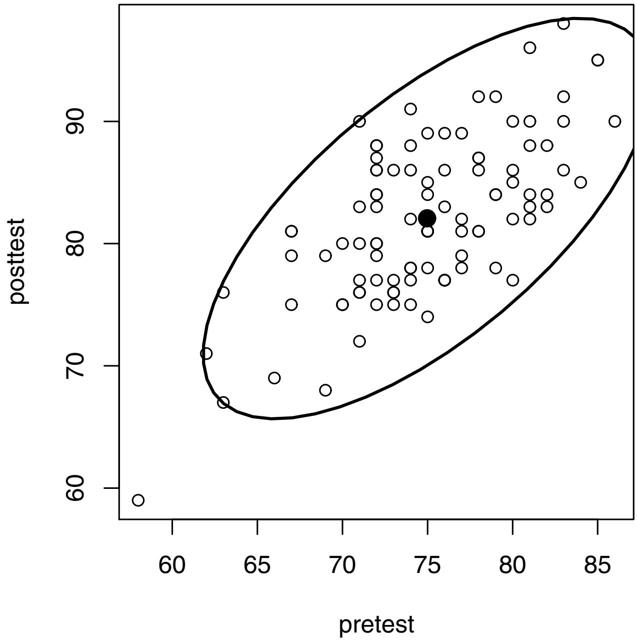


Fig. 15.37. Scatter plot with 95% confidence ellipse

```
> y3 <- c(3, 4, 5, 6)

> yMatrix <- cbind(y1, y2, y3)

> yMatrix

     y1 y2 y3
 [1,]  1  2  3
 [2,]  2  3  4
 [3,]  3  4  5
 [4,]  4  5  6
```

Now we will use the `plot` function to plot x against y2 (Fig. 15.38). We will specify the `xlim` and `ylim` arguments to ensure the axes will be big enough to hold the other y variables. The result is rather dull! I have used the `cex = 1.5` argument to do *c*haracter *ex*pansion of 50% to make it easier to see in this small size:

```
> plot(x, y2, xlim = c(1, 4), ylim = c(1, 6), cex = 1.5)
```

Next, we will use the `matlines` function (Fig. 15.39). It can plot a vector against every column of a matrix. We will use the `lty` argument to specify

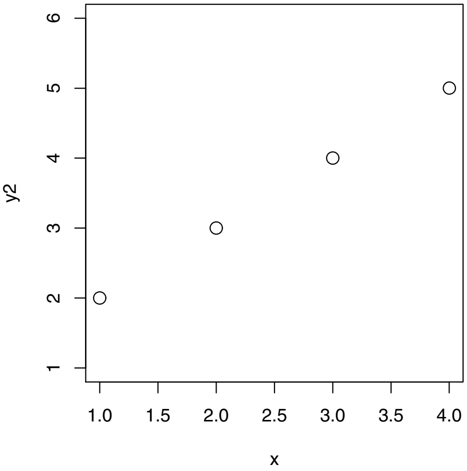


Fig. 15.38. A scatter plot of just four points on a straight line, a simple foundation to build on

line types of dashed, solid, dashed for y1, y2, and y3, respectively. Finally, we will specify the line color as `col = "black"` to prevent it from providing a different color for each line, as it does by default:

```
> matlines( x, yMatrix, lty = c(2, 1, 2), col = "black" )

> rm( x, y1, y2, y3, yMatrix)
```

This plot represents the essence of our goal. Now let us fill in the details. First, we need to create a new data frame that will hold a "well-designed" version of the pretest score. We want one that covers the range of possible values evenly. That will not be important for linear regression, since the spacing of points will not change the line, but we want to use a method that would work even if we were fitting a polynomial regression.

```
> myIntervals <-
  data.frame( pretest = seq(from = 60, to = 100, by = 5) )

> myIntervals

  pretest
1      60
2      65
3      70
4      75
5      80
```

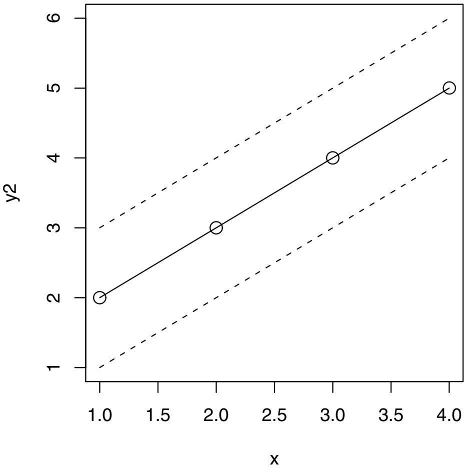


Fig. 15.39. A scatter plot that demonstrates the basic idea of a regression line with confidence intervals. With a more realistic example, the confidence bands would be curved

```
6       85
7       90
8       95
9      100
```

Now let us create a regression model using the `lm` function, and store it in myModel. Note that we have attached the larger practice data set, mydata100, so the model is based on its pretest and posttest scores.

```
> myModel <- lm( posttest ~ pretest )
```

Next, we will use the `predict` function to apply myModel to the myIntervals data frame we just created. There are two types of intervals you might wish to plot around a regression line. The 95% prediction interval (also called tolerance interval) is the wider of the two and is for predicted values of y for new values of x (`interval = "prediction"`). The 95% confidence interval is for the mean of the y values at a given value of x (`interval = "confidence"`). We will run the `predict` function twice to get both types of intervals, and then look at the data. Notice that the `newdata` argument tells the `predict` function which data set to use:

```
> myIntervals$pp <- predict( myModel,
+  interval = "prediction", newdata = myIntervals)

> myIntervals$pc <- predict( myModel,
+  interval = "confidence", newdata = myIntervals)
```

```
> myIntervals

  pretest  pp.fit  pp.lwr  pp.upr  pc.fit  pc.lwr  pc.upr
1      60  69.401  59.330  79.472  69.401  66.497  72.305
2      65  73.629  63.768  83.491  73.629  71.566  75.693
3      70  77.857  68.124  87.591  77.857  76.532  79.183
4      75  82.085  72.394  91.776  82.085  81.121  83.050
5      80  86.313  76.579  96.048  86.313  84.980  87.646
6      85  90.541  80.678 100.405  90.541  88.468  92.615
7      90  94.770  84.696 104.843  94.770  91.855  97.684
8      95  98.998  88.636 109.359  98.998  95.207 102.788
9     100 103.226  92.507 113.945 103.226  98.545 107.906
```

Now we have all of the data we need. Look at the names pp.fit, pp.lwr, pp.upr. They are the fit and lower/upper prediction confidence intervals. The three variables whose names begin with "pc" are the same variables for the line's narrower confidence interval. But why the funny names? Let us check the class of just "pp."

```
> class( myIntervals$pp )

 [1] "matrix"

> myIntervals$pp

      fit    lwr     upr
1  69.401 59.330  79.472
2  73.629 63.768  83.491
3  77.857 68.124  87.591
4  82.085 72.394  91.776
5  86.313 76.579  96.048
6  90.541 80.678 100.405
7  94.770 84.696 104.843
8  98.998 88.636 109.359
9 103.226 92.507 113.945
```

The `predict` function has added two matrices to the myIntervals data frame. Since the `matlines` function can plot all columns of a matrix at once, this is particularly helpful. Now let us use this information to complete our plot with both types of confidence intervals. The only argument that is new is setting the limits of the y-axis using the `range` function. I did that to ensure that the y-axis was wide enough to hold both the pretest scores and the wider prediction interval, pp. Finally, we see the plot in Fig. 15.40.

```
> plot( pretest, posttest,
+   ylim = range(myIntervals$pretest,
+     myIntervals$pp, na.rm = TRUE) )

> matlines(myIntervals$pretest, myIntervals$pc,
+   lty = c(1, 2, 2), col = "black")

> matlines(myIntervals$pretest, myIntervals$pp,
+   lty = c(1, 3, 3), col = "black")
```

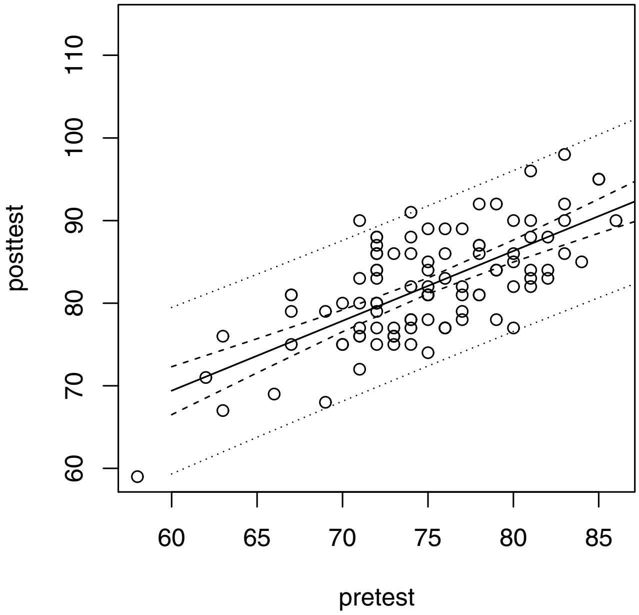


Fig. 15.40. Here we finally see the complete plot with both types of confidence intervals using our practice data set

15.10.8 Plotting Labels Instead of Points

When you do not have too much data, it is often useful to plot labels instead of symbols (Fig. 15.41). If your label is a single character, you can do this using the `pch` argument. If you have a character variable, `pch` will accept it directly. In our case, gender is a factor, so we will use the `as.character` function to covert it on the fly:

```
> plot(pretest, posttest,
+   pch = as.character(gender) )
```

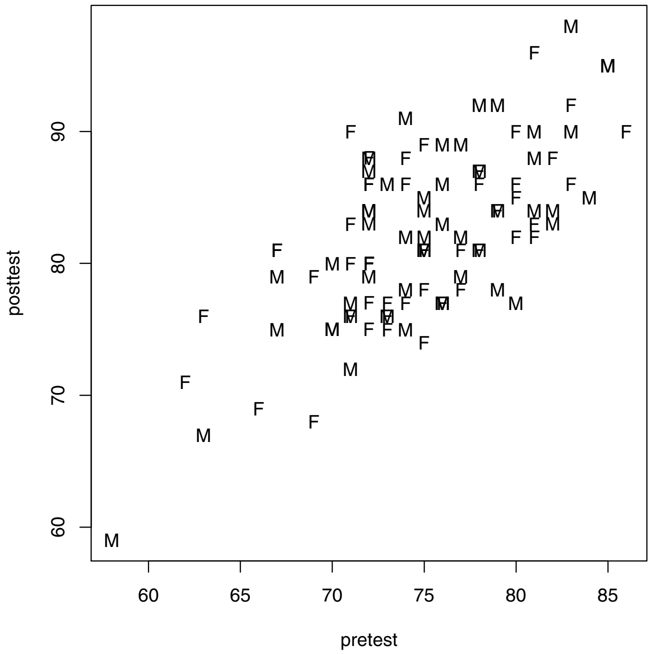


Fig. 15.41. A scatter plot using gender as its point symbol. This method only works with a single character

If you provide the `pch` function a longer label, it will plot only the first character. To see the whole label, you must first plot an empty graph with `type = "n"`, for *n*o points, and then add to it with the `text` function. The `text` function works just like the `plot` function but it plots labels instead of points. In Fig. 15.42, below, we use the `row.names` function to provide labels. If we had wanted to plot the workshop value instead, we could have used `label = as.character(workshop)`:

```
> plot(pretest, posttest, type = "n" )

> text(pretest, posttest,
+   label = row.names(mydata100) )
```

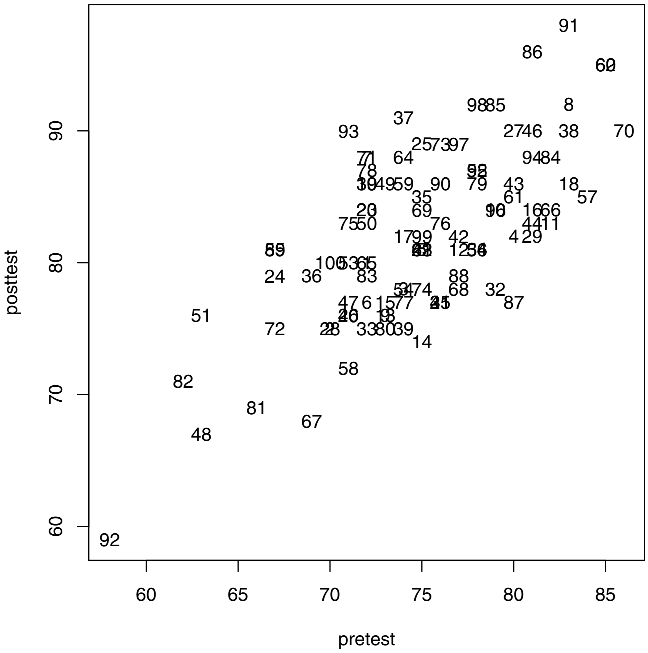


Fig. 15.42. A scatter plot with `row.names` as labels. This used the `text` function because it can handle labels longer than one character

15.10.9 Scatter Plot Matrices

When you have many variables to study, it can be helpful to get pairwise scatter plots of them all (Fig. 15.43). This is easy to do with the `plot` function. The first one below will suffice, but it inserts gaps between each pair of graphs. The second one removes those gaps and shrinks the size of the labels by 10%. The second plot is the only one shown.

```
> plot(mydata100)  # Not shown.

> plot(mydata100, gap = 0, cex.labels = 0.9)
```

You can use the entire data frame in this type of plot, and it will convert factors to numeric and give you the following warning. The plots involving factors would appear as strip plots:

```
Warning message:
  In data.matrix(x) : class information lost
  from one or more columns
```

As with any generic function, you can see what other functions `plot` will call given different classes of data. The `methods` function will show you.

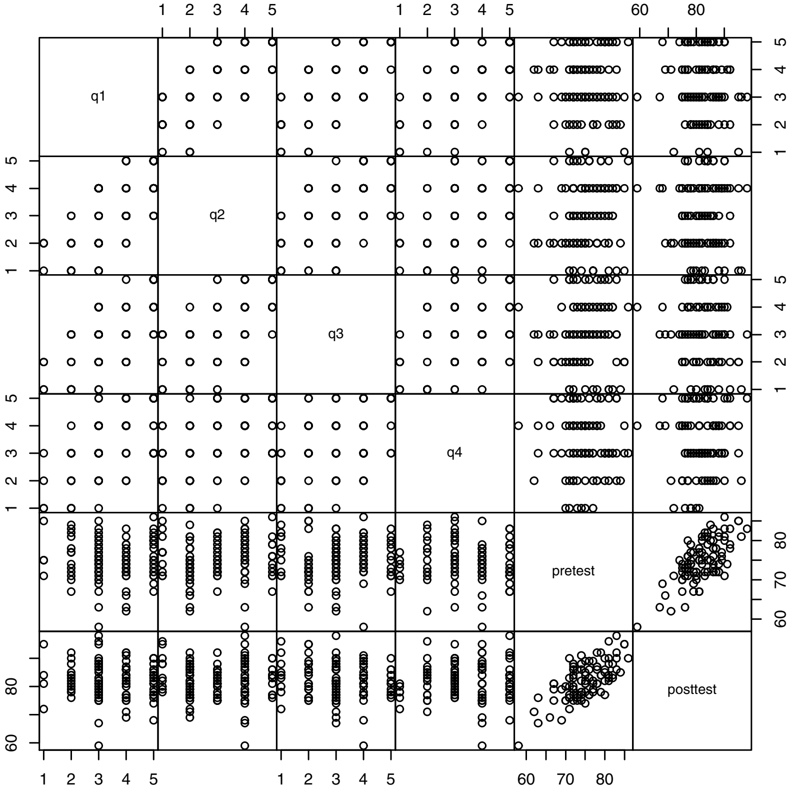


Fig. 15.43. A scatter plot matrix shows small plots for every combination of variables. This was created using the `plot` function

```
> methods(plot)

 [1] plot.acf*             plot.data.frame*    plot.Date*
 [4] plot.decomposed.ts* plot.default        plot.dendrogram*
 [7] plot.density          plot.ecdf           plot.factor*
[10] plot.formula*         plot.hclust*        plot.histogram*
[13] plot.HoltWinters*     plot.isoreg*        plot.lm
[16] plot.medpolish*       plot.mlm            plot.POSIXct*
[19] plot.POSIXlt*         plot.ppr*           plot.prcomp*
[22] plot.princomp*        plot.profile.nls*   plot.spec
[25] plot.spec.coherency plot.spec.phase     plot.stepfun
[28] plot.stl*             plot.table*         plot.ts
```

```
 [31] plot.tskernel*      plot.TukeyHSD
```

When you use the `plot` function on a data frame, it passes the data on to the `plot.data.frame` function. When you read the help file on that, you find that it then calls the `pairs` function! So to find out the options to use, you can finally enter `help("pairs")`. That will show you some interesting options, including adding histograms on the main diagonal and Pearson correlations on the upper right panels.

In the next example, I have added the `panel.smooth` values to draw lowess smoothing. Although I call the `pairs` function directly here, I could have used the `plot` function to achieve the same result. Since the `pairs` function creates a multiframe plot by itself, you must use its own options to modify the plot. In this case, we cannot add a smoothed fit with the `lines` function; we must use `panel.smooth` instead (Fig. 15.44):

```
> pairs(mydata100[3:8], gap = 0,
+   lower.panel = panel.smooth,
+   upper.panel = panel.smooth)
```

15.11 Dual-Axis Plots

The usual plot has a y-axis on only the left side. However, to enhance legibility you may wish to place it on the right side as well (Fig. 15.45). If you have the same y-variable measured in two different units (dollars and euros, fahrenheit & celsius, and so on), you may wish a second axis in those units.

If you have two different y variables to plot, using a dual-axis plot is often not a good idea because you can stretch or shrink the two axes to make the same comparison appear totally different [26]. Stacking two different plots in a multiframe plot is often a better way to handle that situation. The axes can still be manipulated, but the reader focuses more on each axis when it appears alone on one plot. For examples of how to stack plots, see Sect. 15.4, "Graphics Parameters and Multiple Plots on a Page."

We will simply place the same axis on both sides, which can make it easier to read the y-values of points near the right-hand side of the plot. We will plot the same graph, once without axes, then adding one on the right, then the left.

First, we will need to add space in the margins, especially the right side where the new axis will need labeling. That is done with the following command, which changes the margins from their default values of (5,2,2,4)+0.1 to 5, 5, 4, 5 as it applies to the bottom, left, top, and right sides, respectively:

```
> par( mar = c(5, 5, 4, 5) )
```

Next we will draw the main plot. Since we are focusing on improving the axes, let us add labels for the x-axis and the first y-axis:

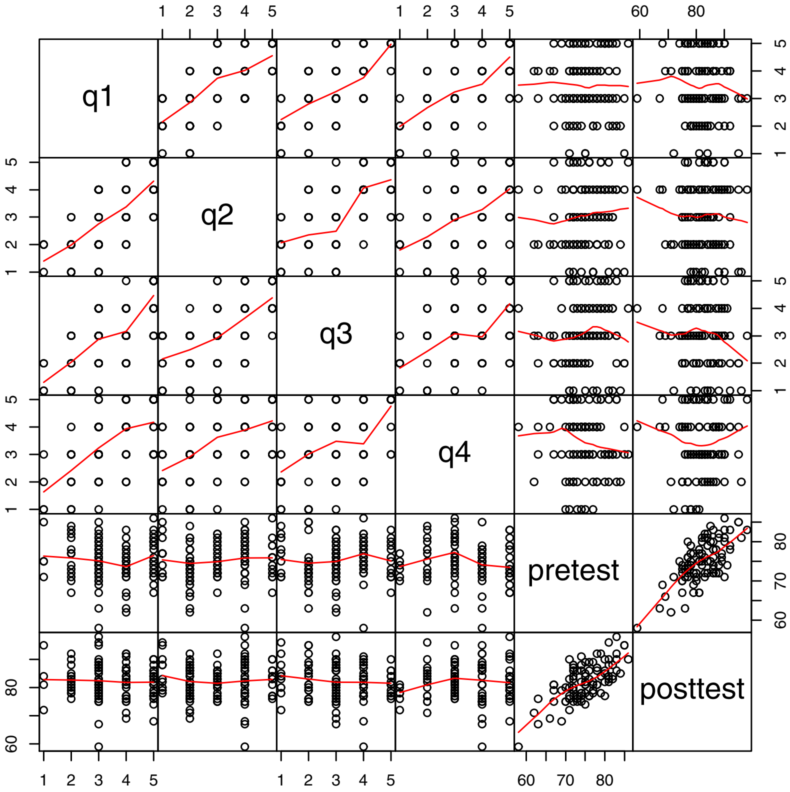


Fig. 15.44. A scatter plot matrix with smoothed fits from the `pairs` function

```
> plot(pretest, posttest,
+   xlab = "Pretest Score",
+   ylab = "Posttest Score")
```

The next call to the `axis` function asks R to place the current axis on the right side (side 4):

```
> axis(4)
```

Next we will add text to the margin around that axis with the `mtext` function. It places its text on line 3 of side 4 (the right).

```
> mtext("Posttest Score", side = 4, line = 3)
```

Since we are making the graph easier to read from the left or right side, let us also embellish it with a grid that will connect the tick-marks on the two axes, (as well as add vertical lines) using the `grid` function:

```
> grid()
```

The complete plot is shown in Fig. 15.45.

```
> par( new = TRUE )
> plot( pretest, posttest, xlim = c(60, 90) )
```

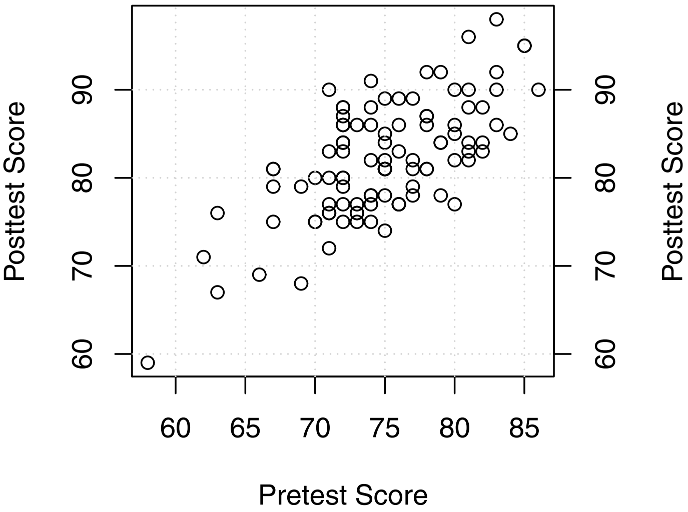


Fig. 15.45. A scatter plot demonstrating an additional axis on the right side and the addition of a grid

15.12 Box Plots

Box plots put the middle 50% of the data in a box with a median line in the middle and lines, called "whiskers," extending to ±1.5 times the height of the box (i.e., the 75th percentile minus the 25th). Points that lie outside the whiskers are considered outliers. You can create side-by-side box plots using the `plot` function when the first variable you provide is a factor (Fig. 15.46):

```
> plot(workshop, posttest)
```

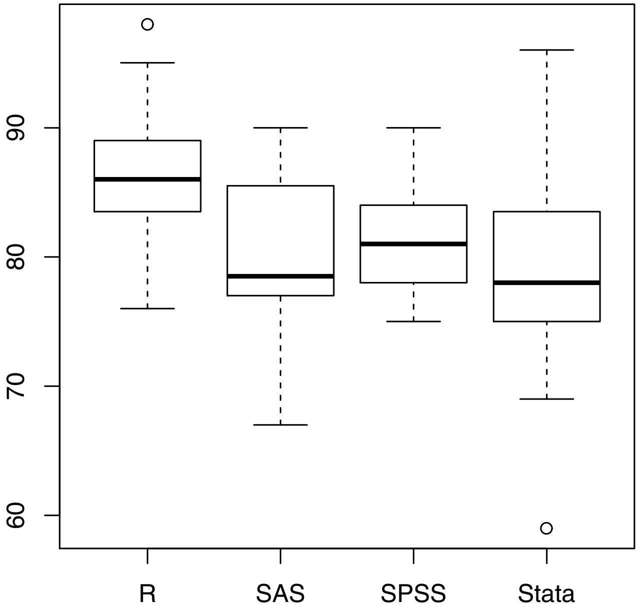


Fig. 15.46. A box plot of posttest scores for each workshop

There are several variations we can do. First, let us use the `mfrow` argument of the `par` function to create a 2 × 2 multiframe plot.

```
> par( mfrow = c(2, 2) )
```

Next we will do a box plot of a single variable, posttest. It will appear in the upper left of Fig. 15.47. We will use the box plot function since it gives us more flexibility.

```
> boxplot(posttest)
```

Then we will put pretest and posttest side side–by–side in the upper right of Fig. 15.47. The `notch` argument tells it to create notches that, when they do not overlap, provide "strong evidence" that the medians differ. It appears in the upper right.

```
> boxplot(pretest, posttest, notch = TRUE)
```

Next we will use a formula to get box plots for each workshop, side–by–side. It appears in the bottom left of Fig. 15.47.

```
> boxplot(posttest ~ workshop)
```

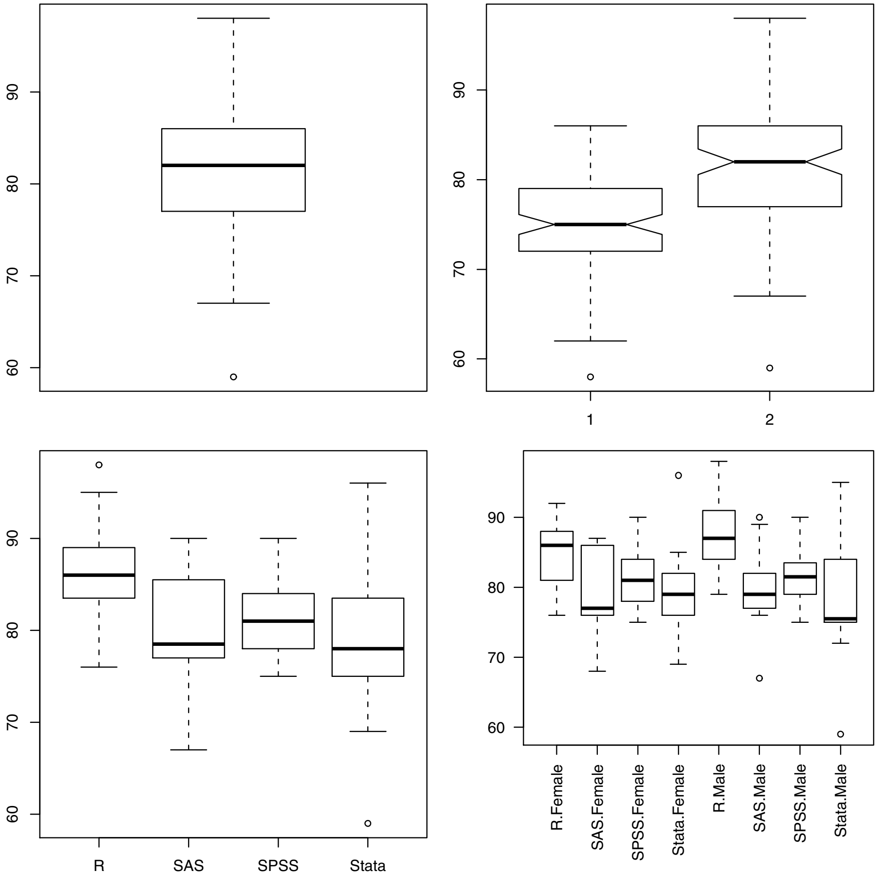


Fig. 15.47. Various box plots. The upper left is the posttest score. The upper right shows pretest and posttest, with notches indicating possible median differences. The lower left shows posttest scores for each workshop. The lower right shows posttest for the gender and workshop combinations, as well as labels perpendicular to the x-axis

Finally, we will create a box plot for each workshop:gender combination. If we tried to use the `plot` function using the workshop:gender syntax, it would create two box plots, one for workshop and another for gender. To get one for all combinations, we must use the `boxplot` function.

Generating the combination of factors will create long value labels, like "SAS.Female." So we will need to change the *l*abel *a*xis *s*tyle of the x-axis using `las = 2` and increase the bottom *mar*gin with `mar = c(8, 4, 4, 2) + 0.1`. We will set those parameters back to their defaults immediately afterwards. The plot appears in the bottom right of Fig. 15.47.

```
> par(las = 2, mar = c(8, 4, 4, 2) + 0.1 )

> boxplot(posttest ~ workshop:gender)

> par( las = 0, mar = c(5, 4, 4, 2) + 0.1 )
```

This is a bit of a mess and would probably be more interpretable if we had done one box plot of workshop for each gender and stacked them in a multiframe plot. For instructions on how to do that, see Sect. 15.4, “Graphics Parameters and Multiple Plots on a Page.” The ggplot2 package does a *much* better version of this same plot in Fig. 16.38.

15.13 Error Bar Plots

The gplots package, by Warnes et al. [67], has a plotmeans function that plots means with 95% confidence bars around each. The confidence intervals assume the data come from a normal distribution (Fig. 15.48). Its main argument is in the form measure~group:

```
> library("gplots")

> plotmeans(posttest ~ workshop)

> detach("package:gplots")
```

15.14 Interaction Plots

R has a built-in interaction.plot function that plots the means for a two-way interaction (Fig. 15.49). For a three-way or higher interactions, you can use the by function to repeat the interaction plot for each level of the other variables.

In Fig. 15.49, the males seem to be doing slightly better with R and the females with Stata. Since the plot does not display variability, we do not have any test of significance for this interpretation.

```
> interaction.plot(workshop, gender, posttest)
```

15.15 Adding Equations and Symbols to Graphs

Any of the functions that add text to a graph, such as main, sub, xlab, and ylab can display mathematical equations. For example, a well well-known formula for multiple regression is, $\hat{\beta} = (X^tX)^{-1}X^tY$. You can add this to any existing graph using the following call to the text function:

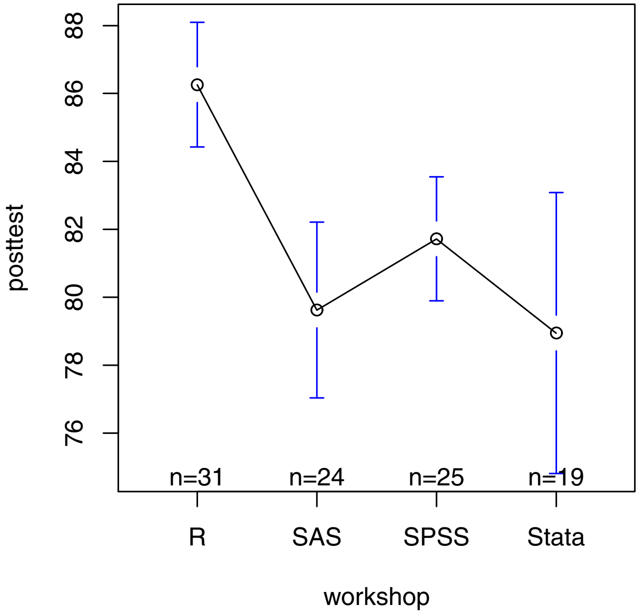


Fig. 15.48. An error bar plot showing the posttest mean for each workshop, along with 95% confidence intervals

```
text(66, 88, "My Example Formula")
text(65, 85,
  expression( hat(beta) ==  (X^t * X)^{-1} * X^t * Y) )
```

The `text` function adds any text at the *x,y* position you specify, in this case 66 and 88. So the use of it above adds, "My Example Formula" to an existing graph at that position. In the second call to the `text` function, we also call the `expression` function. When used on any of the text annotation functions, the `expression` function tells R to interpret its arguments in a special way that allows it to display a wide variety of symbols. In this example, beta will cause the Greek letter β to appear. Two equal signs in a row (x == y) result in the display of one (x = y). Functions like `hat` and `bar` will cause those symbols to appear over their arguments. So `hat(beta)` will display a "`^`" symbol over the Greek letter beta. This example formula appears on the plot in Fig. 15.50. You can see several tables of symbols with `help("plotmath")`.

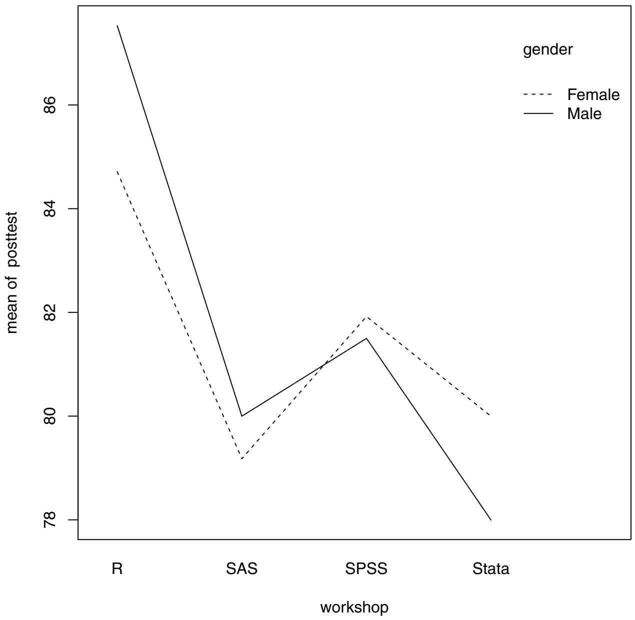


Fig. 15.49. An interaction plot of posttest means by the workshop and gender combinations

15.16 Summary of Graphics Elements and Parameters

As we have seen in the examples above, R's traditional graphics has a range of functions and arguments that you can use to embellish your plots. Tables 15.1 through 15.3 summarize them.

15.17 Plot Demonstrating Many Modifications

Below is a program that creates a rather horrible looking plot (Fig. 15.50), but it does demonstrate many of the options you are likely to need. The repetitive `mtext` function calls that place margin labels could be done with a loop, making it more compact but less clear to beginners unfamiliar with loops.

```
par( mar = c(5, 4, 4, 2) + 0.1 )
par( mfrow = c(1, 1) )
par(family = "serif")
plot(pretest, posttest,
  main = "My Main Title" ,
  xlab = "My xlab text" ,
  ylab = "My ylab text",
  sub  = "My subtitle ",
  pch  = 2)

text(66, 88, "My Example Formula")
text(65, 85,
  expression( hat(beta) ==
  (X^t * X)^{-1} * X^t * Y) )

text(  80, 65, "My label with arrow", pos = 3)
arrows(80, 65, 58.5, 59, length = 0.1)
abline(h = 75, v = 75)
abline(a = 0,  b = 1, lty = 5)
abline(    lm(posttest ~ pretest), lty = 1 )
lines( lowess(posttest ~ pretest), lty = 3 )
legend( 64, 99,
 legend = c("Regression", "Lowess", "Posttest=Pretest"),
 lty    = c(1, 3, 5) )

mtext("line = 0", side = 1, line = 0, at = 57 )
mtext("line = 1", side = 1, line = 1, at = 57 )
mtext("line = 2", side = 1, line = 2, at = 57 )
mtext("line = 3", side = 1, line = 3, at = 57 )
mtext("line = 4", side = 1, line = 4, at = 57 )
mtext("line = 0", side = 2, line = 0, at = 65 )
mtext("line = 1", side = 2, line = 1, at = 65 )
mtext("line = 2", side = 2, line = 2, at = 65 )
mtext("line = 3", side = 2, line = 3, at = 65 )
mtext("line = 0", side = 3, line = 0, at = 65 )
mtext("line = 1", side = 3, line = 1, at = 65 )
mtext("line = 2", side = 3, line = 2, at = 65 )
mtext("line = 3", side = 3, line = 3, at = 65 )
mtext("line = 0", side = 4, line = 0, at = 65 )
mtext("line = 1", side = 4, line = 1, at = 65 )
mtext("line = 0", side = 4, line = 0, at = 65 )
mtext("line = 1", side = 4, line = 1, at = 65 )
```

15.18 Example Traditional Graphics Programs

The SAS and SPSS examples in this chapter are particularly sparse compared to those for R. This is due to space constraints rather than due to lack of capability. The SPSS examples below are done only using their legacy graph-

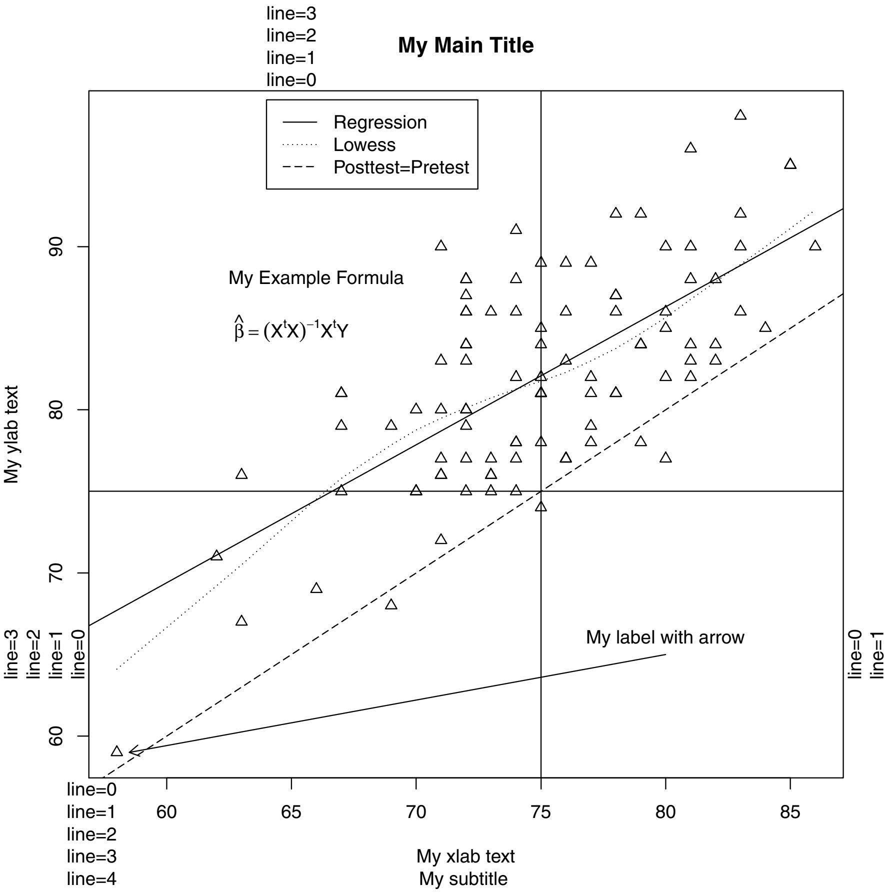


Fig. 15.50. A plot demonstrating many types of text and line annotations. The "line=n" labels around the margins display how the line numbers start at zero next to each axis and move outward as the line numbers increase

ics. I present a parallel set of examples using SPSS's Graphics Production Language in the next chapter.

15.18.1 SAS Program for Traditional Graphics

```
* Filename: GraphicsTraditional.sas ;

LIBNAME myLib 'C:\myRfolder';
OPTIONS _LAST_=myLib.mydata100;

* Histogram of q1;
PROC GCHART; VBAR q1; RUN;

* Bar charts of workshop & gender;
PROC GCHART; VBAR workshop gender; RUN;

* Scatter plot of pretest by posttest;
PROG GPLOT; PLOT posttest*pretest; RUN;

* Scatter plot matrix of all vars but gender;
* Gender would need to be recoded as numeric;

PROC INSIGHT;
SCATTER workshop q1-q4 * workshop q1-q4;
RUN;
```

15.18.2 SPSS Program for Traditional Graphics

```
* Filename: GraphicsTraditional.sps .

CD 'c:\myRfolder'.
GET FILE='mydata100.sav'.

* Legacy SPSS commands for histogram of q1.
GRAPH  /HISTOGRAM=q1 .

* Legacy SPSS commands for bar chart of gender.
GRAPH  /BAR(SIMPLE)=COUNT BY gender .

* Legacy syntax for scatter plot of q1 by q2.
GRAPH   /SCATTERPLOT(BIVAR)=pretest WITH posttest.

* Legacy SPSS commands for scatter plot matrix
* of all but gender. * Gender cannot be used until
* it is recoded numerically.
```

```
GRAPH   /SCATTERPLOT(MATRIX)=workshop q1 q2 q3 q4.

execute.
```

15.18.3 R Program for Traditional Graphics

```
# R Program for Traditional Graphics
# Filename: GraphicsTraditional.R

setwd("c:/myRfolder")
load(file = "mydata100.Rdata")
attach(mydata100)
options(width = 64)

# Request it to ask you to click for new graph.
par(ask = FALSE, mfrow = c(1,1) )

#---The Plot Function---

plot(workshop)            # Bar plot
plot(posttest)            # Index plot
plot(workshop, gender)    # Bar plot split
plot(workshop, posttest)  # Box plot
plot(posttest, workshop)  # Strip plot
plot(pretest, posttest)   # Scatter plot

#---Barplots---

# Barplots of counts via table

barplot( c(40, 60) )

barplot(q4)
table(q4)
barplot( table(q4) )

barplot( workshop )
barplot( table(workshop) )
barplot(gender)
barplot( table(gender) )

barplot( table(workshop), horiz = TRUE)

barplot( as.matrix( table(workshop) ),
  beside = FALSE)

# Grouped barplots & mosaic plots

barplot( table(gender, workshop) )

plot(workshop, gender)

mosaicplot( table(workshop, gender) )

mosaicplot(~ Sex + Age + Survived,
  data = Titanic, color = TRUE)

# Barplots of means via tapply

myMeans <- tapply(q1, gender, mean, na.rm = TRUE)
barplot(myMeans)

myMeans <- tapply(
  q1, list(workshop, gender), mean, na.rm = TRUE)
barplot(myMeans, beside = TRUE)

#---Adding main title, color and legend---

barplot( table(gender, workshop),
  beside = TRUE,
  col    = c("gray90", "gray60"),
  xlab   = "Workshop",
  ylab   = "Count",
  main   = "Number of males and females \nin each workshop" )
legend( "topright",
  legend = c("Female", "Male"),
  fill   = c("gray90", "gray60") )

# A manually positioned legend at 10,15.
legend( 10, 15,
  legend = c("Female", "Male"),
  fill   = c("gray90", "gray60") )

#---Mulitple Graphs on a Page---

par()
head( par() )

opar <- par(no.readonly = TRUE)
```

```
par( mar   = c(3, 3, 3, 1) + 0.1 )
par( mfrow = c(2, 2) )

barplot( table(gender, workshop) )
barplot( table(workshop, gender) )
barplot( table(gender, workshop), beside = TRUE )
barplot( table(workshop, gender), beside = TRUE )

par(opar)  # Reset to original parameters.


#---Piecharts---
pie( table(workshop),
  col  = c("white", "gray90", "gray60", "black" ) )

#---Dotcharts---

dotchart( table(workshop,gender),
  pch=19, cex=1.5)

# ---Histograms---

hist(posttest)

# More bins plus density and ticks at values.
hist(posttest, breaks = 20, probability = TRUE)
lines( density(posttest) )
rug(posttest)

# Histogram of males only.
hist( posttest[ which(gender == "Male") ],
  main = "Histogram of Males Only",
  col = "gray60")

# Plotting above two on one page,
# matching breakpoints.
par(mfrow = c(2, 1) )
hist(posttest, col = "gray90",
  main="Histogram for Both Genders",
  breaks = c(50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100) )
hist(posttest[ which(gender == "Male") ],
  col = "gray60",
  main="Histogram for Males Only",
  breaks  = c(50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100) )
par(mfrow = c(1, 1) )
```

```
# Could have used either of these:
# breaks = seq(from = 50, to = 100, by = 5) )
# breaks = seq(50, 100, 5) )

# Histograms overlaid.

hist( posttest, col = "gray90",
  breaks = seq(from = 50, to = 100, by = 5) )
hist(posttest[ which(gender == "Male") ],
  col = "gray60",
  breaks = seq(from = 50, to = 100, by = 5),
  add = TRUE )
legend( "topright",
  legend = c("Female", "Male"),
  fill   = c("gray90", "gray60") )

# Same plot but extracting $breaks
# from previous graph.

myHistogram <- hist(posttest, col = "gray90")
names(myHistogram)
myHistogram$breaks
myHistogram$xlim
hist(posttest[ which(gender == "Male") ],
  col =  gray60 ,
  add = TRUE, breaks=myHistogram$breaks)
legend( "topright",
  legend = c("Female", "Male"),
  fill   = c("gray90", "gray60") )

# What else does myHistogram hold?
class(myHistogram)
myHistogram

#---Q-Q plots---

library("car")
qq.plot(posttest,
  labels = row.names(mydata100),
  col    = "black" )
detach("package:car")

myQQ <- qqnorm(posttest)  # Not shown in text.
identify(myQQ)
```

```
#---Stripcharts---

par( mar  = c(4, 3, 3, 1) + 0.1 )
par(mfrow = c(2, 1) )
stripchart(posttest, method = "jitter",
  main = "Stripchart with Jitter")
stripchart(posttest, method = "stack",
  main = "Stripchart with Stacking")
par( mfrow = c(1, 1) )
par( mar = c(5, 4, 4, 2) + 0.1 )

par( las = 2, mar = c(4, 8, 4, 1) + 0.1  )
stripchart(posttest ~ workshop, method = "jitter")
par( las = 0, mar = c(5, 4, 4, 2) + 0.1 )

# --- Scatter Plots ---

plot(pretest, posttest)

# Find low score interactively.
# Click 2nd mouse button to choose stop.
identify(pretest, posttest)

# Check it manually.
mydata100[pretest < 60, ]

# Different types of plots.
par( mar   = c(5, 4, 4, 2) + 0.1 )
par( mfrow = c(2, 2) )
plot( pretest, posttest, type = "p", main =  type="p"  )
plot( pretest, posttest, type = "l", main =  type="l"  )
plot( pretest, posttest, type = "b", main =  type="b"  )
plot( pretest, posttest, type = "h", main =  type="h"  )
par( mfrow = c(1, 1) )

# Scatter plots with jitter

par( mar   = c(5, 4, 4, 2) + 0.1 )
par( mfrow = c(1, 2) )
plot( q1, q4,
  main = "Likert Scale Without Jitter")
plot( jitter(q1, 3), jitter(q4, 3),
  main = "Likert Scale With Jitter")
```

```
# Scatter plot of large data sets.

# Example with pch = "." and jitter.
par(mfrow = c(1, 2) )
pretest2   <- round( rnorm( n = 5000, mean = 80, sd = 5) )
posttest2  <-
  round( pretest2 + rnorm( n = 5000, mean = 3, sd = 3) )
pretest2[ pretest2  > 100] <- 100
posttest2[posttest2 > 100] <- 100
plot( pretest2, posttest2,
  main = "5,000 Points, Default Character \nNo Jitter")
plot( jitter(pretest2,4), jitter(posttest2, 4), pch = ".",
  main = "5,000 Points Using pch =  .  \nand Jitter")
par(mfrow = c(1, 1) )

# Hexbins (resets mfrow automatically).
library("hexbin")
plot( hexbin(pretest2, posttest2),
  main = "5,000 Points Using Hexbin")
detach("package:hexbin")

# smoothScatter
smoothScatter(pretest2, posttest2)

rm(pretest2, posttest2) # Cleaning up.

# Scatter plot with different lines added.
plot(posttest ~ pretest)
abline(h = 75, v = 75)
abline(a = 0, b = 1, lty = 5)
abline( lm(posttest ~ pretest),    lty = 1 )
lines( lowess(posttest ~ pretest), lty = 3 )
legend(60, 95,
  legend = c("Regression", "Lowess", "Posttest=Pretest"),
  lty    = c(1, 3, 5) )

# Scatter plot of q1 by q2 separately by gender.
plot(posttest ~ pretest,
  pch = as.numeric(gender) )

abline( lm( posttest[ which(gender == "Male") ]
          ~ pretest[  which(gender == "Male") ] ),
        lty = 1 )

abline( lm( posttest[ which(gender == "Female") ]
          ~ pretest[  which(gender == "Female") ] ),
        lty = 2 )

legend( "topleft",
  legend = c("Male", "Female"),
  lty    = c(1, 2),
  pch    = c(2, 1) )

# Coplots: conditioned scatter plots.
coplot( posttest ~ pretest | workshop)
coplot( posttest ~ pretest | q1)

# Scatter plot with confidence ellipse.
library("car")
data.ellipse(pretest, posttest,
  levels = .95,
  col    = "black")
detach("package:car")

# Confidence Intervals: A small example.
x  <- c(1, 2, 3, 4)
y1 <- c(1, 2, 3, 4)
y2 <- c(2, 3, 4, 5)
y3 <- c(3, 4, 5, 6)
yMatrix <- cbind(y1, y2, y3)
yMatrix

# Just the points.
plot(x, y2, xlim = c(1, 4), ylim = c(1, 6), cex = 1.5)

# Points with pseudo-confidence interval
plot(x, y2, xlim = c(1, 4), ylim = c(1, 6), cex = 1.5)
matlines( x, yMatrix, lty = c(2, 1, 2), col = "black" )
rm( x, y1, y2, y3, yMatrix)

# Confidence Intervals: A realistic example.
myIntervals <-
  data.frame(pretest = seq(from = 60, to = 100, by = 5))
myIntervals
myModel <- lm( posttest ~ pretest )
myIntervals$pp <- predict( myModel,
  interval = "prediction", newdata = myIntervals)
myIntervals$pc <- predict(myModel,
  interval = "confidence", newdata = myIntervals)
myIntervals
```

```
class( myIntervals$pp )
myIntervals$pp

plot( pretest, posttest,
  ylim = range(myIntervals$pretest,
    myIntervals$pp, na.rm = TRUE) )
matlines(myIntervals$pretest, myIntervals$pc,
  lty = c(1, 2, 2), col = "black")
matlines(myIntervals$pretest, myIntervals$pp,
  lty = c(1, 3, 3), col = "black")

# Scatter plot plotting text labels.

plot(pretest, posttest,
  pch = as.character(gender) )

plot(pretest, posttest, type = "n" )
text(pretest, posttest,
  label = row.names(mydata100) )

# Scatter plot matrix of whole data frame.
plot(mydata100[3:8]) #Not shown with text.

plot(mydata100[3:8], gap = 0, cex.labels = 0.9)

pairs(mydata100[3:8], gap = 0,
  lower.panel = panel.smooth,
  upper.panel = panel.smooth)

# Dual axes
# Adds room for label on right margin.
par( mar = c(5, 5, 4, 5) )
plot(pretest, posttest,
  xlab = "Pretest Score",
  ylab = "Posttest Score")
axis(4)
mtext("Posttest Score", side = 4, line = 3)
grid()

#---Boxplots---
plot(workshop, posttest)

par( mfrow = c(2, 2) )
boxplot(posttest)
boxplot(pretest, posttest, notch = TRUE)
boxplot(posttest ~ workshop)
par( las = 2, mar = c(8, 4, 4, 2) + 0.1 )
boxplot(posttest ~ workshop:gender)
par( las = 1, mar = c(5, 4, 4, 2) + 0.1 )

#---Error bar plots---

library("gplots")
par( mfrow = c(1, 1) )
plotmeans( posttest ~ workshop)
detach("package:gplots")

interaction.plot(workshop, gender, posttest)

# ---Adding Labels---

# Many annotations at once.
par( mar = c(5, 4, 4, 2) + 0.1 )
par( mfrow = c(1, 1) )
par(family = "serif")
plot(pretest, posttest,
  main = "My Main Title" ,
  xlab = "My xlab text" ,
  ylab = "My ylab text",
  sub  = "My subtitle ",
  pch  = 2)

text(66, 88, "My Example Formula")
text(65, 85,
  expression( hat(beta) ==
  (X^t * X)^{-1} * X^t * Y) )

text(  80, 65, "My label with arrow", pos = 3)
arrows(80, 65, 58.5, 59, length = 0.1)
abline(h = 75, v = 75)
abline(a = 0,  b = 1, lty = 5)
abline(    lm(posttest ~ pretest), lty = 1 )
lines( lowess(posttest ~ pretest), lty = 3 )
legend( 64, 99,
 legend = c("Regression", "Lowess", "Posttest=Pretest"),
 lty    = c(1, 3, 5) )

mtext("line = 0", side = 1, line = 0, at = 57 )
mtext("line = 1", side = 1, line = 1, at = 57 )
mtext("line = 2", side = 1, line = 2, at = 57 )
mtext("line = 3", side = 1, line = 3, at = 57 )
mtext("line = 4", side = 1, line = 4, at = 57 )
mtext("line = 0", side = 2, line = 0, at = 65 )
mtext("line = 1", side = 2, line = 1, at = 65 )
mtext("line = 2", side = 2, line = 2, at = 65 )
mtext("line = 3", side = 2, line = 3, at = 65 )
mtext("line = 0", side = 3, line = 0, at = 65 )
mtext("line = 1", side = 3, line = 1, at = 65 )
mtext("line = 2", side = 3, line = 2, at = 65 )
mtext("line = 3", side = 3, line = 3, at = 65 )
mtext("line = 0", side = 4, line = 0, at = 65 )
mtext("line = 1", side = 4, line = 1, at = 65 )
mtext("line = 0", side = 4, line = 0, at = 65 )
mtext("line = 1", side = 4, line = 1, at = 65 )

#---Scatter plot with bells & whistles---
#       Not shown in text

plot(pretest, posttest, pch = 19,
  main = "Scatter Plot of Pretest and Postest",
  xlab = "Test score before taking workshop",
  ylab = "Test score after taking workshop" )
myModel <- lm(posttest ~ pretest)
abline(myModel)
arrows(60, 82, 63, 72.5, length = 0.1)
text(60, 82, "Linear Fit", pos  = 3)
arrows(70, 62, 58.5, 59, length = 0.1)
text(70, 62, "Double check this value", pos = 4)
# Use locator() or:
# predict(myModel, data.frame(pretest=75) )
```

16

Graphics with ggplot2

16.1 Introduction

As we discussed in Chap. 14, "Graphics Overview," the ggplot2 package is an implementation of Wilkinson's grammar of graphics (hence the "gg" in its name). The last chapter focused on R's traditional graphics functions. Many plots were easy, but other plots were a lot of work compared to SAS or SPSS. In particular, adding things like legends and confidence intervals was complicated.

The ggplot2 package is usually easier to use, as you will now see as we replicate many of the same graphs. The ggplot2 package has both a shorter qplot function (also called quickplot) and a more powerful ggplot function. We will use both so you can learn the difference and choose whichever you prefer. Although less flexible overall, the built-in lattice package is also capable of doing the examples in this chapter.

While traditional graphics come with R, you will need to install the ggplot2 package. For details, see Chap. 2, "Installing and Updating R." Once installed, you need to load the package using the library function:

```
> library("ggplot2")

Loading required package: grid
Loading required package: reshape
Loading required package: proto
Loading required package: splines
Loading required package: MASS
Loading required package: RColorBrewer
Loading required package: colorspace
```

Notice that this requires the grid package. That is a completely different graphics system than traditional graphics. That means that the par function we used to set graphics parameters, like fonts, in the last chapter does not

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_16, © Springer Science+Business Media, LLC 2011

work with `ggplot2`, nor do any of the base functions that we have covered, including `abline`, `arrows`, `axis`, `box`, `grid`, `lines`, `rug`, and `text`.

16.1.1 Overview of `qplot` and `ggplot`

With the `ggplot2` package, you create your graphs by specifying the following elements:

> Aesthetics: The aesthetics map your data to the graph, telling it what role each variable will play. Some variables will map to an axis, and some will determine the color, shape, or size of a point in a scatter plot. Different groups might have differently shaped or colored points. The size or color of a point might reflect the magnitude of a third variable. Other variables might determine how to fill the bars of a bar chart with colors or patterns; so, for example, you can see the number of males and females within each bar.
>
> Geoms: Short for geometric objects, geoms determine the objects that will represent the data values. Possible geoms include bar, boxplot, errorbar, histogram, jitter, line, path, point, smooth, and text.
>
> Statistics: Statistics provide functions for features like adding regression lines to a scatter plot, or dividing a variable up into bins to form a histogram.
>
> Scales: These match your data to the aesthetic features – for example, in a legend that tells us that triangles represent males and circles represent females.
>
> Coordinate system: For most plots this is the usual rectangular Cartesian coordinate system. However, for pie charts it is the circular polar coordinate system.
>
> Facets: These describe how to repeat your plot for each subgroup, perhaps creating separate scatter plots for males and females. A helpful feature of facets is that they standardize the axes on each plot, making comparisons across groups much easier.

The `qplot` function tries to simplify graph creation by (a) looking a lot like the traditional `plot` function and (b) allowing you to skip specifying as many of the items above as possible. As with the `plot` function, the main arguments to the `qplot` function are the x and y variables. You can identify them with the argument name `x =` or `y =` or you can simply supply them in that order.

The `qplot` function is not generic, but it will give you appropriate plots for factors, vectors and combinations of the two as shown in Fig. 16.1. While `qplot` mimics the `plot` function in some ways, you can see that the two function provide different defaults by comparing `qplot`'s results shown in Fig. 16.1 to those of the `plot` function's back in Fig. 15.1. The two main differences are that the default plot for a numeric vector is a histogram in `qplot` rather

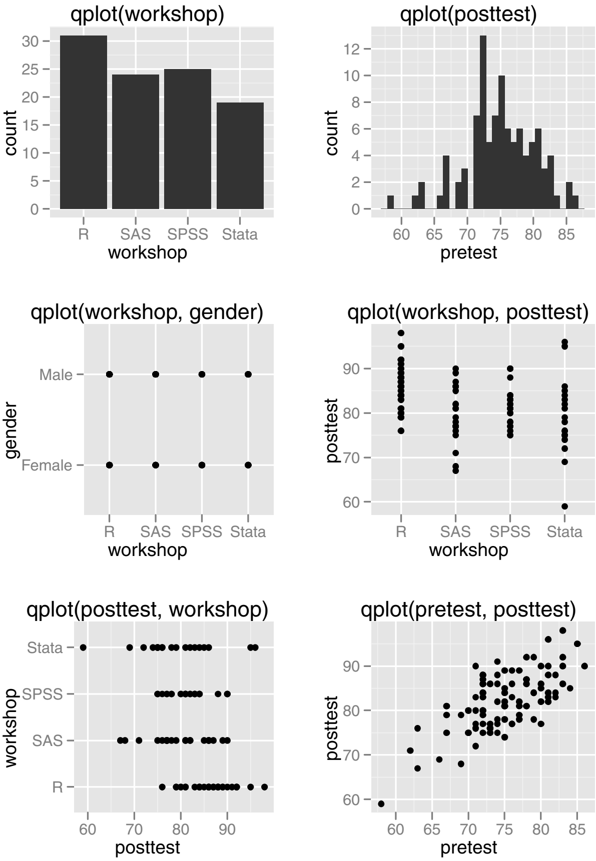


Fig. 16.1. Default plots from `qplot`.

than an index plot, and for two factors `qplot` gives a rather odd plot showing where the combinations of factor values exist rather than a split bar plot.

Unlike the `plot` function, the `qplot` function has a `data` argument. That means you do not have to attach the data frame to use short variable names. (However, to minimize our code, our examples will assume the data are attached.)

Finally, as you would expect, elements are specified by an argument. For example, `geom = "bar"`.

A major difference between `plot` and `qplot` is that `qplot` is limited to working with vectors and factors. So, for example, it will not automatically give you diagnostic plots for a model you have created. As nice as the `ggplot` package is, it does not replace the need for R's traditional graphics.

The ggplot function offers a complete implementation of the grammar of graphics. To do so, it gives up any resemblance to the plot function. It *requires* you to specify the data frame, since you can use different data frames in different layers of the graph. (The qplot function cannot.) Its options are specified by additional *functions* rather than the usual arguments. For example, rather than the geom = "bar" format of qplot, they follow the form +geom_bar(options). The form is quite consistent, so if you know there is a geom named "smooth," you can readily guess how to specify it in either qplot or ggplot.

The simplicity that qplot offers has another limitation. Since it cannot plot in layers, it occasionally needs help interpreting what you want it to do with legends. For example, you could do a scatter plot for which size = q4. This would cause the points to have five sizes, from small for people who did not like the workshop to large for those who did. The qplot function would generate the legend for you automatically. However, what happens when you just want to specify the size of all points with size = 4? It generates a rather useless legend showing one of the points and telling you it represents "4." Whenever you want to tell qplot to *i*nhibit the *i*nterpretation of values, you nest them within the "I()" function: size = I(4). As a mnemonic, think that I() = Inhibit unnecessary legends. The ggplot function does not need the I function since its level of control is fine enough to make your intentions obvious. See Table 16.1 for a summary of the major differences between qplot and ggplot. When possible, I have done the examples in this chapter using both qplot and ggplot. You can decide which you prefer.

Although the ggplot2 is based on *The Grammar of Graphics*, the package differs in several important ways from the syntax described in that book. It depends on R's ability to transform data, so you can use log(x) or any other function within qplot or ggplot. It also uses R's ability to reshape or aggregate data, so the ggplot2 package does not include its own algebra for these steps. Also, the functions in ggplot2 display axes and legends automatically, so there is no "guide" function. For a more detailed comparison, see *ggplot2: Elegant Graphics for Data Analysis* by Wickham [71].

16.1.2 Missing Values

By default, the ggplot2 package will display missing values. This would result in additional bars in bar charts and even entire additional plots when we repeat graphs for each level of a grouping variable. That might be fine in your initial analyses, but you are unlikely to want that in a plot for publication. We will use a version of our data set that has all missing values stripped out with

```
mydata100 <- na.omit(mydata100)
```

See Sect. 10.5, "Missing Values," for other ways to address missing values.

Table 16.1. Comparison of `qplot` and `ggplot` functions

	The `qplot` function	The `ggplot` function
Goal	Designed to mimic the `plot` function.	Designed as a complete grammar of graphics system.
Aesthetics	Like most R functions: `qplot(x= , y= , fill= , color= , shape= ,...)`	You must specify the mapping between each graphical element, even x- and y-axes, and the variable(s): `ggplot(data= , aes(x= , y= , fill= , color= , shape= ,...) ).`
ABline	`...geom="abline", intercept=a, slope=b)`	`+geom_abline( intercept=a, slope=b).`
Aspect ratio	Leave out for interactive adjustment. `+coord_equal(ratio=height/width)` `+coord_equal()` is square	Leave out for interactive adjustment. `+coord_equal(ratio=height/width)` `+coord_equal()` is square.
Axis flip	`+coord_flip()`	`+coord_flip()`

Axis labels	`...xlab="My Text")`
	Just like plot function.
		`+scale_x_discrete("My Text")`
		`+scale_y_discrete("My Text")`
		`+scale_x_continuous("My Text")`
		`+scale_y_continuous("My Text")`

Axis logarithmic	`+scale_x_log10()`
	`+scale_x_log2()`
	`+scale_x_log()`
		`+scale_x_log10()`
		`+scale_x_log2()`
		`+scale_x_log()`

Bars	`...geom="bar", position="stack"` or `dodge`.	`+geom_bar(position="stack")` or `dodge`.
Bar filling	`...posttest, fill=gender)`	`+aes(x=posttest, fill=gender)`
Data	Optional `data=` argument as with most R functions.	You must specify `data=` argument. `ggplot(data = mydata, aes(...`
Facets	`...facets=gender ~ .)`	`+facet_grid( gender ~ . )`
Greyscale	`+scale_fill_grey(start=0, end=1)` Change values to control grey.	`+scale_fill_grey(start=0, end=1)` Change values to control grey.
Histogram	`...geom="histogram", binwidth=1)`	`+geom_bar(binwidth=1)`
Density	`...geom="density")`	`+geom_density()`
Jitter	`..position=position_jitter()` Lessen jitter with, e.g., `(width=.02)`.	`+geom_jitter(position=position_jitter()` Lessen jitter with, e.g., `(width=.02)`.
Legend inhibit	Use `I()` function, e.g., `...geom="point", size = I(4) )`	Precise control makes `I()` function unnecessary.
Line	`...geom="line"`	`+geom_line()`
Line vert.	`...geom="vline", xintercept=?)`	`+geom_vline(xintercept=?)`
Line horiz.	`...geom="hline", yintercept=?)`	`+geom_hline(yintercept=?)`
Pie (polar)	`+coord_polar(theta="y")`	`+coord_polar(theta="y")`
Points	`...geom="point")` That is the default for two variables.	`+geom_point(size=2)` There is no default geom for `ggplot`. The default size is 2.
Polygons (maps)	`...geom="polygon")`	`+geom_polygon()`
QQ plot	`...stat="qq")`	`+stat_qq=()`
Rug	`...geom="rug"`	`+geom_rug()`
Smooth	`...geom="smooth", method="lm")` Lowess is default method.	`+geom_smooth(method="lm")` Lowess is default method.
Smooth no confidence	`...geom="smooth", method="lm", se = FALSE)`	`+geom_smooth(method="lm", se=FALSE)`
Titles	`...main="My Title")` Just like `plot` function.	`+opts(title="My Title" )`

16.1.3 Typographic Conventions

Throughout this book we have displayed R's prompt characters only when input was followed by output. The prompt characters helped us discriminate between the two. Each of our function calls will result in a graph, so there is no chance of confusing input with output. Therefore, we will dispense with the prompt characters for the remainder of this chapter. This will make the

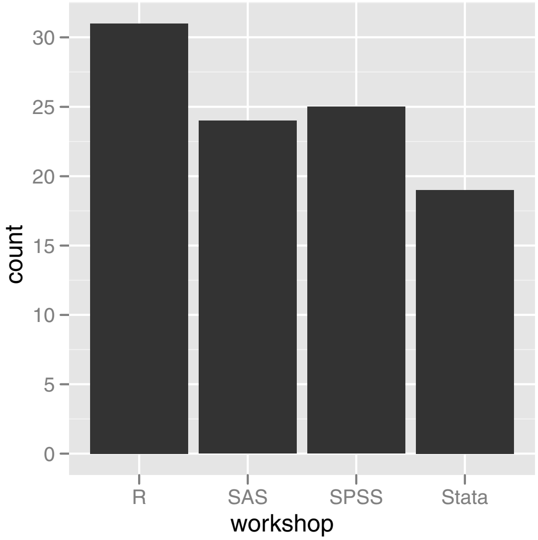


Fig. 16.2. A bar plot of workshop attendance.

code much cleaner to read because our examples of the `ggplot` function often end in a "+" sign. That is something you type. Since R prompts you with "+" at the beginning of a continued line, it looks a bit confusing at first.

16.2 Bar Plots

Let us do a simple bar chart of counts for our workshop variable (Fig. 16.2). Both of the following function calls will do it.

The `qplot` approach to Fig. 16.2 is

```
> attach(mydata100)  # Assumed for all qplot examples

> qplot(workshop)
```

The `ggplot` approach to to Fig. 16.2 is

```
> ggplot(mydata100, aes(workshop) ) +
+   geom_bar()
```

Bars are the default geom when you give `qplot` only one variable, so we only need a single argument, workshop.

The `ggplot` function call above requires three arguments:

1. Unlike most other R functions, it requires that you specify the data frame. As we will see later, that is because `ggplot` can plot multiple layers, and each layer can use a different data frame.
2. The `aes` function defines the *aes*thetic role that workshop will play. It maps workshop to the x-axis. We could have named the argument as in `aes(x = workshop)`. The first two arguments to the `aes` function are `x` and `y`, in that order. To simplify the code, we will not bother listing their names.
3. The `geom_bar` function tells it that the geometric object, or geom, needed is a bar. Therefore, a bar chart will result. This function call is tied to the first one through the "+" sign.

We did the same plot using the traditional graphics `barplot` function, but that required us to summarize the data using `table(workshop)`. The `ggplot2` package is more like SAS and SPSS in this regard (and more like `plot`); it does that type of summarization for you.

If we want to change to a horizontal bar chart (Fig. 16.3), all we need to do is flip the coordinates. In the following examples, it is clear that we simply added the `cord_flip` function to the end of both `qplot` and `ggplot`. There is no argument for `qplot` like `coord = "flip"`.

This brings up an interesting point. Both `qplot` and `ggplot` create the exact same graphics object. Even if there is a `qplot` equivalent, you can always add a `ggplot` function to the `qplot` function.

The `qplot` approach to Fig. 16.3 is

```
qplot(workshop) + coord_flip()
```

The `ggplot` approach to Fig. 16.3 is

```
ggplot(mydata100, aes(workshop) ) +
  geom_bar() + coord_flip()
```

You can create the usual types of grouped bar plots. Let us start with a simple stacked one (Fig. 16.4). You can use either function below. They contain two new arguments. Although we are requesting only a single bar, we must still supply a variable for the x-axis. The function call `factor("")` provides the variable we need, and it is simply an unnamed factor whose value is empty. We use the `factor` function to keep it from labeling the x-axis from 0 to 1, which it would do if the variable were continuous. The `fill = workshop` aesthetic argument tells the function to fill the bars with the number of students who took each workshop.

With `qplot`, we clear labels on the x-axis with `xlab = ""`. Otherwise, the word "factor" would occur there from our `factor("")` statement. The equivalent way to label `ggplot` is to use the `scale_x_discrete` function, also providing an empty label for the x-axis. Finally, the function `scale_fill_grey` tells each function to use shades of grey. You can leave this out, of course, and both functions will choose the same nice color scheme. The start and end

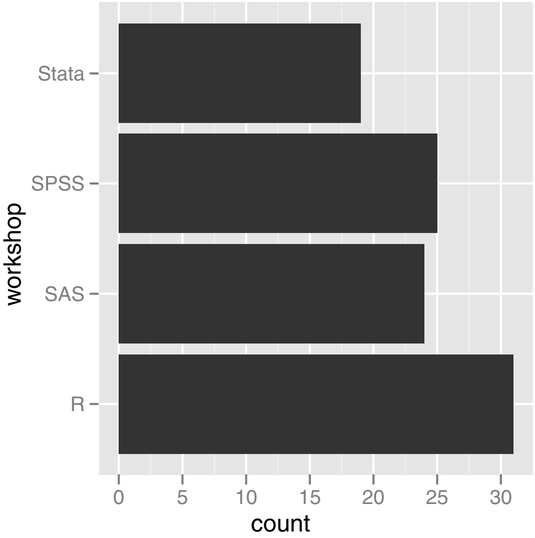


Fig. 16.3. A horizontal bar plot demonstrating the impact of the `coord_flip` function

values tell the function to go all the way to black and white, respectively. If you use just `scale_fill_grey()`, it will use four shades of grey.

The `qplot` approach to Fig. 16.4 is

```
qplot(factor(""), fill = workshop,
  geom = "bar", xlab = "") +
  scale_fill_grey(start = 0, end = 1)
```

The `ggplot` approach to Fig. 16.4 is

```
ggplot(mydata100,
  aes(factor(""), fill = workshop) ) +
  geom_bar() +
  scale_x_discrete("") +
  scale_fill_grey(start = 0, end = 1)
```

16.3 Pie Charts

One interesting aspect to the grammar of graphics concept is that a pie chart (Fig. 16.5) is just a single stacked bar chart (Fig. 16.4) drawn in polar coordinates. So we can use the same function calls that we used for the bar chart

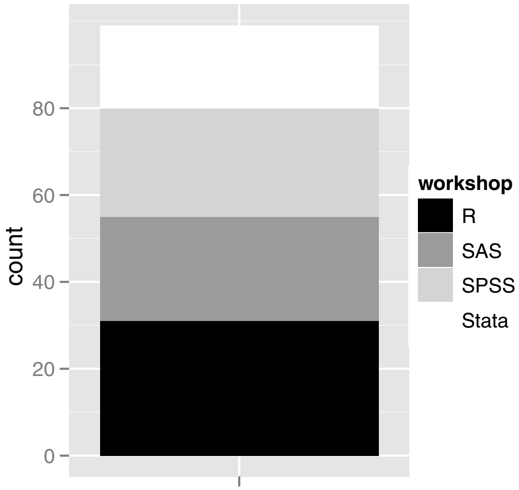


Fig. 16.4. A stacked bar plot of workshop attendance

in the previous section but convert to polar afterward using the `coord_polar` function.

This is a plot that only the `ggplot` function can do correctly. The `geom_bar(width = 1)` function call tells it to put the slices right next to each other. If you included that on a standard bar chart, it would also put the bars right next to each other:

```
ggplot(mydata100,
  aes( factor(""), fill = workshop ) ) +
  geom_bar(width = 1) +
  scale_x_discrete("") +
  coord_polar(theta = "y") +
  scale_fill_grey(start = 0, end = 1)
```

That is a lot of code for a simple pie chart! In the previous chapter, we created this graph with a simple

```
pie( table(workshop) )
```

So traditional graphics are the better approach in some cases. However, as we will see in the coming sections, the `ggplot2` package is the easiest to use for most things.

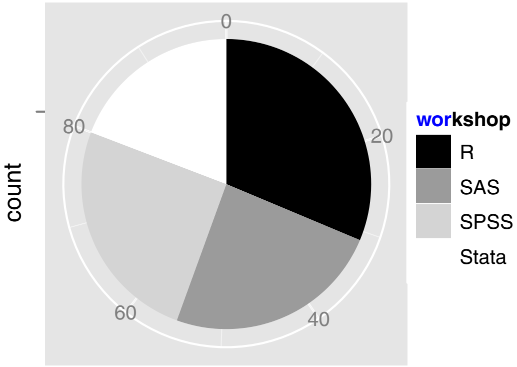


Fig. 16.5. A pie chart of workshop attendance

16.4 Bar Plots for Groups

Let us now look at repeating bar charts for levels of a factor, like gender. This requires having factors named for both the `x` argument and the `fill` argument. Unless you change it, the `position` argument stacks the `fill` groups – in this case, the workshops. That graph is displayed in the upper left frame of Fig. 16.6:

The `qplot` approach to Fig. 16.6, upper left is

```
qplot(gender, geom = "bar",
  fill = workshop, position = "stack") +
  scale_fill_grey(start = 0, end = 1)
```

The `ggplot` approach to Fig. 16.6, upper left is

```
ggplot(mydata100, aes(gender, fill = workshop) ) +
  geom_bar(position = "stack") +
  scale_fill_grey(start = 0, end = 1)
```

Changing either of the above examples to: `position = "fill"` makes every bar fill the y-axis, displaying the proportion in each group rather than the number. That type of graph is called a *spine plot* and it is displayed in the upper right of Fig. 16.6.

Finally, if you set `position = "dodge"`, the filled segments appear beside one another, "dodging" each other. That takes more room on the x-axis, so

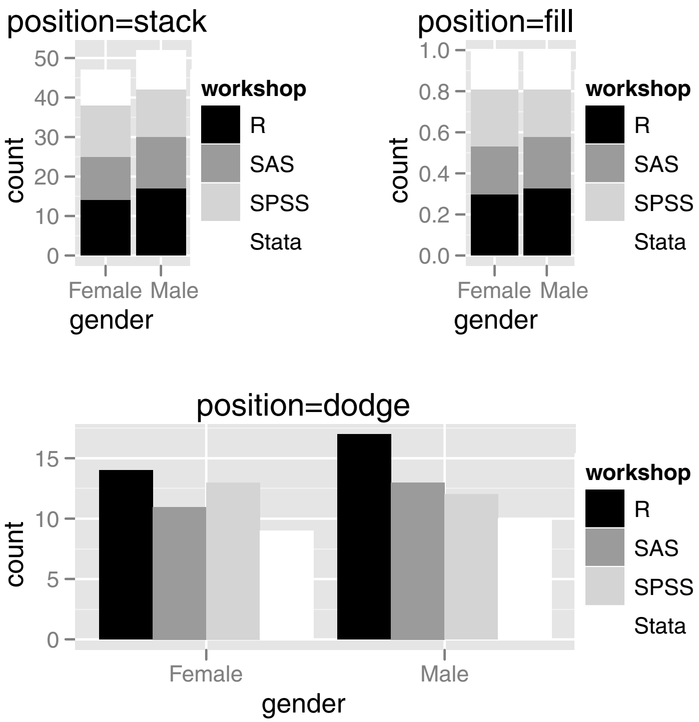


Fig. 16.6. A multiframe plot showing the impact of the various position settings

it appears across the entire bottom row of Fig. 16.6. We will discuss how to convey similar information using multiframe plots in Sect. 16.18, "Multiple Plots on a Page."

16.5 Plots by Group or Level

One of the nicest features of the `ggplot2` package is its ability to easily plot groups within a single plot (Fig. 16.7). To fully appreciate all of the work it is doing for us, let us first consider how to do this with traditional graphics functions.

1. We would set up a multiframe plot, say for males and females.
2. Then we might create a bar chart on workshop, selecting `which(gender == "Female")`.
3. Then we would repeat the step above, selecting the males.
4. We would probably want to standardize the axes to enhance comparisons and do the plots again.

5. We would add a legend, making sure to manually match any color or symbol differences across the plots.
6. Finally, we would turn off the multiframe plot settings to get back to one plot per page.

Thank goodness the ggplot2 package can perform the equivalent of those tedious steps using either of the following simple function calls:

The qplot approach to Fig. 16.7 is

```
qplot(workshop, facets = gender ~ . )
```

The ggplot approach to Fig. 16.7 is

```
ggplot(mydata100, aes(workshop) ) +
  geom_bar() + facet_grid( gender ~ . )
```

The new feature is the facets argument in qplot and the facet_grid function in ggplot. The formula it uses is in the form "rows~columns." In this case, we have "gender~." so we will get rows of plots for each gender and no columns. The "." represents "1" row or column. If we instead did ".~gender," we would have one row and two columns of plots side by side.

You can extend this idea with the various rules for formulas described in Sect. 5.7.3, "Controlling Functions with Formulas." Given the space constraints, the most you are likely to find useful is the addition of one more variable, such as

```
facets = workshop ~ gender
```

In our current example, that leaves us nothing to plot, but we will look at a scatter plot example of that later.

16.6 Presummarized Data

We mentioned earlier that the ggplot2 package assumed that your data needed summarizing, which is the opposite of some traditional R graphics functions. However, what if the data are already summarized? The qplot function makes it quite easy to deal with, as you can see in the program below. We simply use the factor function to provide the x argument and the c function to provide the data for the y argument. Since we are providing both x and y arguments, the qplot function will provide a default point geom, so we override that with geom = "bar". The xlab and ylab arguments *lab*el the axes, which it would otherwise label with the factor and c functions themselves.

The qplot approach to Fig. 16.8 is

```
qplot( factor( c(1, 2) ), c(40, 60), geom = "bar",
  xlab = "myGroup", ylab = "myMeasure")
```

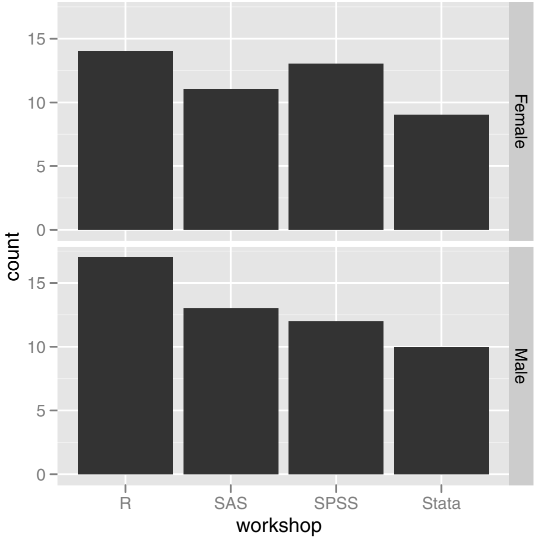


Fig. 16.7. A bar plot of workshop attendance with facets for the genders

The ggplot approach to this type of plot is somewhat different because it requires that the data be in a data frame. I find it much easier to create a temporary data frame containing the summary data. Trying to nest a data frame creation within the ggplot function will work, but you end up with so many parentheses that it can be a challenge getting it to work. The example program at the end of this chapter contains that example as well.

The following is the more complicated ggplot approach to Fig. 16.8. We are displaying R's prompts here to differentiate input from output:

```
> myTemp <- data.frame(
+   myGroup   = factor( c(1, 2) ),
+   myMeasure = c(40, 60)
+ )

> myTemp

  myGroup myMeasure
1       1        40
2       2        60


> ggplot(data = myTemp, aes(myX, myY) ) +
+   geom_bar()
```

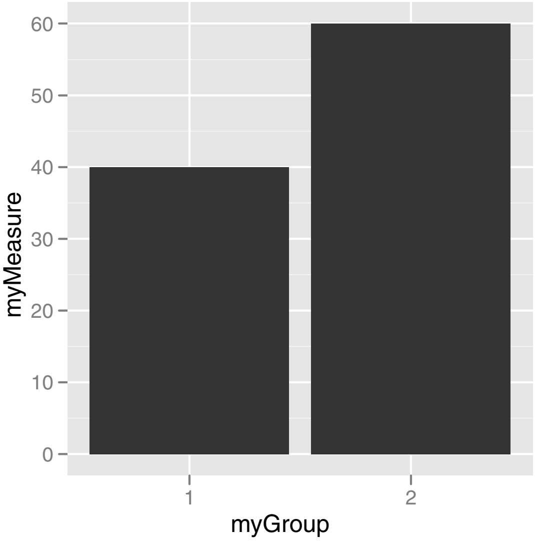


Fig. 16.8. A bar plot of presummarized data

```
> rm(myTemp) #Cleaning up.
```

16.7 Dot Charts

Dot charts are bar charts reduced to just points on lines, so you can take any of the above bar chart examples and turn them into dot charts (Fig. 16.9).

Dot charts are particularly good at packing in a lot of information on a single plot, so let us look at the counts for the attendance in each workshop, for both males and females. This example demonstrates how very different `qplot` and `ggplot` can be. It also shows how flexible `ggplot` is and that it is sometimes much easier to understand than `qplot`.

First, let us look at how `qplot` does it. The variable workshop is in the `x` position, so this is the same as saying `x = workshop`. If you look at the plot, workshop is on the y-axis. However, `qplot` requires an `x` variable, so we cannot simply say `y = workshop` and not specify an `x` variable. Next, it specifies `geom = "point"` and sets the size of the points to `I(4)`, which is much larger than in a standard scatter plot. Remember that the `I()` function around the 4 *i*nhibits *i*nterpretation, which in this case means that it stops `qplot` from displaying a legend showing which point size represents a "4." In this example, that is useless information. You can try various size values to see how it looks. The `stat = "bin"` argument tells it to combine all of the values that it finds for each level of workshop as a histogram might do. So it ends up counting the number of observations in each combination of workshop and gender. The facets argument tells it to create a row for each gender. The coord_flip function rotates it in the direction we desire.

The qplot approach to Fig. 16.9 is

```
qplot(workshop, geom = "point", size = I(4),
  stat = "bin", facets = gender ~ . ) +
 coord_flip()
```

Now let us see how ggplot does the same plot. The aes function supplies the x-axis variable, and the y-axis variable uses the special "..count.." computed variable. That variable is also used by qplot, but it is the default y variable. The geom_point function adds points, bins them, and sets their size. The coord_flip function then reverses the axes. Finally, the facet_grid function specifies the same formula used earlier in qplot. Notice here that we did not need the I() function, as ggplot "knows" that the legend is not needed. If we were adjusting the point sizes based on a third variable, we would have to specify the variable as an additional aesthetic. The syntax to ggplot is verbose, but more precise:

```
ggplot(mydata100,
   aes(workshop, ..count..) ) +
   geom_point(stat = "bin", size = 4) + coord_flip()+
   facet_grid( gender ~ . )
```

16.8 Adding Titles and Labels

Sprucing up your graphs with titles and labels is easy to do (Fig. 16.10). The qplot function adds them exactly like the traditional graphics functions do. You supply the main title with the main argument, and the x and y labels with xlab and ylab, respectively. There is no subtitle argument. As with all labels in R, the characters "\n" causes it to go to a new line, so "\nWorkshops" below will put just the word "Workshops" at the beginning of a new line.

The qplot approach to Fig. 16.10 is

```
qplot(workshop, geom = "bar",
  main = "Workshop Attendance",
  xlab = "Statistics Package \nWorkshops")
```

The ggplot approach to Fig. 16.10 is

```
ggplot(mydata100, aes(workshop, ..count..) ) +
  geom_bar() +
  opts( title = "Workshop Attendance" ) +
  scale_x_discrete("Statistics Package \nWorkshops")
```

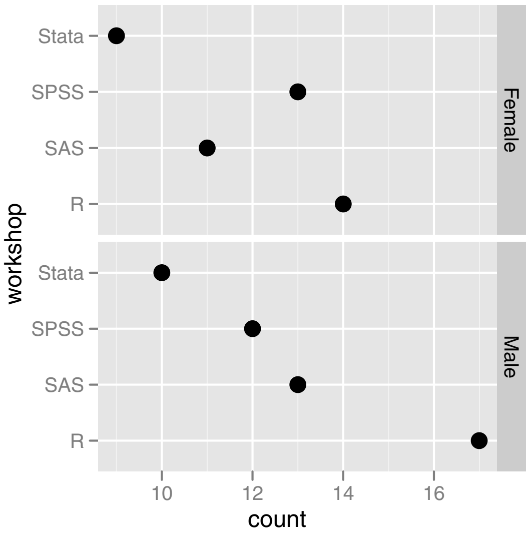


Fig. 16.9. A dot chart of workshop attendance with facets for the genders

Adding titles and labels in `ggplot` is slightly more verbose. The `opts` function sets various *opt*ions, one of which is `title`. The axis labels are attributes of the axes themselves. They are controlled by the functions, `scale_x_discrete` and `scale_y_discrete`, and for continuous axes, they are controlled by the functions `scale_x_continuous` and `scale_y_continuous`, which are clearly named according to their function. I find it odd that you use different functions for labeling axes if they were discrete or continuous, but it is one of the tradeoffs you make when getting all of the flexibility that `ggplot` offers.

16.9 Histograms and Density Plots

Many statistical methods make assumptions about the distribution of your data, or at least of your model residuals. Histograms and density plots are two effective plots to help you assess the distributions of your data.

16.9.1 Histograms

As long as you have 30 or more observations, histograms (Fig. 16.11) are a good way to examine continuous variables. You can use either of the following examples to create one. In `qplot`, the histogram is the default geom for continuous data, making it particularly easy to perform.

The `qplot` approach to Fig. 16.11 is

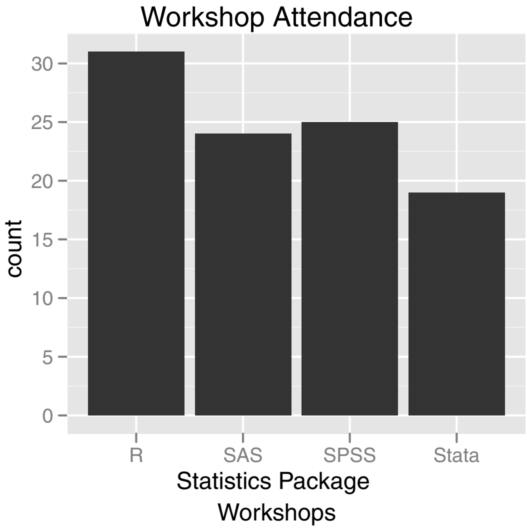


Fig. 16.10. A bar plot demonstrating titles and x-axis labels

```
qplot(posttest)
```

The `ggplot` approach to Fig. 16.11 is

```
ggplot(mydata100, aes(posttest) ) +
  geom_histogram()
```

Both functions, by default, will print to the R console the number of bins they use (not shown). If you narrow the width of the bins, you will get more bars, showing more structure in the data (Fig. 16.12). If you prefer `qplot`, simply add the `binwidth` argument. If you prefer `ggplot`, add the `geom_bar` function with its `binwidth` argument. Smaller numbers result in more bars.

The `qplot` approach to Fig. 16.12 is

```
qplot(posttest, geom = "histogram", binwidth = 0.5)
```

The `qplot` approach to Fig. 16.12 is

```
ggplot(mydata100, aes(posttest) ) +
  geom_bar( binwidth = 0.5 )
```

16.9.2 Density Plots

If you prefer to see a density curve, just change the `geom` argument or function to `density` (Fig. 16.13). The `qplot` approach to Fig. 16.13 is

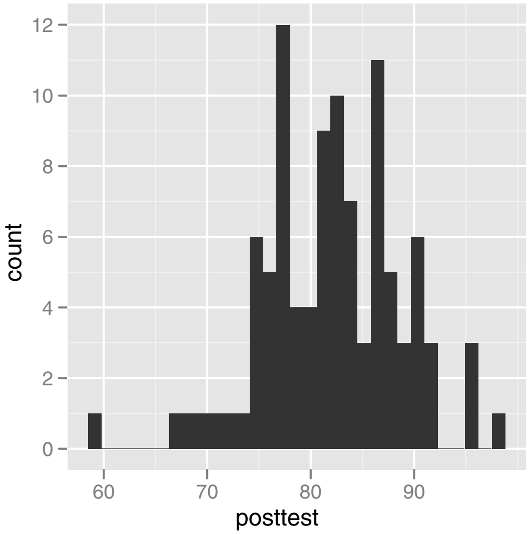


Fig. 16.11. A histogram of posttest

```
qplot(posttest, geom = "density" )
```

The `ggplot` approach to Fig. 16.13 is

```
ggplot(mydata100, aes(posttest) ) +
  geom_density()
```

16.9.3 Histograms with Density Overlaid

Overlaying the density on the histogram, as in Fig. 16.14, is only slightly more complicated. Both `qplot` and `ggplot` compute in the background a special new variable for the y-axis named "`..density..`". It did this behind the scenes in the previous density plot. To get both plots combined, you must use the variable directly. To ask for both a histogram and the density, you must explicitly list `..density..` as the `y` argument. Then for `qplot`, you provide both `histogram` and `density` to the `geom` argument. For `ggplot`, you simply call both functions.

The `qplot` approach to Fig. 16.14 (except for the "rug" points on the x-axis) is

```
qplot(posttest, ..density..,
  geom = c("histogram", "density") )
```

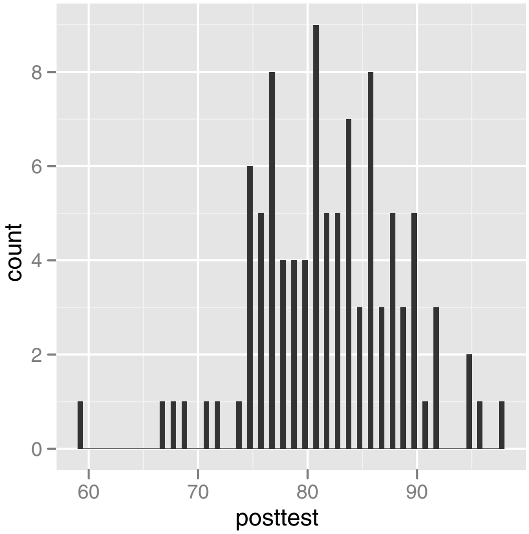


Fig. 16.12. A histogram of posttest with smaller bin widths

Since I added a rug of points to this type of plot using traditional graphics in Sect. 15.19, adding one here is a good example of something that `ggplot` can do that `qplot` cannot. You can add rug points on both the x- and y-axis using `geom = "rug"` in `qplot`. However, we want to add it only to the x-axis, and `ggplot` has that level of flexibility. In the `ggplot` code below for Fig. 16.14, I call `geom_histogram` and `geom_density` with variables for both axes, but for `geom_rug` I give it only the x-axis variable posttest:

```
ggplot(data = mydata100) +
  geom_histogram( aes(posttest, ..density..) ) +
  geom_density(   aes(posttest, ..density..) ) +
  geom_rug( aes(posttest) )
```

16.9.4 Histograms for Groups, Stacked

What if we want to compare the histograms for males and females (Fig. 16.15)? Using base graphics, we had to set up a multiframe plot and learn how to control break points for the bars so that they would be comparable. Using `ggplot2`, the facet feature makes the job trivial.

The `qplot` approach to Fig. 16.15 is

```
qplot(posttest, facets = gender ~ .)
```

The `ggplot` approach to Fig. 16.15 is

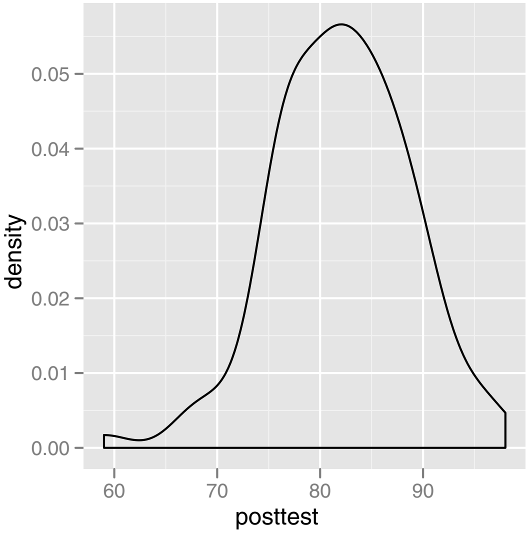


Fig. 16.13. A density plot of posttest

```
ggplot(mydata100, aes(posttest) ) +
  geom_histogram() + facet_grid( gender ~ . )
```

16.9.5 Histograms for Groups, Overlaid

We can also compare males and females by filling the bars by gender as in Fig. 16.16. As earlier, if you leave off the scale_fill_grey function, the bars will come out in two colors rather than black and white.

The qplot approach to Fig. 16.16 is

```
qplot( posttest, fill = gender ) +
  scale_fill_grey(start = 0, end = 1)
```

The ggplot approach to Fig. 16.16 is

```
ggplot(mydata100, aes(posttest, fill = gender) ) +
  geom_bar() + scale_fill_grey( start = 0, end = 1 )
```

16.10 Normal QQ Plots

Earlier I defined a QQ plot in the chapter on traditional graphics. Creating them in the ggplot2 package is straightforward (Fig. 16.17). If you prefer the

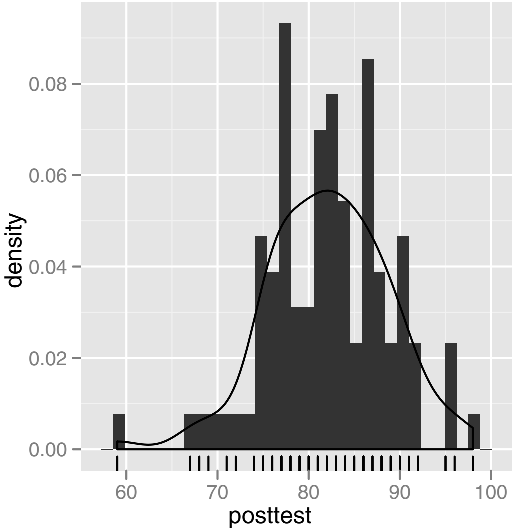


Fig. 16.14. A density plot overlaid on a histogram with rug points on the x-axis

`qplot` function, supply the `stat = "qq"` argument. In `ggplot`, the similar `stat_qq` function will do the trick.

The `qplot` approach to Fig. 16.17 is

```
qplot(posttest, stat = "qq")
```

The `ggplot` approach to Fig. 16.17 is

```
ggplot( mydata100, aes(posttest) ) +
  stat_qq()
```

16.11 Strip Plots

Strip plots are scatter plots of single continuous variables, or a continuous variable displayed at each level of a factor like workshop. As with the single stacked bar chart, the case of a single strip plot still requires a variable on the x-axis (Fig. 16.18). As you see can below, `factor("")` will suffice. The variable to actually plot is the `y` argument. Reversing the x and y variables will turn the plot on its side; in the default way the traditional graphics function, `stripchart`, does it. We prefer the vertical approach, as it matches the style of box plots and error bar plots when you use them to compare groups. The `geom = "jitter"` adds some noise to separate points that would otherwise obscure other points by plotting on top of them. Finally, the `xlab = ""`

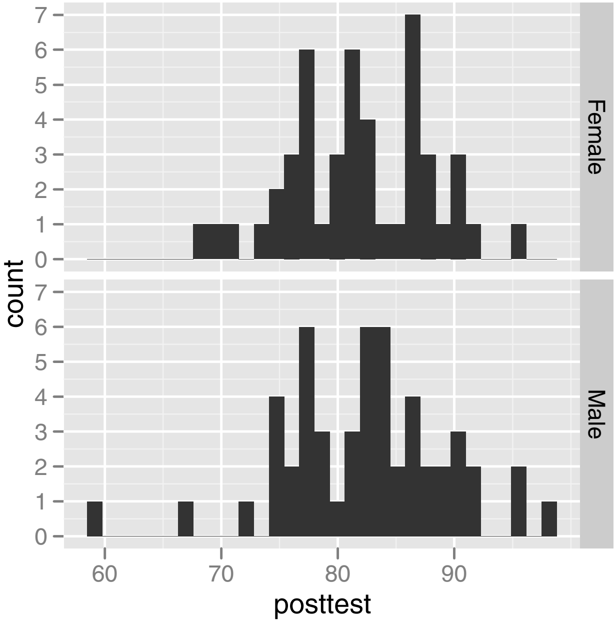


Fig. 16.15. Histograms of posttest with facets for the genders

and `scale_x_discrete("")` labels erase what would have been a meaningless label about `factor("")` for `qplot` and `ggplot`, respectively.

This `qplot` approach does a strip plot with wider jitter than Fig 16.18:

```
qplot( factor(""), posttest, geom = "jitter", xlab = "")
```

This `ggplot` approach does a strip plot with wider jitter than Fig. 16.18:

```
ggplot(mydata100, aes(factor(""), posttest) ) +
  geom_jitter() +
  scale_x_discrete("")
```

The above two examples use an amount of jitter that is best for large data sets. For smaller data sets, it is best to limit the amount of jitter to separate the groups into clear strips of points. Unfortunately, this complicates the syntax.

The `qplot` function controls jitter width with the `position` argument, setting `position_jitter` with `width = scalefactor`.

The `ggplot` approach places that same parameter within its `geom_jitter` function call.

The `qplot` approach to Fig. 16.18 is

```
qplot(factor(""), posttest, data = mydata100, xlab = "",
  position = position_jitter(width = .02) )
```

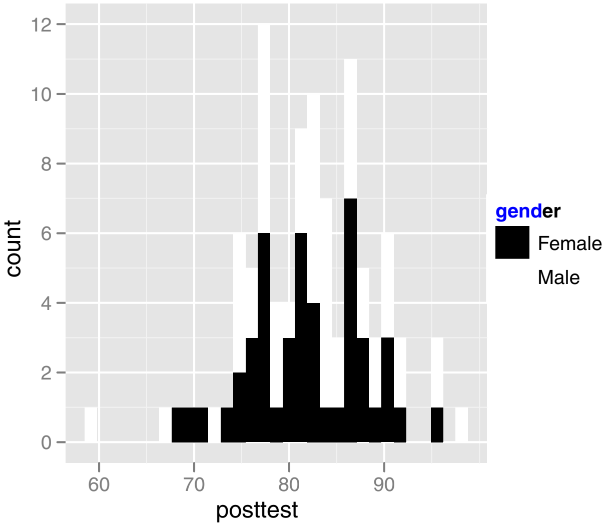


Fig. 16.16. A histogram with bars filled by gender

The ggplot approach to Fig. 16.18 is

```
ggplot(mydata100, aes(factor(""), posttest) ) +
 geom_jitter(position = position_jitter(width = .02) ) +
 scale_x_discrete("")
```

Placing a factor like workshop on the x-axis will result in a strip chart for each level of the factor (Fig. 16.19).

This qplot approach does a grouped strip plot with wider jitter than Fig. 16.19, but its code is simpler:

```
> qplot(workshop, posttest, geom = "jitter")
```

This ggplot approach does a grouped strip plot with wider jitter than Fig. 16.19, but with simpler code:

```
> ggplot(mydata100, aes(workshop, posttest ) ) +
+   geom_jitter()
```

Limiting the amount of jitter for a grouped strip plot uses exactly the same parameters we used for a single strip plot.

The qplot approach to Fig. 16.19 is

```
qplot(workshop, posttest, data = mydata100, xlab = "",
  position = position_jitter(width = .08))
```

The ggplot approach to Fig. 16.19 is

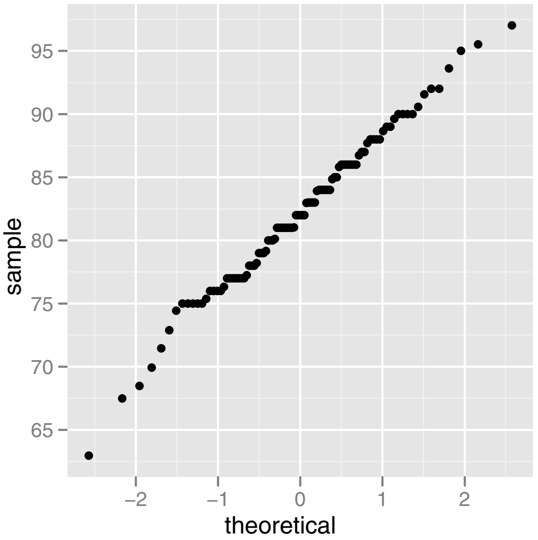


Fig. 16.17. A normal quantile–quantile plot of posttest

```
ggplot(mydata100, aes(workshop, posttest) ) +
 geom_jitter(position = position_jitter(width = .08) ) +
 scale_x_discrete("")
```

16.12 Scatter Plots and Line Plots

The simplest scatter plot hardly takes any effort in `qplot`. Just list `x` and `y` variables in that order. You could add the `geom = "point"` argument, but it is the default when you list only those first two arguments.

The `ggplot` function is slightly more complicated. Since it has no default geometric object to display, we must specify `geom_point()`.

The `qplot` approach to Fig. 16.20, upper left, is

```
qplot(pretest, posttest)
```

The `ggplot` approach to Fig. 16.20, upper left, is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point()
```

We can connect the points using the line geom, as you see below. However, the result is different from what you get in traditional R graphics. The line

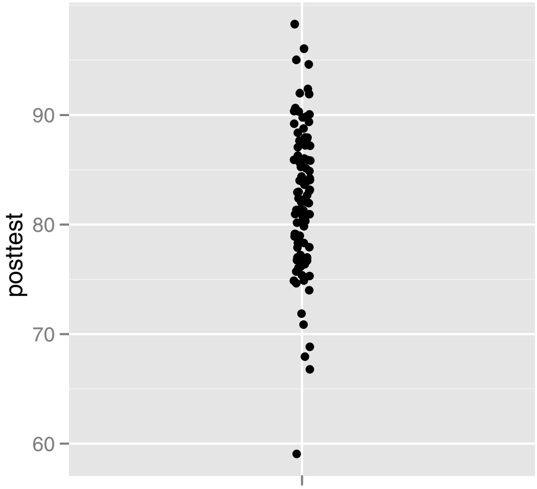


Fig. 16.18. A strip chart done using the `jitter` geom

connects the points in the order in which they appear on the x-axis. That almost makes our data appear as a time series, when they are not.

The `qplot` approach to Fig. 16.20, upper right, is

```
qplot( pretest, posttest, geom = "line")
```

The `qplot` approach to Fig. 16.20, upper right, is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_line()
```

Although the line geom ignored the order of the points in the data frame, the path geom will connect them in that order. You can see the result in the lower left quadrant of Fig. 16.20. The order of the points in our data set has no meaning, so it is just a mess! The following is the code to do it in `qplot` and `ggplot`.

The `qplot` approach to Fig. 16.20, lower left, is

```
qplot( pretest, posttest, geom = "path")
```

The `ggplot` approach to Fig. 16.20, lower left, is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_path()
```

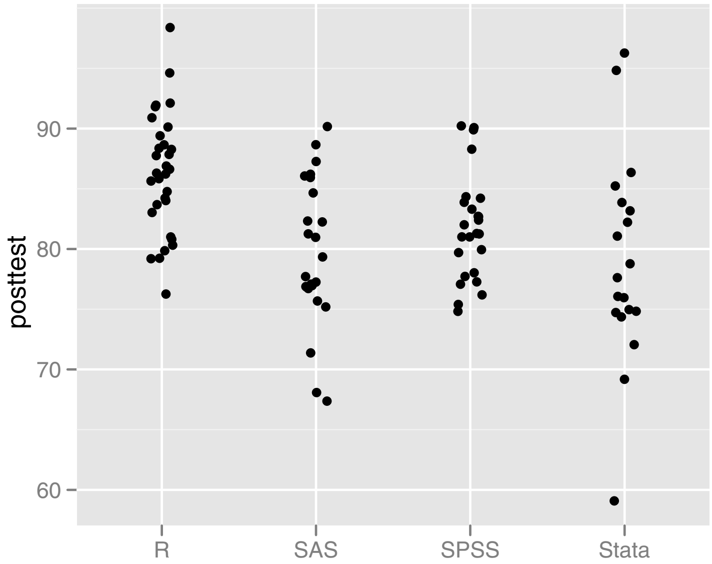


Fig. 16.19. A strip chart with facets for the workshops

Now let us run a vertical line to each point. When we did that using traditional graphics, it was a very minor variation. In ggplot2, it is quite different but an interesting example. It is a plot that is much more clear using ggplot, so we will skip qplot for this one.

In the ggplot code below, the first line is the same as the above examples. Where it gets interesting is the geom_segment function. It has its own aes function, repeating the x and y arguments, but in this case, they are the beginning points for drawing line segments! It also has the arguments xend and yend, which tell it where to end the line segments. This may look overly complicated compared to the simple "type = h" argument from the plot function, but you could use this approach to draw all kinds of line segments. You could easily draw them coming from the top or either side, or even among sets of points. The "type = h" approach is a one-trick pony. With that approach, adding features to a function leads to a very large number of options, and the developer is still unlikely to think of all of the interesting variations in advance.

The following is the code, and the resulting plot is in the lower right panel of Fig. 16.20.

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_segment(   aes(pretest, posttest,
```

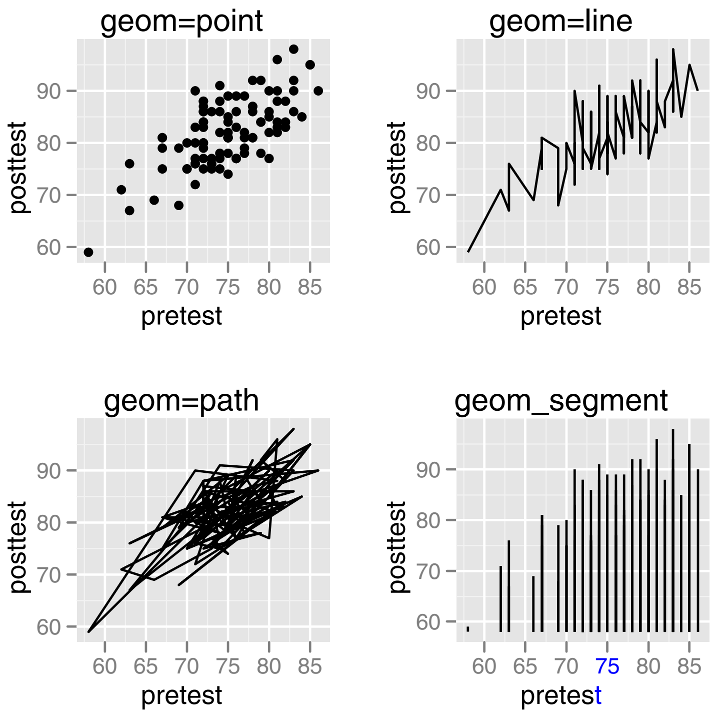


Fig. 16.20. A multiframe plot demonstrating various styles of scatter plots and line plots. The top two and the bottom left show different geoms. The bottom right is done a very different way, by drawing line segments from each point to the x-axis

```
                    xend = pretest, yend = 58) )
```

16.12.1 Scatter Plots with Jitter

We discussed the benefits of jitter in the previous chapter. To get a nonjittered plot of q1 and q4, we will just use `qplot` (Fig. 16.21, left).

```
qplot(q1,q4)
```

To add jitter, below are both the `qplot` and `gglot` approaches (Fig. 16.21, right). Note that the `geom = "point"` argument is the default in `qplot` when two variables are used. Since that default is not shown, the fact that the `position` argument applies to it is not obvious. That relationship is clearer in the `ggplot` code, where the `position` argument is clearly part of the `geom_point` function. You can try various amounts of jitter to see which provides the best view of your data.

The `qplot` approach to Fig. 16.21, right, is

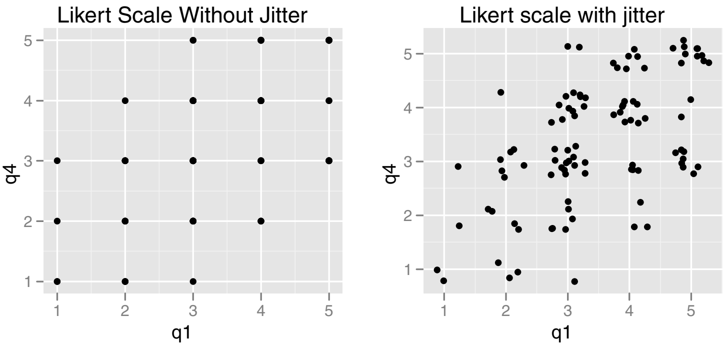


Fig. 16.21. A multiframe plot showing the impact of jitter on five-point Likert–scale data. The plot on the left is not jittered, so many points are obscured. The plot on the right is jittered, randomly moving points out from behind one another

```
qplot(q1, q4, position =
  position_jitter(width = .3, height = .3) )
```

The ggplot approach to Fig. 16.21, right, is

```
ggplot(mydata100, aes(x = q1, y = q2) ) +
 geom_point(position =
   position_jitter(width = .3, height = .3) )
```

16.12.2 Scatter Plots for Large Data Sets

When plotting large data sets, points often obscure one another. The ggplot2 package offers several ways to deal with this problem, including decreasing point size, adding jitter or transparency, displaying density contours, and replacing sets of points with hexagonal bins.

Scatter Plots with Jitter and Transparency

By adjusting the amount of jitter and the amount of transparency, you can find a good combination that lets you see through points into the heart of a dense scatter plot (Fig. 16.22). For example, let us consider three points occupied the exact same location. In a small data set, adding jitter will move them about, perhaps plotting them with no overlap at all. You can then see clearly the number of plots near that location. But with a large data set, the scatter of points is dense and the jitter may just move those points on top of other points nearby. When you make points semi-transparent, the more points overlap, the less transparent that plot location becomes. This ends up making dense parts of the plot darker, showing structure that exists deeper into the point cloud.

Unfortunately, transparency is not yet supported in Windows metafiles. So if you are a Windows user, choose "Copy as bitmap" when cutting and pasting graphs into your word processor. For a higher-resolution image, route your graph to a file using the `png` function. For an example, see Sect. 14.10, "Graphics Devices." You can also use the `ggsave` function, which is part of the `ggplot2` package. For details, see Sect. 16.19, "Saving `ggplot2` Graphs to a File."

To get 5000 points to work with, we generated them with the following code:

```
pretest2  <- round( rnorm(n = 5000, mean = 80, sd = 5) )

posttest2 <- round( pretest2 +
  rnorm(n = 5000, mean = 3, sd = 3) )

pretest2[pretest2 > 100] <- 100

posttest2[posttest2 > 100] <- 100

temp <- data.frame(pretest2, posttest2)
```

Now let us plot these data. This builds on our previous plots that used jitter and size. In computer terminology, controlling transparency is called *alpha compositing*. The `qplot` function makes this easy with a simple `alpha` argument. You can try various levels of transparency until you get the result you desire.

The `size` and `alpha` arguments could be set as variables, in which case they would vary the point size or transparency to reflect the levels of the assigned variables. That would require a legend to help us interpret the plot. However, when you want to set them equal to fixed values, you can nest the numbers using the `I()` function. The `I()` function *i*nhibits the *i*nterpretation of its arguments. Without the `I()` function, the `qplot` function would print a legend saying that "size = 2" and "alpha = 0.15," which in our case is fairly useless information.

The `ggplot` function controls transparency with the `colour` argument to the `geom_jitter` function. That lets you control color and amount of transparency in the same option.

The `qplot` approach to Fig. 16.22 is

```
qplot(pretest2, posttest2, data = temp,
  geom = "jitter", size = I(2), alpha = I(0.15),
  position = position_jitter(width = 2) )
```

The `ggplot` approach to Fig. 16.22 is

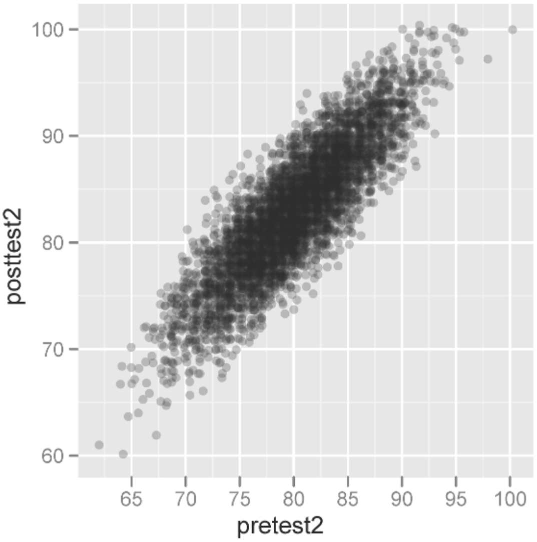


Fig. 16.22. A scatter plot demonstrating how transparency allows you to see many points at once

```
ggplot(temp, aes(pretest2, posttest2),
  size = 2, position = position_jitter(x = 2, y = 2) ) +
  geom_jitter(colour = alpha("black", 0.15) )
```

Scatter Plots with Density Contours

A different approach to studying a dense scatter plot is to draw density contours on top of the data (Fig. 16.23). With this approach, it is often better not to jitter the data, so that you can more clearly see the contours. You can do this with the density2d geom in qplot or the geom_density function in ggplot. The size = I(1) argument below reduces the point size to make it easier to see many points at once. As before, the I() function simply suppressed a superfluous legend.

The qplot approach to Fig. 16.23 is

```
qplot(pretest2, posttest2, data = temp,
  geom = c("point","density2d"), size = I(1) )
```

The ggplot approach to Fig. 16.23 is

```
ggplot(temp, aes( pretest2, posttest2) ) +
  geom_point( size = 1 ) + geom_density_2d()
```

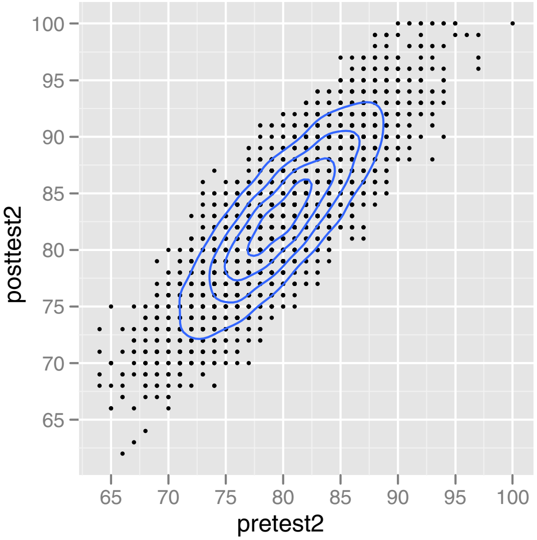


Fig. 16.23. This scatter plot shows an alternate way to see the structure in a large data set. These points are small, but not jittered, making more space for us to see the density contour lines

Scatter Plots with Density Shading

Another way to deal with densely packed points on a scatter plot is to shade them using a density function. This is best done with `ggplot`. The function does this type of plot is `stat_density2d`. It uses the `tile` geom and the built-in `..density..` variable as the fill aesthetic. To prevent it from drawing contour lines, that argument is set to FALSE. What follows is the code that creates Fig. 16.24:

```
ggplot(temp, aes( x = pretest2, y = posttest2) ) +
  stat_density2d(geom = "tile",
    aes(fill = ..density..), contour = FALSE) +
  scale_fill_continuous(
    low = "grey80", high = "black")
```

If the `scale_fill_continous` function were left out, the background color would be blue and the more dense the plots were the brighter red they would become.

Hexbin Plots

Another approach to plotting large data sets is to divide the plot surface into a set of hexagons and shade each hexagon to represent the number of points

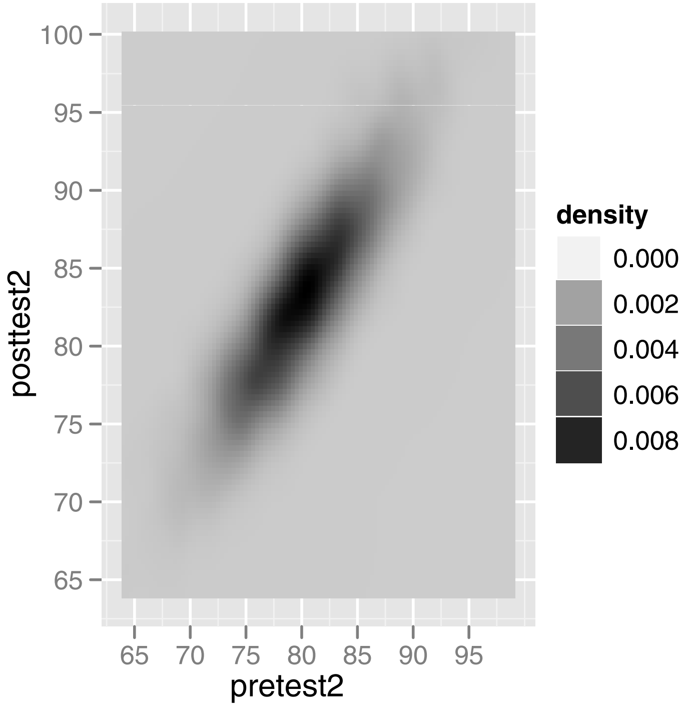


Fig. 16.24. A scatter plot with 2-D density shading

that fall within it (Fig. 16.25). In that way, you scale millions of points down into tens of bins.

In `qplot`, we can use the `hex` geom. In `ggplot`, we use the equivalent `geom_hex` function. Both use the `bins` argument to set the number of hexagonal bins you want. The default is 30; we use it here only so that you can see how to change it. As with histograms, increasing the number of bins may reveal more structure within the data.

The following function call uses `qplot` to create a color version of Fig. 16.25:

```
qplot(pretest2, posttest2, geom = "hex", bins = 30)
```

The following code uses `ggplot` to create the actual greyscale version of Fig. 16.25. The `scale_fill_continuous` function allows us to shade the plot using levels of grey. You can change the `low = "grey80"` argument to other values to get the range of grey you prefer. Of course, you could add this function call to the above `qplot` call to get it to be grey instead of color.

```
ggplot(temp, aes(pretest2, posttest2))  +
```

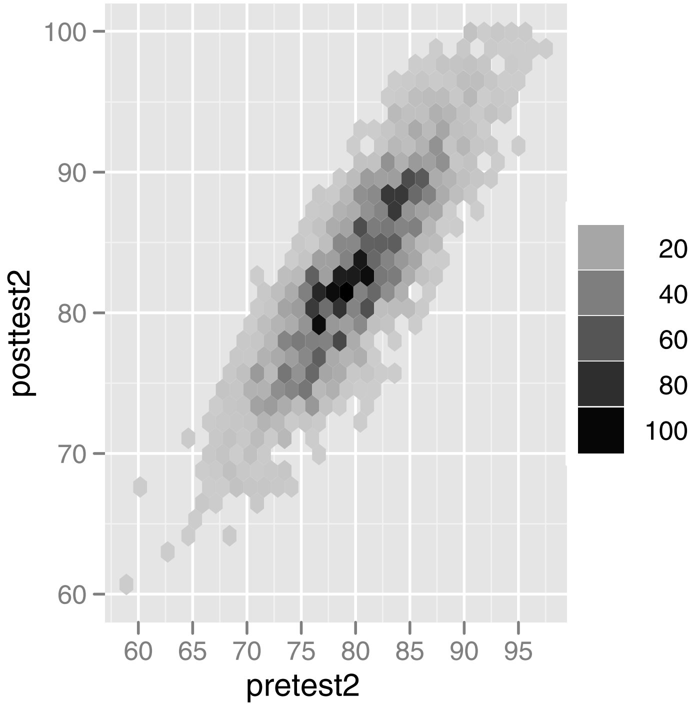


Fig. 16.25. A hexbin plot of pretest and posttest

```
geom_hex( bins = 30 ) +
scale_fill_continuous(
  low = "grey80", high = "black")
```

16.12.3 Scatter Plots with Fit Lines

While the traditional graphics plot function took quite a lot of extra effort to add confidence lines around a regression fit (Fig. 15.40), the ggplot2 package makes that automatic. Unfortunately, the transparency used to create the confidence band is not supported when you cut and paste the image as a metafile in Windows. The image in Fig. 16.26 is a slightly lower-resolution 600-dpi bitmap.

To get a regression line in qplot, simply specify geom = "smooth". However, that alone will replace the default of geom = "point", so if you want both, you need to specify geom = c("point", "smooth").

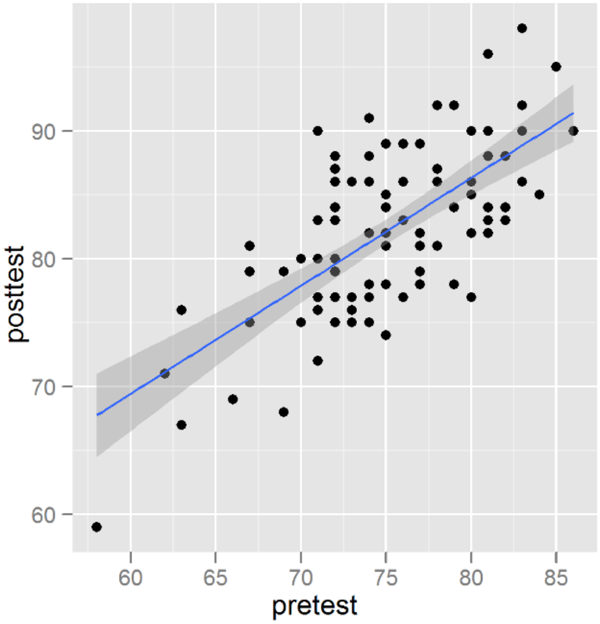


Fig. 16.26. A scatter plot with regression line and default confidence band

In ggplot, you use both the geom_point and geom_smooth functions. The default smoothing method is a lowess function, so if you prefer a linear model, include the method = lm argument.

The qplot approach to Fig. 16.26 is

```
qplot(pretest, posttest,
  geom = c("point", "smooth"), method = lm )
```

The ggplot approach to Fig. 16.26 is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() + geom_smooth(method = lm)
```

Since the confidence bands appear by default, we have to set the se argument (standard error) to FALSE to turn it off.

The qplot approach to Fig. 16.27 is

```
qplot(pretest, posttest,
  geom = c("point","smooth"), method = lm, se = FALSE )
```

The ggplot approach to Fig. 16.27 is

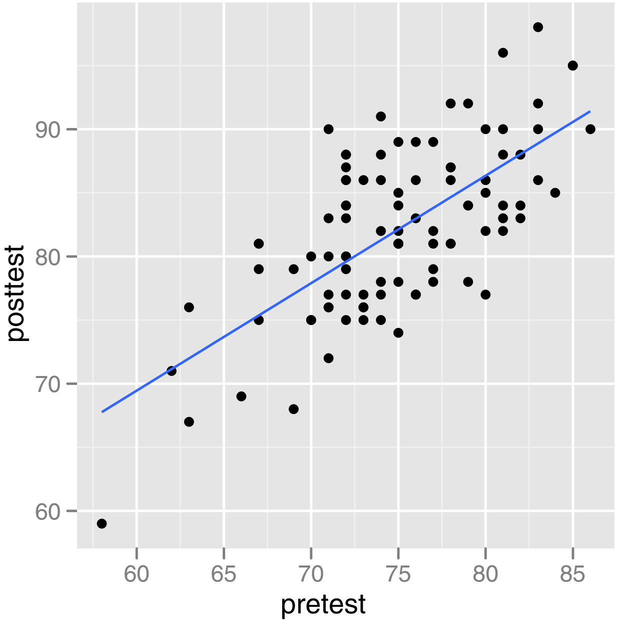


Fig. 16.27. A scatter plot with regression line with default confidence band removed

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() + geom_smooth(method = lm, se = FALSE)
```

16.12.4 Scatter Plots with Reference Lines

To place an arbitrary straight line on a plot, use the `abline` geom in `qplot`. You specify your slope and intercept using clearly named arguments. Here we are using `intercept = 0` and `slope = 1` since this is the line where posttest = pretest. If the students did not learn anything in the workshops, the data would fall on this line (assuming a reliable test). The `ggplot` function adds the `abline` function with arguments for intercept and slope.

The `qplot` approach to Fig. 16.28 is

```
qplot(pretest, posttest,
  geom = c("point", "abline"),
  intercept = 0, slope = 1)
```

The `ggplot` approach to Fig. 16.28 is

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point()+ geom_abline(intercept = 0, slope = 1)
```

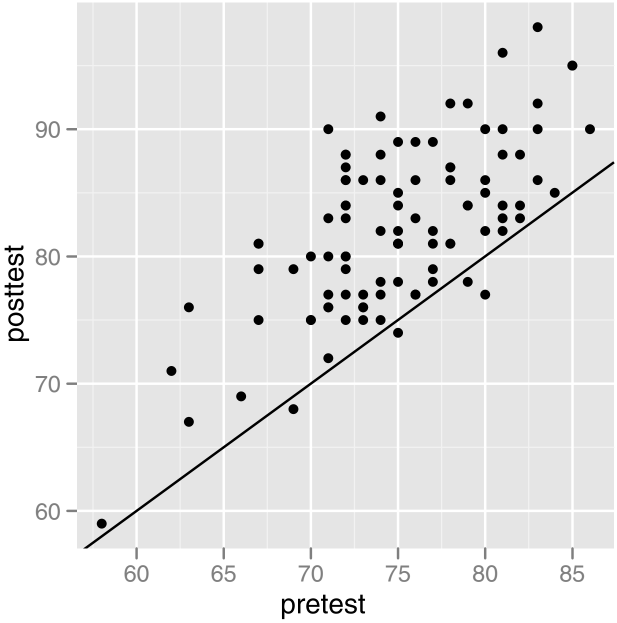


Fig. 16.28. A scatter plot with a line added where pretest=posttest. Most of the points lie above this line, showing that students did learn

Vertical or horizontal reference lines can help emphasize points or cutoffs. For example, if our students are required to get a score greater than 75 before moving on, we might want to display those cutoffs on our plot (Fig. 16.29).

In qplot, we can do this with the xintercept and yintercept arguments. In ggplot, the functions are named geom_vline and geom_hline, each with an intercept argument.

The qplot approach to Fig. 16.29 is

```
qplot(pretest, posttest,
 geom = c("point", "vline", "hline"),
 xintercept = 75, yintercept = 75)
```

The ggplot approach to Fig. 16.29 is

```
ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline( xintercept = 75 ) +
  geom_hline( yintercept = 75 )
```

To add a series of reference lines, we need to use the geom_vline or geom_hline functions (Fig. 16.30). The qplot example does not do much

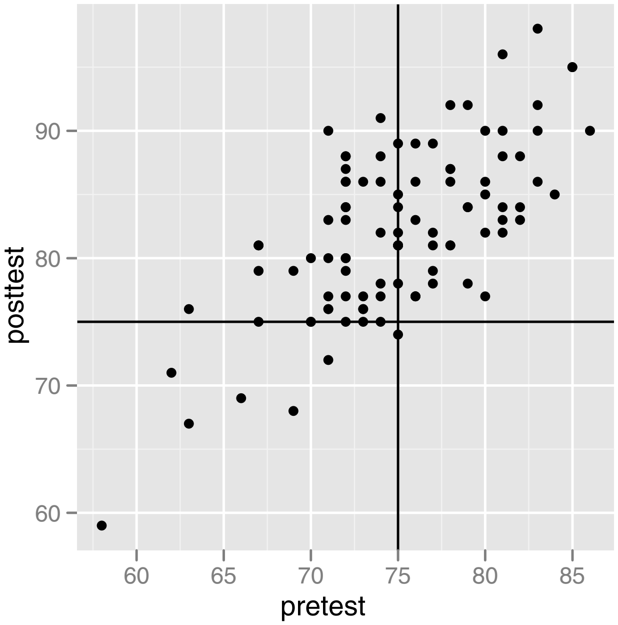


Fig. 16.29. A scatter plot with vertical and horizontal reference lines

with `qplot` itself since it cannot create multiple reference lines. So for both examples, we use the identical `geom_vline` function. It includes the `seq` function to generate the *seq*uence of numbers we needed. Without it we could have used `xintercept = c(70, 72, 74, 76, 78, 80)`. In this case, we did not save much effort, but if we wanted to add dozens of lines, the `seq` function would be much easier.

The `qplot` approach to Fig. 16.30 is

```
qplot(pretest, posttest, type = "point") +
  geom_vline( xintercept = seq(from = 70, to = 80, by = 2) )
```

The `ggplot` approach to Fig. 16.30 is

```
ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline( xintercept = seq(from = 70, to = 80, by = 2) )
```

16.12.5 Scatter Plots with Labels Instead of Points

If you do not have much data or you are only interested in points around the edges, you can plot labels instead of symbols (Fig. 16.31). The labels could be

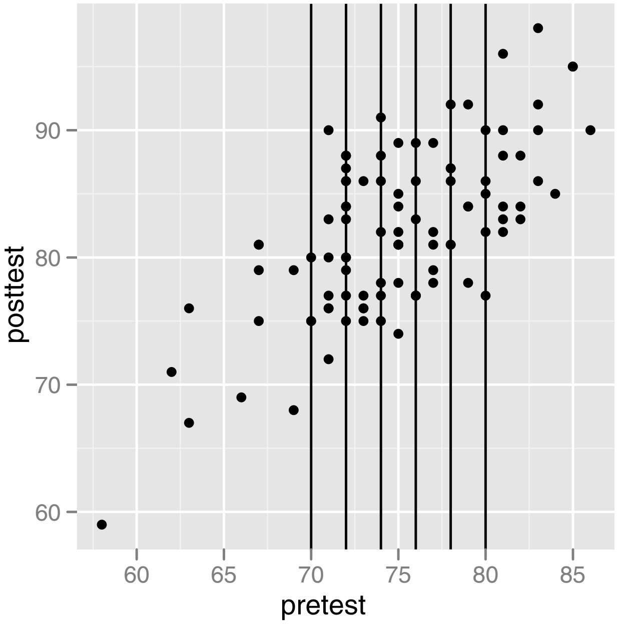


Fig. 16.30. A scatter plot with multiple vertical reference lines

identifiers such as ID numbers, people's names, or row names, or they could be values of other variables of interest to add a third dimension to the plot.

You do this using the `geom = "text"` argument in `qplot` or the `geom_text` function in `ggplot`. In either case, the `label` argument points to the values to use. Recall that in R, `row.names(mydata)` gives you the stored row names, even if these are just the sequential characters, "1," "2," and so on. We will store them in a variable named `mydata$id` and then use it with the `label` argument. The reason we do not use the form `label = row.names(mydata100)` is that the `ggplot2` package puts all of the variables it uses into a separate temporary data frame before running.

The `qplot` approach to Fig. 16.31 is

```
mydata100$id <- row.names(mydata100)
qplot(pretest, posttest, geom = "text",
  label = mydata100$id )
```

The `ggplot` approach to Fig. 16.31 is

```
ggplot(mydata100, aes(pretest, posttest,
  label = mydata100$id ) ) + geom_text()
```

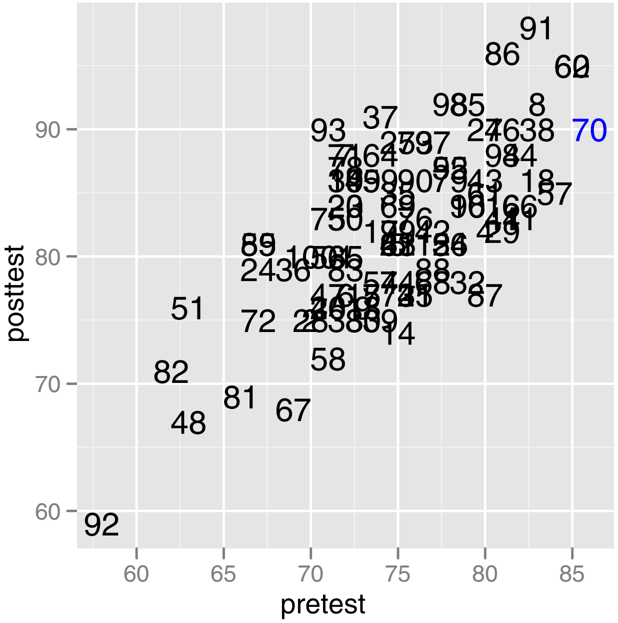


Fig. 16.31. A scatter plot with ID numbers plotted instead of points

16.12.6 Changing Plot Symbols

You can use different plot symbols to represent levels of any third variable. Factor values, such as those representing group membership, can be displayed by different plot symbols (shapes) or colors. You can use a continuous third variable to shade the colors of each point or to vary the size of each point. The ggplot2 package makes quick work of any of these options. Let us consider a plot of pretest versus posttest that uses different points for males and females (Fig. 16.32).

The qplot function can do this using the shape argument.

The ggplot function must bring a new variable into the geom_point function. Recall that aesthetics map variables into their roles, so we will nest aes(shape = gender) within the call to geom_point.

You can also set colour (note the British spelling of that argument) and size by substituting either of those arguments for shape.

The qplot approach to Fig. 16.32 is

```
qplot(pretest, posttest, shape = gender)
```

The ggplot approach to Fig. 16.32 is

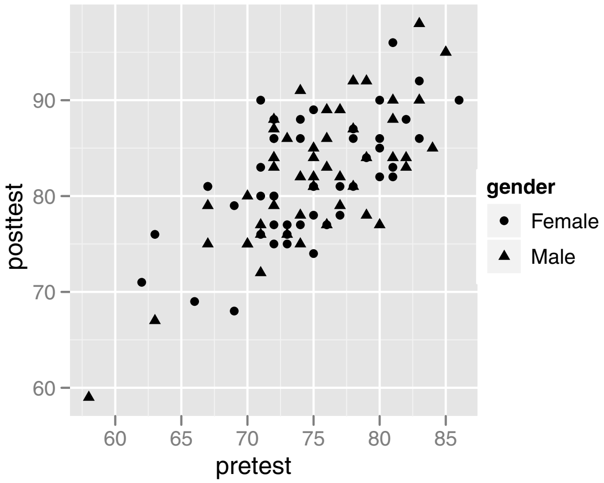


Fig. 16.32. A scatter plot with point shape determined by gender

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point( aes(shape = gender) )
```

16.12.7 Scatter Plot with Linear Fits by Group

We have seen that the `smooth` geom adds a lowess or regression line and that `shape` can include group membership. If we do both of these in the same plot, we can get separate lines for each group as shown in Fig. 16.33. The `linetype` argument causes each group to get its own style of line, solid for the females and dashed for the males. If we were publishing in color, we could use the `colour` argument to give each gender its own color.

The `qplot` approach to Fig. 16.33 is

```
qplot(pretest, posttest,
  geom = c("smooth", "point"),
  method = "lm", shape = gender,
  linetype  = gender)
```

The `ggplot` approach to Fig. 16.33 is

```
ggplot(mydata100,
  aes(pretest, posttest, shape = gender,
  linetype = gender) ) +
  geom_smooth(method = "lm") +
  geom_point()
```

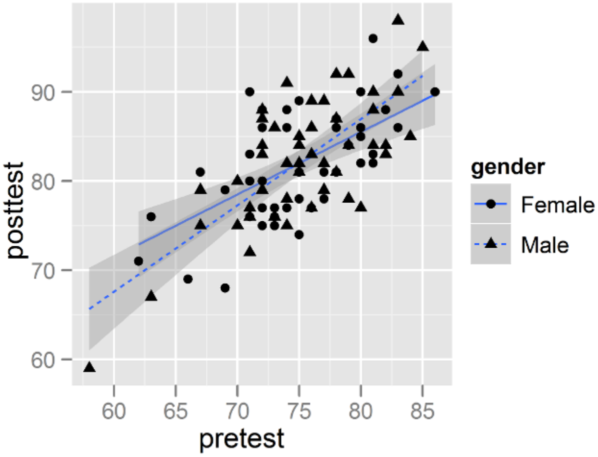


Fig. 16.33. A scatter plot with regression lines and point shape determined by gender

16.12.8 Scatter Plots Faceted by Groups

Another way to compare the scatter plots of different groups, with or without lines of fit, is through facets (Fig. 16.34). As we have already seen several times, simply adding the `facets` argument to the `qplot` function allows you to specify `rows ~ columns` of categorical variables. So

```
facets = workshop ~ gender+
```

requests a grid of plots for each workshop:gender combination, with workshop determining the rows and gender determining the columns.

The `ggplot` function works similarly, using the `facet_grid` function to do the same. If you have a continuous variable to condition on, you can use the `chop` function from the `ggplot2` package or the `cut` function that is built into R to break up the variable into groups.

The `qplot` approach to Fig. 16.34 is

```
qplot(pretest, posttest, geom = c("smooth", "point"),
  method = "lm", shape = gender, facets = workshop ~ gender)
```

The `ggplot` approach to Fig. 16.34 is

```
ggplot(mydata100, aes( pretest, posttest) ) +
  geom_smooth( method = "lm" ) + geom_point() +
  facet_grid( workshop ~ gender )
```

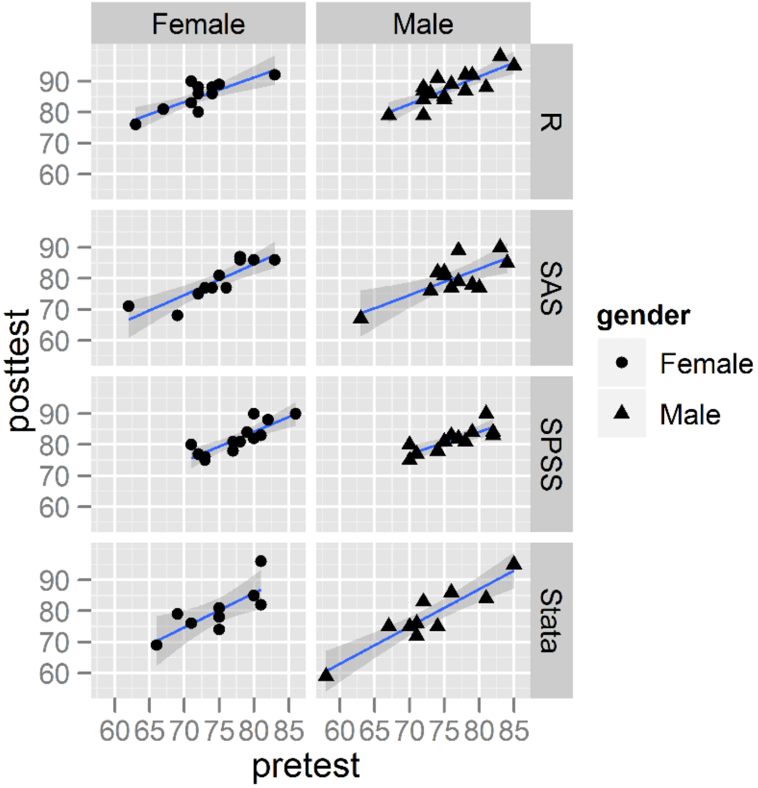


Fig. 16.34. A scatter plot with facets showing linear fits for each workshop and gender combination

16.12.9 Scatter Plot Matrix

When you have many variables to plot, a scatter plot matrix is helpful (Fig. 16.35). You lose a lot of detail compared to a set of full-sized plots, but if your data set is not too large, you usually get the gist of the relationships.

The `ggplot2` package has a separate `plotmatrix` function for this type of plot. Simply entering the following function call will plot variables 3 through 8 against one another (not shown):

```
plotmatrix( mydata100[3:8] )
```

You can embellish the plots with many of the options we have covered earlier in this chapter. Shown below is an example of a scatter plot matrix (Fig. 16.35) with smoothed lowess fits for the entire data set (i.e., not by group). The density plots on the diagonals appear by default:

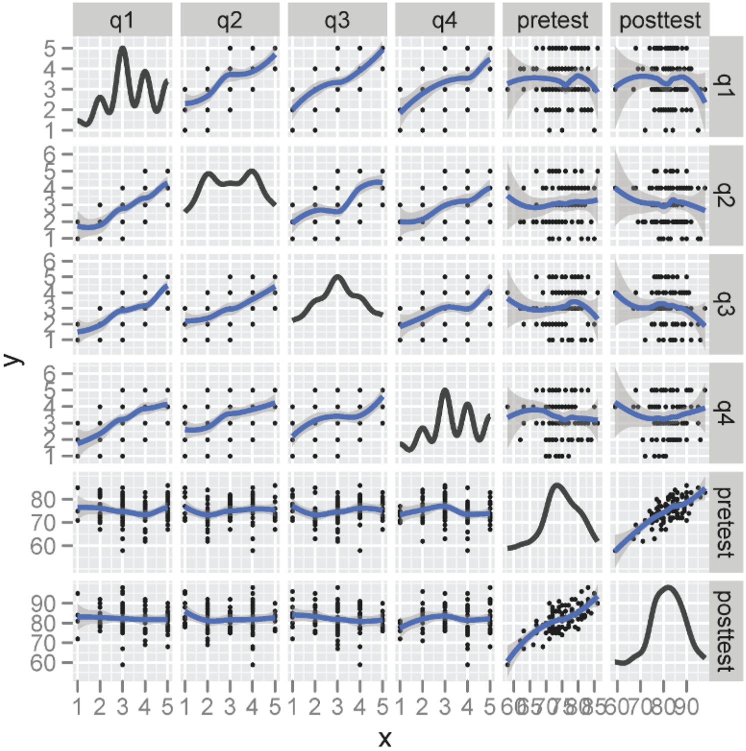


Fig. 16.35. A scatter plot matrix with lowess curve fits on the off-diagonal plots, and density plots on the diagonals. The x-axis for pretest and posttest are too dense to read. Plotting fewer variables would solve that problem

```
plotmatrix( mydata100[3:8] ) +
  geom_smooth()
```

The lowess fit generated some warnings, but that is not a problem. It said, “There were 50 or more warnings (use `warnings()` to see the first 50).” Note that on the right side of the x-axis the numbers are packed so densely that you cannot read them. The overall images are still clear but if you need to be able to read the axes on a scatter plot matrix, simply plot fewer variables at a time.

The next example gets fancier by assigning a different symbol shape and linear fits per group (plot not shown):

```
plotmatrix( mydata100[3:8],
  aes( shape = gender ) ) +
  geom_smooth(method = lm)
```

16.13 Box Plots

We discussed what box plots are in the Chap. 15 "Traditional Graphics," Sect. 15.12. We can recreate all those examples using the ggplot2 package, except for the "notches" to indicate possible group differences, shown in the upper right of Fig. 15.47.

The simplest type of box plot is for a single variable (Fig. 16.36). The qplot function uses the simple form of factor("") to act as its x-axis value. The y argument is the variable to plot: in this case, posttest. The geom of boxplot specifies the main display type. The xlab = "" argument blanks out the label on the x-axis, which would have been a meaningless "factor("")".

The equivalent ggplot approach is almost identical with its ever-present aes arguments for x and y and the geom_boxplot function to draw the box. The scale_x_discrete function simply blanks out the x-axis label.

The qplot approach to Fig. 16.36 is

```
qplot(factor(""), posttest,
  geom = "boxplot", xlab = "")
```

The ggplot approach to Fig. 16.36 is

```
ggplot(mydata100,
  aes(factor(""), posttest) ) +
  geom_boxplot() +
  scale_x_discrete("")
```

Adding a grouping variable like workshop makes box plots much more informative (Fig. 16.37; ignore the overlaid strip plot points for now). These are the same function calls as above but with the x argument specified as workshop. We will skip showing this one in favor of the next.

The qplot approach to box plots (figure not shown) is

```
qplot(workshop, posttest, geom = "boxplot" )
```

The ggplot approach to box plots (figure not shown) is

```
ggplot(mydata100,
  aes( workshop, posttest) ) +
  geom_boxplot()
```

Now we will do the same plot but with an added jittered strip plot on top of it (Fig. 16.37). This way we get the box plot information about the median and quartiles plus we get to see any interesting structure in the points that would otherwise have been lost. As you can see, qplot now has jitter added to its geom argument, and ggplot has an additional geom_jitter function. Unfortunately, the amount of jitter that both functions provide by default is optimized for a much larger data set. So these next two sets of code do the plot shown in Fig. 16.37, but with much more jitter.

The qplot approach to Fig. 16.37 with more jitter added is

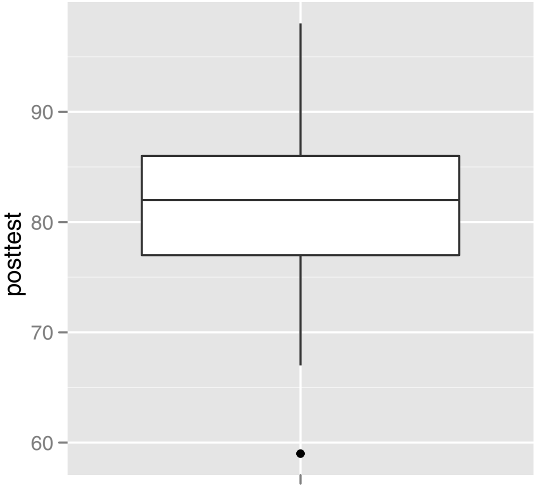


Fig. 16.36. A box plot of posttest

```
qplot(workshop, posttest,
  geom = c("boxplot", "jitter") )
```

The ggplot approach to Fig. 16.37 with more jitter added is

```
ggplot(mydata100,
  aes(workshop, posttest )) +
  geom_boxplot() + geom_jitter()
```

What follows is the exact code that created Fig. 16.37. The qplot function does not have enough control to request both the box plot and jitter while adjusting the amount of jitter:

```
ggplot(mydata100,
 aes(workshop, posttest )) +
 geom_boxplot() +
 geom_jitter(position = position_jitter(width = .1))
```

To add another grouping variable, you only need to add the fill argument to either qplot or ggplot. Compare the resulting Fig. 16.38 to the result we obtained from traditional graphics in the lower right panel of Fig. 15.47. The ggplot2 version is superior in many ways. The genders are easier to compare for a given workshop because they are now grouped side by side. The shading makes it easy to focus on one gender at a time to see how they changed across the levels of workshop. The labels are easier to read and did not require the

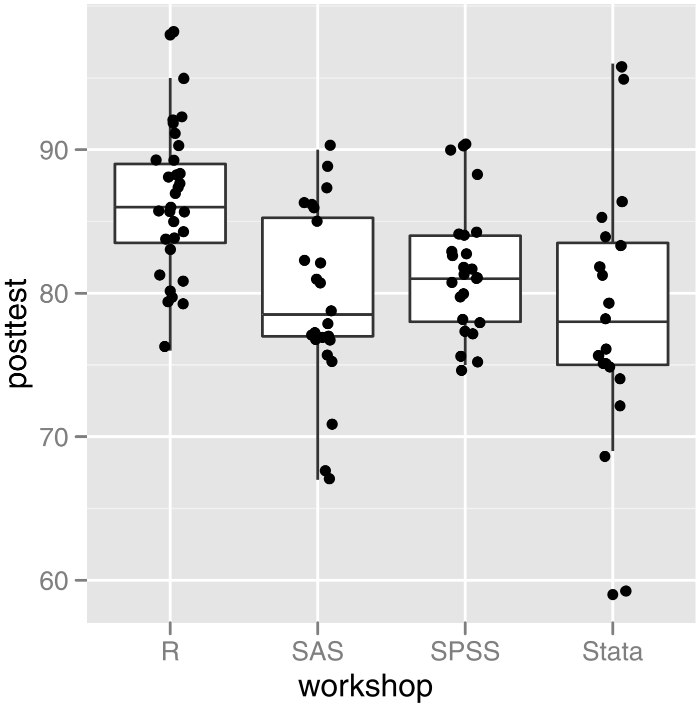


Fig. 16.37. A box plot comparing workshop groups on posttest, with jittered points on top

custom sizing that we did earlier to make room for the labels. The ggplot2 package usually does a better job with complex plots and makes quick work of them, too.

The qplot approach to Fig. 16.38 is

```
qplot(workshop, posttest,
  geom = "boxplot", fill = gender ) +
  scale_fill_grey( start = 0, end = 1 )
```

The ggplot approach to Fig. 16.38 is

```
ggplot(mydata100,
  aes(workshop, posttest) ) +
  geom_boxplot( aes(fill = gender), colour = "black") +
  scale_fill_grey( start = 0, end = 1 )
```

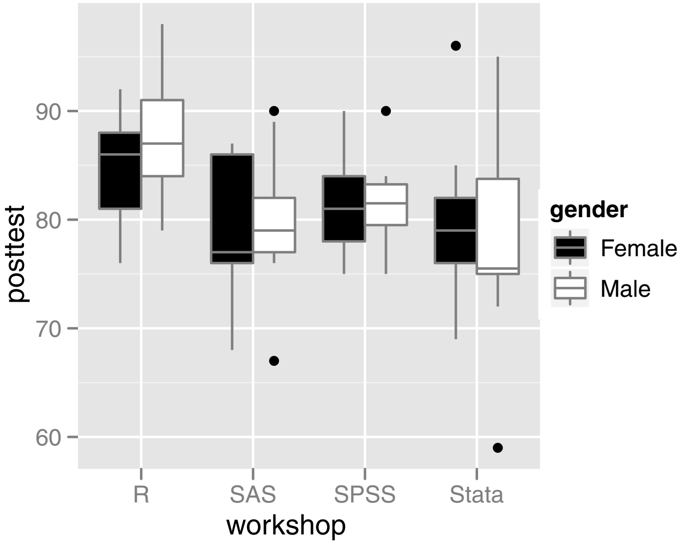


Fig. 16.38. A box plot comparing workshop and gender groups on posttest

16.14 Error Bar Plots

Plotting means and 95% confidence intervals, as in Fig. 16.39, is a task that stretches what `qplot` was designed to do. Due to its complexity, this type of plot is really not worth doing using `qplot`, so I only show the code for `ggplot`. This plot builds on a jittered strip plot of points which we did earlier in Sect. 16.11). Notice that I had to use the `as.numeric` function for our `x` variable: workshop. Since workshop is a factor, the software would not connect the means across the levels of `x`. Workshop is not a continuous variable, so that makes sense! Still, connecting the means with a line is a common approach, one that facilitates the study of higher-level interactions.

The key addition for this plot is `stat_summary`, which we use twice. First, we use the argument `fun.y = "mean"` to calculate the group means. We also use the `geom = "smooth"` argument to connect them with a line. Next, we use `fun.data = "mean_cl_normal"` to calculate *c*onfidence *l*imits for the means based on a normal distribution and display them with the `errorbar` geom. You can try various values for the `width` argument until you are satisfied with the error bar widths. The `size = 1` argument sets the thickness of the error bar lines.

```
ggplot(mydata100,
  aes( as.numeric(workshop), posttest ) ) +
  geom_jitter( size = 1,
```

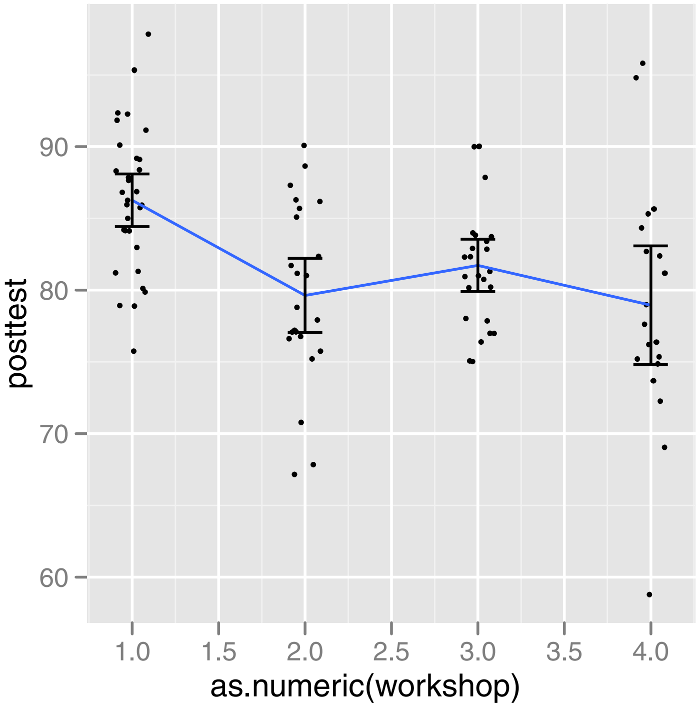


Fig. 16.39. An error bar plot with lines running through the means, with default axis labels

```
    position = position_jitter(width = .1) )  +
  stat_summary(fun.y = "mean",
    geom = "smooth", se = FALSE) +
  stat_summary(fun.data = "mean_cl_normal",
    geom = "errorbar", width = .2, size = 1)
```

16.15 Geographic Maps

Displaying data on maps as they change values across geographic regions is useful in many fields. As of this writing, SPSS, Inc. does not offer mapping. However, if you use SPSS, you can use the SPSS Statistics-R Integration Plug-in to do your maps in R.

SAS offers mapping in SAS/GRAPH and even offers a more comprehensive Geographic Information System (GIS) in SAS/GIS. While maps are usually static images displaying geographic information, a GIS allows you to interact with a map. For example, SAS/GIS allows you to click on any map region and have it execute any SAS program you like.

R offers mapping and extensive spatial analysis methods. However, it defers to other GIS systems for interactive capabilities. The Geographic Resources Analysis Support System (GRASS) is a popular free and open source GIS system available at http://grass.fbk.eu/. The R package spgrass6 provides an interface between R and GRASS.

For the analysis of spatial data, R's capabilities are quite powerful. The Spatial Task View at CRAN lists almost one hundred packages which offer a wide range of spatial analysis capabilities. An excellent book on the subject is *Applied Spatial Data Analysis with R* [8]. A good book that shows how to use R to control several other free and open source mapping or GIS packages is Hengl's *A Practical Guide to Geostatistical Mapping* [29].

R offers three main approaches to mapping. The oldest is the built-in maps package [7]. That comes with an eclectic set of maps stored as lists of latitude and longitude values. It includes a map function to display them, which works similarly to the traditional plot function.

The second approach is provided by the sp package [44]. It offers a rich set of *Spatial* map data structures that store points, lines, polygons, and even dataframes. The structures are rich enough to allow a region that contains a lake, on which there is an island, which can have its own lake! That level of flexibility has encouraged many other packages to adopt these Spatial structures. As useful as it is, that level of complexity is beyond our scope.

The third approach is the one we will use: the ggplot2 package. As you have seen, ggplot2 is very powerful and flexible. In addition, it stores its map data in dataframes, so you have much less to learn about using them. The maps from the maps and sp packages can both be converted to dataframes for use in ggplot2.

Let us now work through an example map from one of the ggplot2 help files. First we need a map which we can get from the maps package:

```
> library("maps")
> library("ggplot2")
> myStates <- map_data("state")
> head(myStates)
       long      lat group order  region subregion
1 -87.46201 30.38968     1     1 alabama      <NA>
2 -87.48493 30.37249     1     2 alabama      <NA>
3 -87.52503 30.37249     1     3 alabama      <NA>
4 -87.53076 30.33239     1     4 alabama      <NA>
5 -87.57087 30.32665     1     5 alabama      <NA>
6 -87.58806 30.32665     1     6 alabama      <NA>
```

The map_data function is a ggplot2 function which converts map classes of objects into dataframes. The map named state is one of the maps included in the maps package.

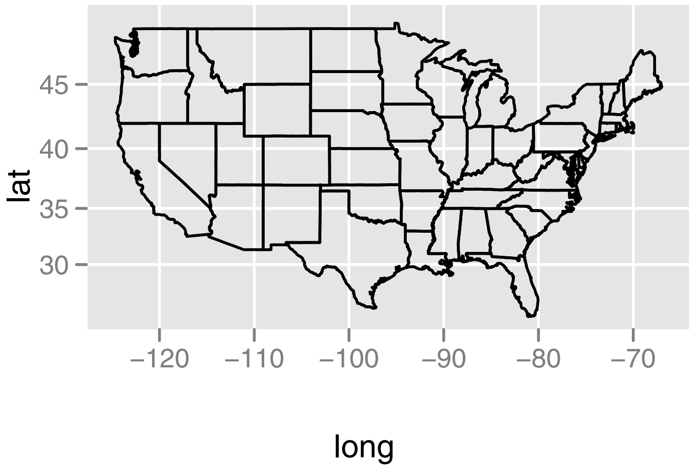


Fig. 16.40. A map of the USA using the `path` geom

We now have a dataframe myStates which has the longitude (long) and latitude (lat) for the United States of America.

The group variable stores information about which points form a polygon so R will know to "pick up the pen" when drawing the various regions.

The order variable will tell R the order in which to draw the points. At the moment, that is redundant information because the points are already in the proper order. However, after merging with the data we wish to display on the map, the order is likely to get scrambled. We can use the order variable to restore it.

Finally, we have the region and subregion variables. In this case they store the state names and pieces of states where applicable. Alabama has subregions set to NA indicating that the state map that we converted to a dataframe contains no subregions. If we had instead converted the county map, then the counties of Alabama would have shown up as subregions. However, the state map does contain some subregions. New York as different pieces such as Manhattan Island, and that is how they are stored.

We can display the map itself with no data (Fig. 16.40) using `qplot`:

```
qplot(long, lat, data = myStates, group = group,
  geom = "path", asp  = 1)
```

The long and lat variables supply the x-y coordinates from myStates and the group variable supplies "lift the pen" information to the `group` argument. Since I am only drawing the lines around each state I am using the path geom.

Maintaining the proper aspect ratio for a map is important. I am doing that with the `asp = 1` setting. The very similar code using the `ggplot` function follows:

```
ggplot(data = myStates, aes(long, lat, group = group) )+
  geom_path() +
  coord_map()
```

You see it uses a `coord_map` function to maintain the proper aspect ration.

Now we need some data to display. R has a built-in dataframe of crime measures by state:

```
> myArrests <- USArrests

> head(myArrests)
           Murder Assault UrbanPop Rape
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
Colorado      7.9     204       78 38.7
```

There are two problems with this file. First, the state names are capitalized while in our map dataframe they are all lower case. Second, the state names are stored as row names rather than as a variable. We will need a variable to merge by soon. We can fix both those problems with the following:

```
> myArrests$region <- tolower( rownames(USArrests) )

> head(myArrests)
           Murder Assault UrbanPop Rape     region
Alabama      13.2     236       58 21.2    alabama
Alaska       10.0     263       48 44.5     alaska
Arizona       8.1     294       80 31.0    arizona
Arkansas      8.8     190       50 19.5   arkansas
California    9.0     276       91 40.6 california
Colorado      7.9     204       78 38.7   colorado
```

The `tolower` function is part of the `ggplot2` package and it simply lowers the case of all letters in a character string.

The next step in any mapping program would be to establish the link between our two data sets: myStates and myArrests. If we were in SAS, we would leave the two files separate and use:

```
PROC GMAP  MAP = myStates  DATA = myArrests;
```

```
ID region;
CHORO Assault;
```

SAS would then use the ID statement to match the proper assault value to each region.

The ggplot approach is to merge the two files and then supply the combined set to the plotting functions. The merge function can handle that task:

```
> myBoth <- merge(
+   myStates,
+   myArrests, by = "region")

> myBoth[1:4, c(1:5,8)]

   region      long      lat group order Assault
1 alabama -87.46201 30.38968     1     1     236
2 alabama -87.48493 30.37249     1     2     236
3 alabama -87.95475 30.24644     1    13     236
4 alabama -88.00632 30.24071     1    14     236
```

The two dataframes merged just fine, but now the values of the order variable indicate that our state outlines will no longer be drawn in their proper order. We can fix that with:

```
> myBoth <- myBoth[ order(myBoth$order), ]

> myBoth[1:4, c(1:5, 8)]

   region      long      lat group order Assault
1 alabama -87.46201 30.38968     1     1     236
2 alabama -87.48493 30.37249     1     2     236
6 alabama -87.52503 30.37249     1     3     236
7 alabama -87.53076 30.33239     1     4     236
```

That first statement looks a bit odd since we are calling the order function on the order variable. If you need to brush up on sorting, see Sect. 10.18.

I am now ready to draw the map. Before I was only displaying state outlines, so I used the path geom. Now, however, I want to fill in the states with colors that reflect the amount of Assault in each state, so I must view them as polygons and use that geom instead. A polygon is a line that encloses space by ending at the same point at which it started. The fill argument supplies the variable. Now that color is involved, we have an additional function call to scale_fill_continuous If I left that out, the map would be in color. That's much nicer but it would also raise the price of this book and on anything you publish as well. Therefore, I added it to specify the shades of grey I liked. If assault were a factor, I would have to use the scale_fill_grey function to control the shades of grey.

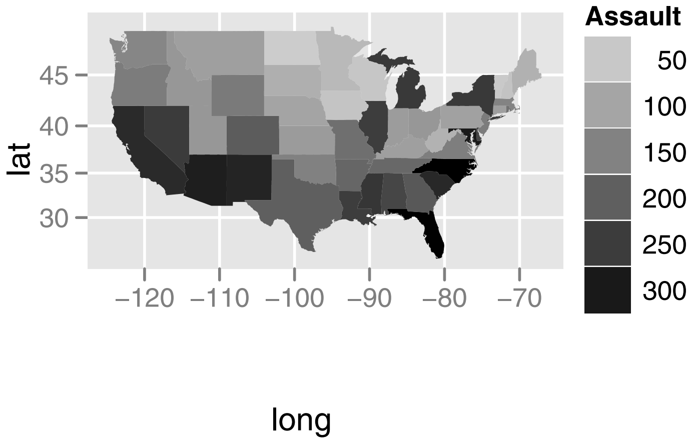


Fig. 16.41. A map of the USA using the `polygon` geom

The following is the `qplot` code for Fig. 16.41, bottom:

```
qplot(long, lat, data = myBoth, group = group,
  fill = Assault, geom = "polygon", asp = 1) +
  scale_fill_continuous(low = "grey80", high = "black")
```

And this is the `ggplot` code for Fig. 16.41, bottom:

```
ggplot(data = myBoth,
    aes(long, lat, fill = Assault, group = group) )+
  geom_polygon() +
  coord_map() +
  scale_fill_continuous(low = "grey80", high = "black")
```

16.15.1 Finding and Converting Maps

While SAS has a comprehensive set of maps built in, R comes with an eclectic mix of maps. However, maps are available from many locations so you should have no problem finding what you need.

The `cshapes` package, written by Weidmann, et al. [69], contains world country maps as they existed each year since 1948 until recently. When I wrote this, the maps end at 2008 but this will grow with time. In *Mapping and Measuring Country Shapes* Weidmann and Gleditsch [69] demonstrate how to use the maps, which are stored as *Spatial* objects.

The Internet is a good source of maps. The Global Administrative Areas site at http://www.gadm.org/ is particularly useful since their maps are already R Spatial objects. You can load them directly into R by putting their URLs into the load function. For example, to load a map of Italy, you can use:

```
load( url( "http://gadm.org/data/rda/ITA_adm1.RData") )
```

The maps there are all named *gadm*.

Another good source of maps is the Natural Earth Web site at http://www.naturalearthdata.com/. The maps there are not in the more convenient Spatial format, but their license allows for commercial use without getting permission.

As shown above, you can convert any of the maps in the maps package using ggplot2's map_data function. It also has a fortify function that will convert maps in the sp package's Spatial classes, including the SpatialPolygon or SpatialPolygonDataFrame.

The maptools package has a readShapeSpatial function that reads the popular shapefile map format and converts it to a SpatialPolygon or SpatialPolygonDataFrame. From there you can use fortify to finally get the map into a data frame useful with ggplot2.

16.16 Logarithmic Axes

If your data have a very wide range of values, working in a logarithmic scale is often helpful. In ggplot2 you can approach this in three different ways. First, you can take the logarithm of the data before plotting:

```
qplot( log(pretest), log(posttest) )
```

Another approach is to use evenly placed tick marks on the plot but have the axis values use logarithmic values such as 10^1, 10^2, and so on. This is what the scale_x_log10 function does (similarly for the y-axis, of course). There are similar functions for natural logarithms, scale_x_log, and base 2 logarithms, scale_x_log2:

```
qplot(pretest, posttest, data = mydata100) +
  scale_x_log10() + scale_y_log10()
```

Finally, you can have the tick marks spaced unevenly and use values on your original scale. The coord_trans function does that. Its arguments for the various bases of logarithms are log10, log, and log2.

```
qplot(pretest, posttest, data = mydata100) +
  coord_trans("log10", "log10")
```

With our data set, the range of values is so small that this last plot will not noticeably change the axes. Therefore, we do not show it.

16.17 Aspect Ratio

Changing the aspect ratio of a graph can be far more important than you might first think. When a plot displays multiple lines, or a cyclical time series (in essence a set of multiple lines), you generally want to compare the slopes or angles of the lines. Research has shown that when the average of the lines' angles is approximately 45°, people make more accurate comparisons [13].

Unless you specify an aspect ratio for your graph, `qplot` and `ggplot` will match the dimensions of your output window and allow you to change those dimensions using your mouse, as you would for any other window.

If you are routing your output to a file, however, it is helpful to be able to set the aspect ratio using code. You set it using the `coord_equal` function. If you leave it empty, as in `coord_equal()`, it will make the x- and y-axes of equal lengths. If you specify this while working interactively, you can still reshape your window, but the graph will remain square. Specifying a ratio parameter follows the form "height/width." For a mnemonic, think of how R specifies [rows,columns]. The following example would result in a graph that is four times wider than it is high (not shown):

```
qplot(pretest, posttest) + coord_equal(ratio = 1/4)
```

16.18 Multiple Plots on a Page

In the previous chapter on traditional graphics, we discussed how to put multiple plots on a page. However, `ggplot2` uses the grid graphics system, so that method does not work. We previously saw the multiframe plot shown again in Fig. 16.42. Let us now look at how it was constructed. This time we will skip the bar plot details and focus on how we combined the plots.

We first clear the page with the `grid.newpage` function. This is an important step as otherwise plots printed using the following methods will appear on top of others.

```
grid.newpage()
```

Next, we use the `pushViewport` function to define the various frames called *viewports* in the grid graphics system. The `grid.layout` argument uses R's common format of [rows, columns]. The following example sets up a 2×2 grid for us to use:

```
pushViewport( viewport(layout = grid.layout(2, 2) ) )
```

In traditional graphics, you would now just do the graphs in order and they would find their place. However, in the grid system, we must save the plot to an object and then use the `print` function to print it into the viewport we desire. The object name "p" is commonly used as an object name for the plot. Since there are many ways to add to this object, it is helpful to keep it short. To emphasize that this is something we get to name, we will use "myPlot." The name is not particularly important since it usually used again for plots that follow. It is only the filename that you create for publication that needs to be descriptive.

The `print` function has a `vp` argument that lets you specify the *v*iew*p*ort's position in row(s) and column(s). In the following example, we will print the graph to row 1 and column 1:

```
myPlot <- ggplot(mydata100,
    aes(gender, fill = workshop) ) +
  geom_bar(position  = "stack") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title = "position = stack" )

print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 1) )
```

The next plot prints to row 1 and column 2.

```
myPlot <- ggplot(mydata100,
    aes(gender, fill = workshop) ) +
  geom_bar(position = "fill") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title = "position = fill" )

print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 2) )
```

The third and final plot is much wider than the first two. So we will print it to row 2 in both columns 1 and 2. Since we did not set the aspect ratio explicitly, the graph will resize to fit the double-wide viewport.

```
myPlot <- ggplot(mydata100,
    aes(gender, fill = workshop) ) +
  geom_bar(position = "dodge")  +
  scale_fill_grey(start = 0, end = 1) +
  opts( title = "position = dodge" )

print(myPlot, vp = viewport(
  layout.pos.row = 2,
  layout.pos.col = 1:2) )
```

The next time you print a plot without specifying a viewport, the screen resets back to its previous full-window display. The code for the other multiframe plots is in the example program in Sect. 16.22.2.

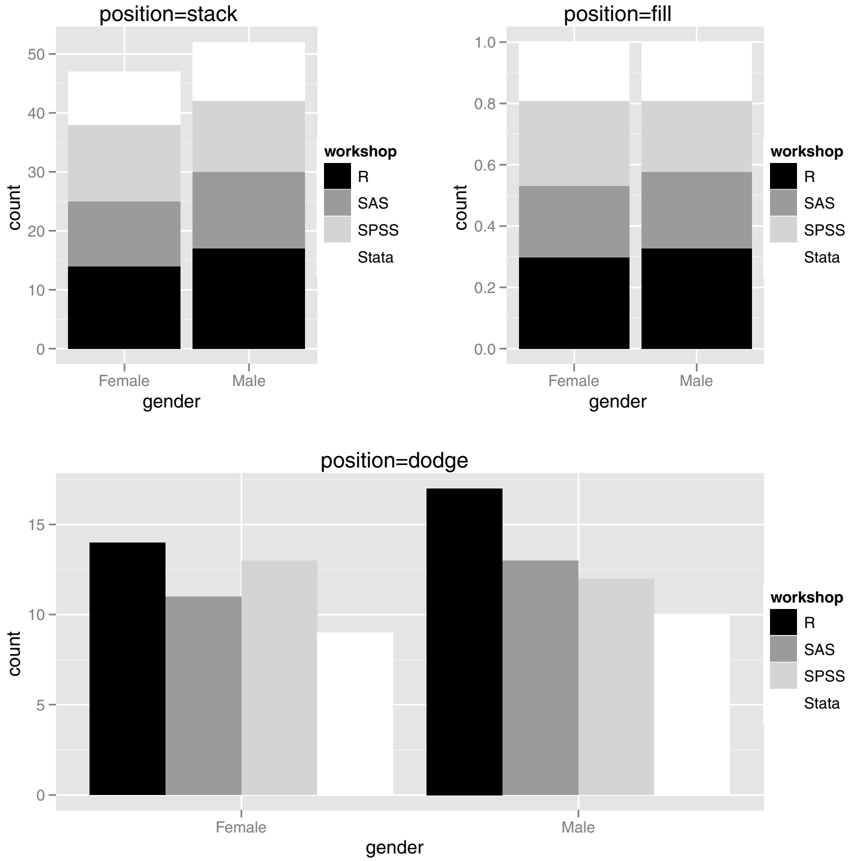


Fig. 16.42. Multiframe demonstration plot

16.19 Saving `ggplot2` Graphs to a File

In Sect. 14.10, "Graphics Devices," we discussed various ways to save plots in files. Those methods work with the `ggplot2` package, and in fact they are the only way to save a multiframe plot to a file.

However, the `ggplot2` package has its own function that is optimized for saving single plots to a file. To save the last graph you created, with either `qplot` or `ggplot`, use the `ggsave` function. It will choose the proper graphics device from the file extension.

For example, the following function call will save the last graph created in an encapsulated postscript file:

```
> ggsave("mygraph.eps")
```

```
Saving 4.00" x 3.50" image
```

It will choose the width and height from your computer monitor and will report back those dimensions. If you did not get it right, you can change those dimensions and rerun the function. Alternatively, you can specify the `width` and `height` arguments in inches or, for bitmapped formats like Portable Network Graphics (png), in dots per inch. See `help("ggsave")` for additional options.

The file will go to your working directory unless you specify a full path as part of the filename.

16.20 An Example Specifying All Defaults

Now that you have seen some examples of both `qplot` and `ggplot`, let us take a brief look at the full power of `ggplot` by revisiting the scatter plot with a regression line (Fig. 16.43). We will first review both sets of code, exactly as described in Sect. 16.12.3.

First, done with `qplot`, it is quite easy and it feels similar to the traditional graphics `plot` function:

```
qplot(pretest, posttest,
  geom = c("point", "smooth"), method = "lm" )
```

Next, let us do it using `ggplot` with as many default settings as possible. It does not require much additional typing, and it brings us into the grammar-of-graphics world. We see the new concepts of aesthetic mapping of variables and geometric objects, or geoms:

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() +
  geom_smooth(method = "lm")
```

Finally, here it is again in `ggplot` but with no default settings. We see that the plot actually has two layers: one with points and another with the smooth line. Each layer can use different data frames, variables, geometric objects, statistics, and so on. If you need graphics flexibility, then `ggplot2` is the package for you!

```
ggplot() +
layer(
  data    = mydata100,
  mapping = aes(pretest, posttest),
  geom    = "point",
  stat    = "identity"
) +
layer(
```

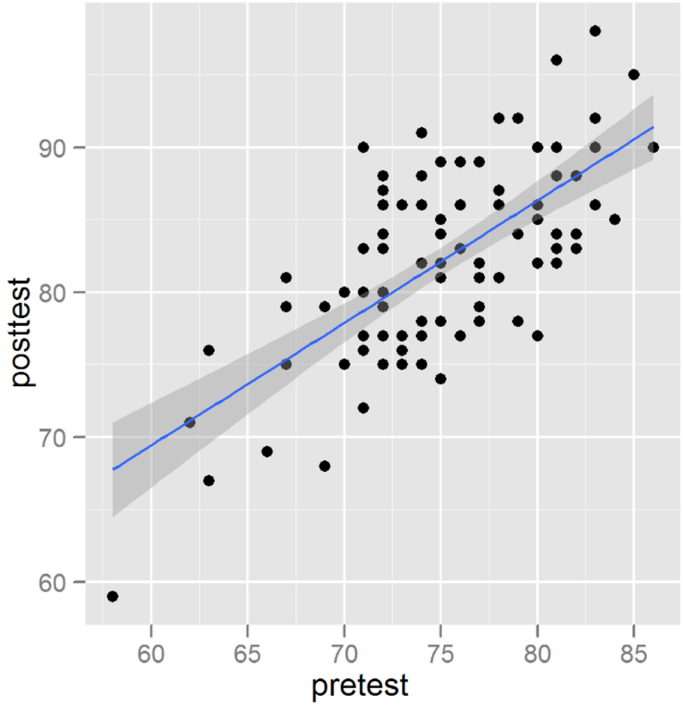


Fig. 16.43. This same scatter plot results from several types of programs shown in the text

```
  data    = mydata100,
  mapping = aes(pretest, posttest),
  geom    = "smooth",
  stat    = "smooth",
  method  = "lm"
) +
coord_cartesian()
```

16.21 Summary of Graphics Elements and Parameters

We have seen many ways to modify plots in the `ggplot2` package. The `ggopt` function is another way. You can set the parameters of all future graphs in the current session with the following function call. See `help("ggopt")` function for many more parameters:

```
ggopt(
  background.fill  = "black",
  background.color = "white",
  axis.colour = "black"  # default axis fonts are grey.
  )
```

The opts function is useful for modifying settings for a single plot. For example, when colors, shapes, or labels make a legend superfluous, you can suppress it with

```
+ opts(legend.position = "none")
```

See help("opts") for more examples.

The plots created with both qplot and ggplot make copious use of color. Since our examples did not really need color we suppressed it with

```
...+ scale_fill_grey(start = 0, end = 1)
```

An alternate way of doing this is with the theme_set function. To use levels of grey, use:

```
theme_set( theme_grey() )
```

To limit colors to black and white, use:

```
theme_set( theme_bw() )
```

To return to the default colors, use:

```
theme_set()
```

Enter help("theme_set") for details.

16.22 Example Programs for Grammar of Graphics

SAS does not follow the grammar-of-graphics model. See previous chapter for SAS/Graph examples. The SPSS examples are sparse compared to those in R. That is due to space constraints, not lack of capability. For examples using SPSS legacy graphics, see previous chapter.

16.22.1 SPSS Program for Graphics Production Language

```
* Filename: GraphicsGG.sps .

CD  C:\myRfolder .
GET FILE =  mydata.sav .

* GPL statements for histogram of q1.
```

```
GGRAPH
  /GRAPHDATASET NAME="graphdataset"
  VARIABLES=q1 MISSING=LISTWISE
  REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: q1=col(source(s), name("q1"))
 GUIDE: axis(dim(1), label("q1"))
 GUIDE: axis(dim(2), label("Frequency"))
 ELEMENT: interval(position(summary.count(bin.rect(q1))) ,
  shape.interior(shape.square))
END GPL.

* GPL statements for bar chart of gender.
GGRAPH
  /GRAPHDATASET NAME="graphdataset" VARIABLES=gender
  COUNT()[name="COUNT"] MISSING=LISTWISE REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: gender=col(source(s), name("gender"), unit.category())
 DATA: COUNT=col(source(s), name("COUNT"))
 GUIDE: axis(dim(1), label("gender"))
 GUIDE: axis(dim(2), label("Count"))
 SCALE: cat(dim(1))
 SCALE: linear(dim(2), include(0))
 ELEMENT: interval(position(gender*COUNT),
   shape.interior(shape.square))
END GPL.

* GPL syntax for scatter plot of q1 by q2.
GGRAPH
  /GRAPHDATASET NAME="graphdataset"
    VARIABLES=q1 q2 MISSING=LISTWISE
  REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: q1=col(source(s), name("q1"))
 DATA: q2=col(source(s), name("q2"))
 GUIDE: axis(dim(1), label("q1"))
 GUIDE: axis(dim(2), label("q2"))
 ELEMENT: point(position(q1*q2))
END GPL.
```

```
* Chart Builder.
GGRAPH
  /GRAPHDATASET NAME="graphdataset"
    VARIABLES=workshop q1 q2 q3 q4
  MISSING=LISTWISE REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: workshop=col(source(s), name("workshop"))
 DATA: q1=col(source(s), name("q1"))
 DATA: q2=col(source(s), name("q2"))
 DATA: q3=col(source(s), name("q3"))
 DATA: q4=col(source(s), name("q4"))
 TRANS: workshop_label = eval("workshop")
 TRANS: q1_label = eval("q1")
 TRANS: q2_label = eval("q2")
 TRANS: q3_label = eval("q3")
 TRANS: q4_label = eval("q4")
 GUIDE: axis(dim(1.1), ticks(null()))
 GUIDE: axis(dim(2.1), ticks(null()))
 GUIDE: axis(dim(1), gap(0px))
 GUIDE: axis(dim(2), gap(0px))
 ELEMENT: point(position((
  workshop/workshop_label+q1/q1_label+
  q2/q2_label+q3/q3_label+q4/q4_label)*(
  workshop/workshop_label+q1/q1_label+
  q2/q2_label+q3/q3_label+q4/q4_label)))
END GPL.

* GPL statements for scatter plot matrix
* of workshop to q4 excluding gender.
* Gender cannot be used in this context.
GGRAPH
  /GRAPHDATASET NAME="graphdataset"
    VARIABLES=workshop q1 q2 q3 q4
  MISSING=LISTWISE REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: workshop=col(source(s), name("workshop"))
 DATA: q1=col(source(s), name("q1"))
 DATA: q2=col(source(s), name("q2"))
 DATA: q3=col(source(s), name("q3"))
 DATA: q4=col(source(s), name("q4"))
 TRANS: workshop_label = eval("workshop")
 TRANS: q1_label = eval("q1")
 TRANS: q2_label = eval("q2")
 TRANS: q3_label = eval("q3")
 TRANS: q4_label = eval("q4")
 GUIDE: axis(dim(1.1), ticks(null()))
 GUIDE: axis(dim(2.1), ticks(null()))
 GUIDE: axis(dim(1), gap(0px))
 GUIDE: axis(dim(2), gap(0px))
 ELEMENT: point(position((
  workshop/workshop_label+q1/q1_label+
  q2/q2_label+q3/q3_label+q4/q4_label)*(
  workshop/workshop_label+q1/q1_label+
  q2/q2_label+q3/q3_label+q4/q4_label)))
END GPL.
```

16.22.2 R Program for ggplot2

This program brings together the examples discussed in this chapter and a few variations that were not.

```
# Filename: GraphicsGG.R

setwd("c:/myRfolder")
load(file = "mydata100.Rdata")

# Get rid of missing values for facets
mydata100 <- na.omit(mydata100)
attach(mydata100)
library("ggplot2")

# ---Bar Plots---

# Bar plot - vertical

qplot(workshop, geom = "bar")

ggplot(mydata100, aes( workshop ) ) +
  geom_bar()

# Bar plot - horizontal

qplot(workshop, geom = "bar") + coord_flip()

ggplot(mydata100, aes(workshop) ) +
  geom_bar() + coord_flip()
```

```
# Bar plot - single bar stacked

qplot(factor(""), fill = workshop,
  geom = "bar", xlab = "") +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100,
  aes(factor(""), fill = workshop) ) +
  geom_bar() +
  scale_x_discrete("") +
  scale_fill_grey(start = 0, end = 1)

# Pie charts, same as stacked bar but polar coordinates

qplot(factor(""), fill = workshop,
  geom = "bar", xlab = "") +
  coord_polar(theta = "y") +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100,
  aes( factor(""), fill = workshop ) ) +
  geom_bar( width = 1 ) +
  scale_x_discrete("") +
  coord_polar(theta = "y") +
  scale_fill_grey(start = 0, end = 1)

# Bar Plots - Grouped

qplot(gender, geom = "bar",
    fill = workshop, position = "stack") +
  scale_fill_grey(start = 0, end = 1)
qplot(gender, geom = "bar",
    fill = workshop, position = "fill")  +
  scale_fill_grey(start = 0, end = 1)
qplot(gender, geom = "bar",
    fill = workshop, position = "dodge") +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100, aes(gender, fill = workshop) ) +
  geom_bar(position = "stack") +
  scale_fill_grey(start = 0, end = 1)
ggplot(mydata100, aes(gender, fill = workshop) ) +
  geom_bar(position = "fill") +
  scale_fill_grey(start = 0, end = 1)
```

```
ggplot(mydata100, aes(gender, fill = workshop ) ) +
  geom_bar(position = "dodge") +
  scale_fill_grey(start = 0, end = 1)

# Bar Plots - Faceted

qplot(workshop, geom = "bar", facets = gender ~ . )

ggplot(mydata100, aes(workshop) ) +
  geom_bar() + facet_grid(gender ~ . )

# Bar Plots - Pre-summarized data

qplot( factor( c(1, 2) ), c(40, 60), geom = "bar",
  xlab = "myGroup", ylab = "myMeasure")

myTemp <- data.frame(
  myGroup   = factor( c(1, 2) ),
  myMeasure = c(40, 60)
)
myTemp
ggplot(data = myTemp, aes(myGroup, myMeasure) ) +
  geom_bar()

# ---Dot Charts---

qplot(workshop, stat = "bin",
     facets = gender ~ . , geom = "point", size = I(4) ) +
  coord_flip()

# Same thing but suppressing legend a different way
qplot(workshop, stat = "bin",
     facets = gender ~ . , geom = "point", size = 4 ) +
  opts(legend.position = "none") +
  coord_flip()

ggplot(mydata100,
   aes(workshop, ..count.. ) ) +
   geom_point(stat = "bin", size = 4) + coord_flip()+
   facet_grid( gender ~ . )

# ---Adding Titles and Labels---

qplot(workshop, geom = "bar",
  main = "Workshop Attendance",
  xlab = "Statistics Package \nWorkshops")

ggplot(mydata100, aes(workshop, ..count..)) +
  geom_bar() +
  opts( title = "Workshop Attendance" ) +
  scale_x_discrete("Statistics Package \nWorkshops")

# Example not in text: labels of continuous scales.
ggplot(mydata100, aes(pretest,posttest ) ) +
  geom_point() +
  scale_x_continuous("Test Score Before Training") +
  scale_y_continuous("Test Score After Training")  +
  opts(title = "The Relationship is Linear")

# ---Histograms and Density Plots---

# Simle Histogram
qplot(posttest, geom = "histogram")
qplot(posttest, geom = c("histogram", "rug") ) #not shown

ggplot(mydata100, aes(posttest) ) +
  geom_histogram() + geom_rug() # not shown in text

# Histogram with more bars.
qplot(posttest, geom = "histogram", binwidth = 0.5)

ggplot(mydata100, aes(posttest) ) +
  geom_histogram(binwidth = 0.5)

# Density plot
qplot(posttest, geom = "density")

ggplot(mydata100, aes(posttest)) +
  geom_density()

# Histogram with density

qplot(data = mydata100, posttest, ..density..,
  geom = c("histogram", "density") )

ggplot(data=mydata100) +
  geom_histogram( aes(posttest, ..density..) ) +
  geom_density(   aes(posttest, ..density..) ) +
  geom_rug( aes(posttest) )
```

```
# Histogram - separate plots by group

qplot(posttest, geom = "histogram", facets = gender ~ . )

ggplot(mydata100, aes(posttest) ) +
  geom_histogram() + facet_grid( gender ~ . )

# Histogram with Stacked Bars

qplot(posttest, geom = "histogram", fill = gender) +
  scale_fill_grey(start = 0, end = 1)

ggplot(mydata100, aes(posttest, fill = gender) ) +
  geom_bar() +
  scale_fill_grey(start = 0, end = 1)


 # ---QQ Plots---
qplot(sample = posttest, stat = "qq")

ggplot(mydata100, aes(sample = posttest) ) +
  stat_qq()


# ---Strip Plots---

# Simple, but jitter too wide for our small data

qplot( factor(""), posttest, geom = "jitter", xlab = "")

ggplot(mydata100, aes(factor(""), posttest) ) +
  geom_jitter() +
  scale_x_discrete("")

# Again, with more narrow jitter

qplot( factor(""), posttest, data = mydata100, xlab = "",
  position = position_jitter(width = .02))

ggplot(mydata100, aes(factor(""), posttest) ) +
 geom_jitter(position = position_jitter(width = .02)) +
 scale_x_discrete("")

# Strip plot by group.
# First, the easy way, with too much jitter for our data:
```

```
qplot(workshop, posttest, geom = "jitter")

ggplot(mydata100, aes(workshop, posttest) ) +
  geom_jitter()

# Again, limiting the jitter for our small data set:

qplot(workshop, posttest, data = mydata100, xlab = "",
  position = position_jitter(width = .08) )

ggplot(mydata100, aes(workshop, posttest) ) +
  geom_jitter(position = position_jitter(width = .08) ) +
  scale_x_discrete("")


# ---Scatter Plots---

# Simple scatter Plot

qplot(pretest, posttest)
qplot(pretest, posttest, geom = "point")

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point()

# Scatter plot connecting points sorted on x.

qplot(pretest, posttest, geom = "line")

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_line()

# Scatter plot connecting points in data set order.

qplot(pretest, posttest, geom = "path")

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_path()

# Scatter plot with skinny histogram-like bars to X axis.

qplot(pretest,posttest,
  xend = pretest, yend = 50,
  geom = "segment")
```

```
ggplot(mydata100, aes(pretest, posttest) ) +
  geom_segment( aes(  pretest, posttest,
                 xend = pretest, yend = 50) )

# Scatter plot with jitter

# qplot without:
qplot(q1, q4)

# qplot with:
qplot(q1, q4, position =
  position_jitter(width = .3, height = .3) )

# ggplot without:
ggplot(mydata100, aes(x = q1, y = q2) ) +
 geom_point()

# ggplot with:
ggplot(mydata100, aes(x = q1, y = q2) ) +
  geom_point(position =
  position_jitter(width = .3,height = .3) )

# Scatter plot on large data sets

pretest2  <- round( rnorm( n = 5000, mean = 80, sd = 5) )
posttest2 <- round( pretest2 +
  rnorm( n = 5000, mean = 3, sd = 3) )
pretest2[pretest2   > 100] <- 100
posttest2[posttest2 > 100] <- 100
temp <- data.frame(pretest2, posttest2)

# Small, jittered, transparent points.

qplot(pretest2, posttest2, data = temp,
  geom = "jitter", size = I(2), alpha = I(0.15),
  position = position_jitter(width = 2) )

ggplot(temp, aes(pretest2, posttest2),
  size = 2, position = position_jitter(x = 2, y = 2) ) +
  geom_jitter(colour = alpha("black", 0.15) )

# Hexbin plots

# In qplot using default colors.
```

```
qplot(pretest2, posttest2, geom = "hex", bins = 30)

# This works too:
ggplot(temp, aes(pretest2, posttest2) )  +
  stat_binhex(bins = 30) +

# In ggplot, switching to greyscale.
ggplot(temp, aes(pretest2, posttest2) )  +
  geom_hex( bins = 30 ) +
  scale_fill_continuous(
    low = "grey80", high = "black")

# Using density contours and small points.

qplot(pretest2, posttest2, data = temp, size = I(1),
  geom = c("point","density2d"))

ggplot(temp, aes( x = pretest2, y = posttest2) ) +
 geom_point(size = 1) + geom_density2d()

# Density shading
ggplot(temp, aes( x = pretest2, y = posttest2) ) +
  stat_density2d(geom = "tile",
    aes(fill = ..density..), contour = FALSE) +
  scale_fill_continuous(
    low = "grey80", high = "black")

rm(pretest2,posttest2,temp)

# Scatter plot with regression line, 95% confidence intervals.

qplot(pretest, posttest,
  geom = c("point", "smooth"), method = lm )

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() + geom_smooth(method = lm)

# Scatter plot with regression line but NO confidence intervals.

qplot(pretest, posttest,
  geom = c("point", "smooth"),
  method = lm, se = FALSE )

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE)

# Scatter with x = y line

qplot(pretest, posttest,
  geom = c("point", "abline"),
  intercept = 0, slope = 1 )

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point() +
  geom_abline(intercept = 0, slope = 1)

# Scatter plot with different point shapes for each group.

qplot(pretest, posttest, shape = gender)

ggplot(mydata100, aes(pretest, posttest) ) +
  geom_point( aes(shape = gender ) )

# Scatter plot with regressions fit for each group.

qplot(pretest, posttest,
  geom   = c("smooth", "point"),
  method = "lm", shape = gender,
  linetype  = gender)

ggplot(mydata100,
  aes(pretest, posttest, shape = gender,
  linetype = gender) ) +
  geom_smooth(method = "lm") +
  geom_point()

# Scatter plot faceted for groups

qplot(pretest, posttest,
  geom   = c("smooth", "point"),
  method = "lm", shape = gender,
  facets = workshop ~ gender)

ggplot(mydata100,
  aes(pretest, posttest, shape = gender) ) +
  geom_smooth(method = "lm") + geom_point() +
  facet_grid(workshop ~ gender)

# Scatter plot with vertical or horizontal lines
```

```
qplot(pretest, posttest,
 geom = c("point", "vline", "hline"),
 xintercept = 75, yintercept = 75)

ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline(intercept = 75) +
  geom_hline(intercept = 75)

# Scatter plot with a set of vertical lines

qplot(pretest, posttest, type = "point") +
  geom_vline(xintercept = seq(from = 70,to = 80,by = 2) )

ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline(xintercept = seq(from = 70,to = 80,by = 2) )

ggplot(mydata100, aes(pretest, posttest)) +
  geom_point() +
  geom_vline(xintercept = 70:80)

# Scatter plotting text labels

qplot(pretest, posttest, geom = "text",
  label = rownames(mydata100) )

ggplot(mydata100,
  aes(pretest, posttest,
  label = rownames(mydata100) ) ) +
  geom_text()

# Scatter plot matrix

plotmatrix( mydata100[3:8] )

# Small points & lowess fit.
plotmatrix( mydata100[3:8], aes( size = 1 ) ) +
  geom_smooth()

# Shape and gender fits.
plotmatrix( mydata100[3:8],
  aes( shape = gender ) ) +
  geom_smooth(method = lm)
```

```
# ---Box Plots---

# Box plot of one variable

qplot(factor(""), posttest,
  geom = "boxplot", xlab = "")

ggplot(mydata100,
  aes(factor(""), posttest) ) +
  geom_boxplot() +
  scale_x_discrete("")

# Box plot by group

qplot(workshop, posttest, geom = "boxplot" )

ggplot(mydata100,
  aes(workshop, posttest) ) +
  geom_boxplot()

# Box plot by group with jitter

# Wide jitter

qplot(workshop, posttest,
  geom = c("boxplot", "jitter") )

ggplot(mydata100,
  aes(workshop, posttest )) +
  geom_boxplot() + geom_jitter()

# Narrow jitter

ggplot(mydata100,
 aes(workshop, posttest )) +
 geom_boxplot() +
 geom_jitter(position = position_jitter(width = .1))

# Box plot for two-way interaction.

qplot(workshop, posttest,
  geom = "boxplot", fill = gender ) +
  scale_fill_grey(start = 0, end = 1)
```

```
ggplot(mydata100,
  aes(workshop, posttest) ) +
  geom_boxplot( aes(fill = gender), colour = "grey50") +
  scale_fill_grey(start = 0, end = 1)

# Error bar plot

ggplot(mydata100,
  aes( as.numeric(workshop), posttest ) ) +
  geom_jitter(size = 1,
    position = position_jitter(width = .1) )  +
  stat_summary(fun.y = "mean",
    geom = "smooth", se = FALSE) +
  stat_summary(fun.data = "mean_cl_normal",
    geom = "errorbar", width = .2, size = 1)


# ---Geographic Maps---

library("maps")
library("ggplot2")
myStates <- map_data("state")
head(myStates)
myStates[ myStates$region == "new york", ]

qplot(long, lat, data = myStates, group = group,
  geom = "path", asp = 1)

ggplot(data = myStates, aes(long, lat, group = group) )+
  geom_path() +
  coord_map()

myArrests <- USArrests
head(myArrests)

myArrests$region <- tolower( rownames(USArrests) )
head(myArrests)

myBoth <- merge(
  myStates,
  myArrests, by = "region")
myBoth[1:4, c(1:5,8)]

myBoth <- myBoth[order(myBoth$order), ]
myBoth[1:4, c(1:5,8)]
```

```
qplot(long, lat, data = myBoth, group = group,
  fill = Assault, geom = "polygon", asp = 1) +
  scale_fill_continuous(low = "grey80", high = "black")

ggplot(data = myBoth,
    aes(long, lat, fill = Assault, group = group) )+
  geom_polygon() +
  coord_map() +
  scale_fill_continuous(low = "grey80", high = "black")


# ---Logarithmic Axes---

# Change the variables
qplot( log(pretest), log(posttest) )

ggplot(mydata100,
  aes( log(pretest), log(posttest) ) ) +
  geom_point()

# Change axis labels

qplot(pretest, posttest, log = "xy")

ggplot(mydata100, aes( x = pretest, y = posttest) ) +
  geom_point() + scale_x_log10() + scale_y_log10()

# Change axis scaling

qplot(pretest, posttest, data = mydata100)  +
  coord_trans(x = "log10", y = "log10")

ggplot(mydata100, aes( x = pretest, y = posttest) ) +
  geom_point() + coord_trans(x = "log10", y = "log10")

# ---Aspect Ratio---

# This forces x and y to be equal.
qplot(pretest, posttest) + coord_equal()

# This sets aspect ratio to height/width.
qplot(pretest, posttest) + coord_equal(ratio = 1/4)

#---Multiframe Plots: Barchart Example---
```

```
grid.newpage()  # clear page

# Sets up a 2 by 2 grid to plot into.
pushViewport( viewport(layout = grid.layout(2, 2) ) )

# Bar plot dodged in row 1, column 1.
myPlot <- ggplot(mydata100,
    aes(gender, fill = workshop) ) +
  geom_bar(position = "stack") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title = "position = stack " )
print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 1) )

# Bar plot stacked, in row 1, column 2.
myPlot <- ggplot(mydata100,
    aes(gender, fill = workshop) ) +
  geom_bar(position = "fill") +
  scale_fill_grey(start = 0, end = 1) +
  opts( title = "position = fill" )
print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 2) )

# Bar plot dodged, given frames,
# in row 2, columns 1 and 2.
myPlot <- ggplot(mydata100,
    aes(gender, fill = workshop) ) +
  geom_bar(position = "dodge")  +
  scale_fill_grey(start = 0, end = 1) +
  opts( title = "position = dodge" )
print(myPlot, vp = viewport(
  layout.pos.row = 2,
  layout.pos.col = 1:2) )

#---Multiframe Scatter Plots---

# Clears the page
grid.newpage()

# Sets up a 2 by 2 grid to plot into.
pushViewport( viewport(layout = grid.layout(2,2) ) )
```

```
# Scatter plot of points
myPlot <- qplot(pretest, posttest,main = "geom = point")
print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 1) )

myPlot <- qplot( pretest, posttest,
          geom = "line", main = "geom = line" )
print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 2) )

myPlot <- qplot( pretest, posttest,
          geom = "path", main = "geom = path" )
print(myPlot, vp = viewport(
  layout.pos.row = 2,
  layout.pos.col = 1) )

myPlot <- ggplot( mydata100, aes(pretest, posttest) ) +
          geom_segment( aes(x = pretest, y = posttest,
                        xend = pretest, yend = 58) ) +
          opts( title = "geom_segment example" )

print(myPlot,
  vp = viewport(layout.pos.row = 2, layout.pos.col = 2) )

# ---Multiframe Scatterplot for Jitter---
grid.newpage()
pushViewport( viewport(layout = grid.layout(1, 2) ) )

# Scatterplot without
myPlot <- qplot(q1, q4,
          main = "Likert Scale Without Jitter")
print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 1) )

myPlot <- qplot(q1, q4,
  position = position_jitter(width = .3, height = .3),
  main = "Likert Scale With Jitter")
print(myPlot, vp = viewport(
  layout.pos.row = 1,
  layout.pos.col = 2) )

# ---Detailed Comparison of qplot and ggplot---
```

```
qplot(pretest, posttest,
  geom = c("point", "smooth"), method = "lm" )

# Or ggplot with default settings:

ggplot(mydata100, aes(x = pretest, y = posttest) ) +
  geom_point() +
  geom_smooth(method = "lm")

# Or with all the defaults displayed:
ggplot() +
layer(
  data    = mydata100,
  mapping = aes(x = pretest, y = posttest),
  geom    = "point",
  stat    = "identity"
) +
layer(
  data    = mydata100,
  mapping = aes(x = pretest, y = posttest),
  geom    = "smooth",
  stat    = "smooth",
  method  = "lm"
) +
coord_cartesian()
```

17

Statistics

This chapter demonstrates some basic statistical methods. More importantly, it shows how even in the realm of fairly standard analyses, R differs sharply from the approach used by SAS and SPSS. Since this book is aimed at people who already know SAS or SPSS, I assume you are already familiar with most of these methods. I briefly list each test's goal and assumptions and how to get R to perform them. For more statistical coverage see Dalgaard's Introductory Statistics with R [16], or Venable and Ripley's much more advanced *Modern Applied Statistics with S* [65].

The examples in this chapter will use the mydata100 data set described in Sect. 1.7. To get things to fit well on these pages, I have set

```
options(linesize = 63)
```

You can use that if you want your output to match perfectly, but it is not necessary.

17.1 Scientific Notation

While SAS and SPSS tend to print their small probability values as 0.000, R often uses scientific notation. An example is 7.447e-5 which means 7.447×10^{-5}, or 0.00007447. When the number after the "e" is negative, you move the decimal place that many places to the left.

You may also see p-values of just "0." That value is controlled by the digits option, which is set to be seven significant digits by default. If you wanted to increase the number of digits to 10, you could do so with the following function call:

```
options(digits=10)
```

SAS has a similar option, `OPTIONS PROBSIG=10`, but it applies only to the p-values it prints. SPSS has a command that works more like R, `SET SMALL 0.0000000001`, which affects the printing of all values that contain decimals.

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_17, © Springer Science+Business Media, LLC 2011

In all three packages, setting the number of digits affects only their display. The full precision of p-values is always available when the numbers are stored.

Supplying a positive number to R's *sci*entific *pen*alty option `scipen` biases the printing away from scientific notation and more toward fixed notation. A negative number does the reverse. So if you want to completely block scientific notation, you can do so with the following funcion call:

```
options(scipen=999)
```

This is the equivalent of "`SET SMALL 0.`" in SPSS.

17.2 Descriptive Statistics

In SAS, you get frequencies from PROC FREQ and means, standard deviations, and the like from PROC MEANS or PROC UNIVARIATE.

SPSS allows you to get both at once with the FREQUENCIES command, although you can also get means and other descriptive statistics from the CONDESCRIPTIVE and EXAMINE commands.

R also has functions that handle categorical and continuous variables together or separately. Let us start with functions that are most like those of SAS or SPSS and then move on to functions that are more fundamental to R.

17.2.1 The `Deducer frequencies` Function

We discussed Fellow's `Deducer` package in the section on GUIs (Sect. 3.11.1). However, that package also includes some very useful functions such as `frequencies`. To use `Deducer`, you must install it. See Chap. 2, "Installing and Updating R." Then you must load it with either the *Packages> Load Packages* menu item or the function call

```
library("Deducer")
```

You use its `frequencies` function with a data frame or a individual variable. To save space, I will only show the output for workshop:

```
> frequencies(mydata100)

$workshop
------------------------------------------------------------
--                         Frequencies                    --
--                                                        --
  Value # of Cases       % Cumulative %
1     R         31    31.3         31.3
2   SAS         24    24.2         55.6
3  SPSS         25    25.3         80.8
```

```
4 Stata          19    19.2        100.0
--                                                          --
--                         Case Summary                     --
--                                                          --
           Vaild Missing Total
# of cases    99       1   100
--                                                          --
--                                                          --
------------------------------------------------------------
```

This is information we would get from SAS's FREQ procedure or SPSS's FREQUENCIES command `tab` command.

17.2.2 The `Hmisc` `describe` Function

Frank Harrell's `Hmisc` package [32] offers a wide selection of functions that have more comprehensive output than the standard R functions. One of these is the `describe` function. It is similar to the `summary` function we have used throughout this book. Before using the `describe` function, you must install `Hmisc`. See Chap. 2, "Installing and Updating R." Then you must load it with either the *Packages> Load Packages* menu item or the function call:

```
library("Hmisc")
```

You can select variables in many ways. See Chap. 7, "Selecting Variables," for details. One of the nicest features of the `describe` function is that it provides frequencies on nonfactors as well as factors, so long as they do not have too many values:

```
> describe(mydata100L)

mydata100L
8  Variables      100  Observations
-------------------------------------------------------------
gender
      n missing  unique
     99       1       2
Female (47, 47%), Male (52, 53%)
-------------------------------------------------------------
workshop
      n missing  unique
     99       1       4
R (31, 31%), SAS (24, 24%), SPSS (25, 25%), Stata (19, 19%)
-------------------------------------------------------------
q1 : The instructor was well prepared.
      n missing  unique    Mean
```

```
    100       0       5    3.45
          1  2  3  4  5
Frequency 4 14 36 25 21
%         4 14 36 25 21
-------------------------------------------------------------
q2 : The instructor communicated well.
      n missing  unique    Mean
    100       0       5    3.06
           1  2  3  4  5
Frequency 10 28 21 28 13
%         10 28 21 28 13
-------------------------------------------------------------
q3 : The course materials were helpful.
      n missing  unique    Mean
     99       1       5   3.081
           1  2  3  4  5
Frequency 10 20 34 22 13
%         10 20 34 22 13
-------------------------------------------------------------
q4 : Overall, I found this workshop useful.
      n missing  unique    Mean
    100       0       5     3.4
          1  2  3  4  5
Frequency 6 14 34 26 20
%         6 14 34 26 20
-------------------------------------------------------------
pretest
      n missing  unique    Mean     .05     .10     .25     .50
    100       0      23   74.97   66.95   69.00   72.00   75.00
    .75     .90     .95
  79.00   82.00   83.00
lowest : 58 62 63 66 67, highest: 82 83 84 85 86
-------------------------------------------------------------
posttest
      n missing  unique    Mean     .05     .10     .25     .50
    100       0      28   82.06   71.95   75.00   77.00   82.00
    .75     .90     .95
  86.00   90.00   92.00
lowest : 59 67 68 69 71, highest: 91 92 95 96 98
-------------------------------------------------------------
```

Unlike SAS and SPSS, the `describe` function does not provide percentages that include missing values. You can change that by setting the `exclude.missing` argument to FALSE. The `describe` function will automatically provide a table of frequencies whenever a variable has no more than

20 unique values. Beyond that, it will print the five largest and five smallest values, just like SAS's UNIVARIATE procedure and SPSS's EXPLORE procedure.

Notice that the survey questions themselves appear in the output. That is because this version of the data frame was prepared as described in Sect. 11.2, "Variable Labels," using the `Hmisc` package's approach.

17.2.3 The `summary` Function

R's built-in function for univariate statistics is `summary`. We have used the `summary` function extensively throughout this book, but I repeat its output here for comparison:

```
> summary(mydata100)

    gender    workshop        q1             q2
 Female:47   R    :31   Min.   :1.00   Min.   :1.00
 Male  :52   SAS  :24   1st Qu.:3.00   1st Qu.:2.00
 NA s  : 1  SPSS :25   Median :3.00   Median :3.00
             Stata:19   Mean   :3.45   Mean   :3.06
             NA s : 1  3rd Qu.:4.00   3rd Qu.:4.00
                        Max.   :5.00   Max.   :5.00

       q3              q4          pretest
 Min.   :1.000   Min.   :1.0   Min.   :58.00
 1st Qu.:2.000   1st Qu.:3.0   1st Qu.:72.00
 Median :3.000   Median :3.0   Median :75.00
 Mean   :3.081   Mean   :3.4   Mean   :74.97
 3rd Qu.:4.000   3rd Qu.:4.0   3rd Qu.:79.00
 Max.   :5.000   Max.   :5.0   Max.   :86.00
 NA s   :1.000

    posttest
 Min.   :59.00
 1st Qu.:77.00
 Median :82.00
 Mean   :82.06
 3rd Qu.:86.00
 Max.   :98.00
```

As you can see, it is much sparser, lacking percentages for factors, frequencies and percentages for numeric variables (even if they have a small number of values), number of nonmissing values, and so on. However, it is much more compact. The numbers labeled "1st Qu." and "3rd Qu." are the first and third quartiles, or the 25th and 75th percentiles, respectively.

Notice that the variable labels are now ignored. At the time of this writing, only the print function and the functions that come with the Hmisc package display variable labels created by the Hmisc label function. If we wanted to add value labels to the q variables, we would have to convert them to factors. The summary function works with a much wider range of objects, as we will soon see. The describe function works only with data frames, vectors, matrices, or formulas. For data frames, choose whichever of these two functions meets your needs.

17.2.4 The table Function and Its Relatives

Now let us review R's built-in functions for frequencies and proportions. We have covered them in earlier sections also, but I repeat them here for ease of comparison and elaboration.

R's built-in function for frequency counts provides output that is much sparser than those of the describe function:

```
> table(workshop)

workshop
    R   SAS  SPSS Stata
   31    24    25    19

> table(gender)

gender
Female   Male
    47     52
```

The above output is quite minimal, displaying only the frequencies. This sparsity makes it very easy to use this output as input to other functions such as barplot; see Chap. 15, "Traditional Graphics," for examples.

We can get proportions by using the prop.table function:

```
> prop.table( table(workshop) )

workshop

        R       SAS      SPSS     Stata
0.3131313 0.2424242 0.2525253 0.1919192

> prop.table( table(gender) )

gender

   Female      Male
0.4747475 0.5252525
```

You can round off the proportions using the round function. The only arguments you need are the object to round and the number of decimals you would like to keep:

```
> round( prop.table( table(gender) ), 2 )

gender

Female   Male
  0.47   0.53
```

Converting that to percentages is, of course, just a matter of multiplying by 100. If you multiply before rounding, you will not even need to specify the number of decimals to keep since the default is to round to whole numbers:

```
> round( 100* ( prop.table( table(gender) ) ) )

gender

Female   Male
    47     53
```

When examining test scores, it is often helpful to get cumulative proportions. R does this by adding the cumsum function that *cum*ulatively *sum*s a variable. Here we apply it to the output of prop.table to sum its proportions.

```
> cumsum( prop.table( table(posttest) ) )

  59   67   68   69   71   72   74   75   76   77   78
0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.13 0.18 0.26 0.31

  79   80   81   82   83   84   85   86   87   88   89
0.35 0.39 0.48 0.53 0.58 0.65 0.68 0.76 0.79 0.84 0.87

  90   91   92   95   96   98
0.92 0.93 0.96 0.98 0.99 1.00
```

We can see that 0.92 of the students received a score of 90 or lower.

You can easily combine just the pieces you need into your own function. For example, here I add the original values, frequencies, proportions, and cumulative proportions into a data frame:

```
myTable <- function(score)  {
  myDF <- data.frame( table(score) )
  myDF$Prop <- prop.table( myDF$Freq )
  myDF$CumProp <-  cumsum( myDF$Prop )
  myDF
 }
```

Here is what the output looks like:

```
> myTable(posttest)

   score Freq Prop CumProp
1     59    1 0.01    0.01
2     67    1 0.01    0.02
3     68    1 0.01    0.03
...
26    95    2 0.02    0.98
27    96    1 0.01    0.99
28    98    1 0.01    1.00
```

Hilbe has added this function to his COUNT package [30].

A word of caution about the table function. Unlike the summary function, if you use it on a whole data frame, it will not give you all one-way frequency tables. Instead, it will cross-tabulate all of the variables at once. You can use it on a surprising number of factors at once. When you convert its output to a data frame, you have a concise list of counts for all possible combinations. For an example, see Sect. 10.12.4, "Tabular Aggregation."

17.2.5 The mean Function and Its Relatives

R's built-in functions offers similarly sparse output for univariate statistics. To get just the means of variables q1 through posttest (variables 3 through 8), we can use

```
> sapply( mydata100[3:8], mean,  na.rm = TRUE)

      q1       q2       q3       q4  pretest posttest
  3.4500   3.0600   3.0808   3.4000  74.9700  82.0600
Similarly, for the standard deviations:

> sapply( mydata100[3:8], sd,    na.rm = TRUE)

      q1       q2       q3       q4  pretest posttest
  1.0952   1.2212   1.1665   1.1371   5.2962   6.5902
```

You can also substitute the var function for variance or the median function for that statistic. You can apply several of these functions at once by combining them into your own single function. For an example of that, see Sect. 5.9 "Writing Your Own Functions (Macros)." For details about the sapply function, see Sect. 10.2, "Procedures or Functions? The Apply Function Decides."

17.3 Cross-Tabulation

You can compare groups on categorical measures with the chi-squared test, for example, testing to see if males and females attended the various workshops in the same proportions. In SAS, you would use PROC FREQ and in SPSS the CROSSTABS or procedure, perhaps in conjunction with the SPSS Exact Tests module.

Assumptions:

No more than 20% of the cells in your cross-tabulation have counts fewer than 5. If you have sparse tables, the exact `fisher.test` function is more appropriate.

Observations are independent. For example, if you measured the same subjects repeatedly, it would be important to take that into account in a more complex model.

The variables are not the same thing measured at two times. If that is the case, the `mcnemar.test` function may be what you need.

17.3.1 The `CrossTable` Function

To get output most like that from SAS and SPSS, we will first use the functions from Warnes' `gmodels` package [68]. Then we will cover the cross-tabulation functions that are built into R. To use `gmodels`, you must first install it. See Chap. 2, "Installing and Updating R." Then you must load it with either the *Packages> Load Packages* menu item or the function call

```
library("gmodels")
```

Now you are ready to use the `CrossTable` function:

```
> CrossTable(workshop, gender,
+   chisq = TRUE, format = "SAS")

   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|
Total Observations in Table:  99
             | gender
    workshop |    Female |      Male | Row Total |
-------------|-----------|-----------|-----------|
           R |        14 |        17 |        31 |
             |     0.035 |     0.032 |           |
             |     0.452 |     0.548 |     0.313 |
             |     0.298 |     0.327 |           |
             |     0.141 |     0.172 |           |
-------------|-----------|-----------|-----------|
         SAS |        11 |        13 |        24 |
             |     0.014 |     0.012 |           |
             |     0.458 |     0.542 |     0.242 |
             |     0.234 |     0.250 |           |
             |     0.111 |     0.131 |           |
-------------|-----------|-----------|-----------|
        SPSS |        13 |        12 |        25 |
             |     0.108 |     0.097 |           |
             |     0.520 |     0.480 |     0.253 |
             |     0.277 |     0.231 |           |
             |     0.131 |     0.121 |           |
-------------|-----------|-----------|-----------|
       Stata |         9 |        10 |        19 |
             |     0.000 |     0.000 |           |
             |     0.474 |     0.526 |     0.192 |
             |     0.191 |     0.192 |           |
             |     0.091 |     0.101 |           |
-------------|-----------|-----------|-----------|
Column Total |        47 |        52 |        99 |
             |     0.475 |     0.525 |           |
-------------|-----------|-----------|-----------|
Statistics for All Table Factors
Pearson s Chi-squared test
------------------------------------------------------------
Chi^2 =  0.2978553     d.f. =  3     p =  0.9604313
```

The `CrossTable` function call above used three arguments.

1. The variable that determines the table rows.
2. The variable that determines the table columns.
3. The `chisq = TRUE` argument, which tells R to perform that test. As with SAS and SPSS, if you leave this argument out, it will perform the cross-tabulation, but not the chi-squared test.
4. The `format="SAS"` argument, which tells it to create a table as SAS would. That is the default so you do not need to list it if that is what you want. Placing `format="SPSS"` here would result in a table in the style of SPSS.

17.3.2 The `table` and `chisq.test` Functions

R also has built-in functions to do cross-tabulation and the chi-squared test. As usual, the built-in functions present sparse results. We will first use the `table` function. To simplify the coding and to demonstrate a new type of data structure, we will save the table and name it myWG for workshop and gender:

```
> myWG <- table(workshop, gender)
```

Printing myWG will show us that it contains the form of counts to which SAS and SPSS users are accustomed.

```
> myWG

        gender

workshop Female Male
   R         14   17
   SAS       11   13
   SPSS      13   12
   Stata      9   10
```

You may recall from our discussion of factors that you can create factor levels (and their labels) that do not exist in your data. That would help if you were to enter more data later that is likely to contain those values or if you were to merge your data frame with others that had a full set of values. However, when performing a cross-tabulation, the levels with zero values will become part of the table. These empty cells will affect the resulting chi-squared statistic, which will drastically change its value. To get rid of the unused levels, append `[ ,drop = TRUE]` to the variable reference, for example,

```
myWG <- table( workshop[ , drop = TRUE], gender)
```

Some R functions work better with this type of data in a data frame. You probably associate this style of tabular data with output from the SAS SUMMARY procedure or SPSS AGGREGATE command. The `as.data.frame` function can provide it:

```
> myWGdata <- as.data.frame(myWG)

> myWGdata

  workshop gender Freq
1        R Female   14
2      SAS Female   11
3     SPSS Female   13
4    Stata Female    9
5        R   Male   17
6      SAS   Male   13
7     SPSS   Male   12
8    Stata   Male   10
```

The functions we discuss now work well on table objects, so we will use myWG. We can use the `chisq.test` function to perform the chi-squared test:

```
> chisq.test(myWG)

        Pearson s Chi-squared test

data:  myWG
X-squared = 0.2979, df = 3, p-value = 0.9604
```

The `table` function does not calculate any percents or proportions. To get row or column proportions, we can use the `prop.table` function. The arguments in the example below are the table and the margin to analyze where 1=row and 2=column. So this will calculate row proportions:

```
> prop.table(myWG, 1)

        gender

workshop  Female    Male
   R     0.45161 0.54839
   SAS   0.45833 0.54167
   SPSS  0.52000 0.48000
   Stata 0.47368 0.52632
```

Similarly, changing the “1” to “2” requests column proportions:

```
> prop.table(myWG, 2)

        gender

workshop  Female    Male
   R     0.29787 0.32692
   SAS   0.23404 0.25000
   SPSS  0.27660 0.23077
   Stata 0.19149 0.19231
```

If you do not provide the `margin` argument, the function will calculate total proportions:

```
> prop.table(myWG)

        gender

workshop  Female    Male
   R     0.14141 0.17172
   SAS   0.11111 0.13131
   SPSS  0.13131 0.12121
   Stata 0.09091 0.10101
```

The round function will round off unneeded digits by telling it how many decimal places you want – in this case, 2. These are the row proportions:

```
> round( prop.table(myWG, 1), 2 )

        gender

workshop Female Male
   R       0.45 0.55
   SAS     0.46 0.54
   SPSS    0.52 0.48
   Stata   0.47 0.53
```

To convert proportions to percentages, multiply by 100 and round off. If you want to round to the nearest whole percentage, multiply by 100 before rounding off. That way you do not even have to tell the round function how many decimal places to keep as its default is to round off to whole numbers:

```
> round( 100* ( prop.table(myWG, 1) ) )

        gender

workshop Female Male
   R         45   55
   SAS       46   54
   SPSS      52   48
   Stata     47   53
```

If you wish to add marginal totals, the addmargins function will do so. It works much like the prop.table function, in that its second argument is a 1 to add a row with totals or 2 to add a column.

```
> addmargins(myWG, 1)

        gender

workshop Female Male
   R         14   17
   SAS       11   13
   SPSS      13   12
   Stata      9   10
   Sum       47   52
```

If you do not specify a preference for row or column totals, you will get both:

```
> addmargins(myWG)

        gender

workshop Female Male Sum
   R         14   17  31
   SAS       11   13  24
   SPSS      13   12  25
   Stata      9   10  19
   Sum       47   52  99
```

17.4 Correlation

Correlations measure the strength of linear association between two continuous variables. SAS calculates them with PROC CORR for both parametric and nonparametric correlations. SPSS uses the CORRELATIONS procedure for parametric and NONPAR CORR procedure for nonparametric correlations. R has both built-in functions to calculate correlations and, of course, more functions to do so in add-on packages.

Assumptions:

1. Scatter plots of the variables shows essentially a straight line. The function `plot(x,y)` does the scatter plots. If you have a curve, transformations such as square roots or logarithms often help.
2. The spread in the data is the same at low, medium, and high values. Transformations often help with this assumption also.
3. For a Pearson correlation, the data should be at least interval level and normally distributed. As discussed in Chap. 15, "Traditional Graphics," `hist(myvar)` or `qqnorm(myvar)` is a quick way to examine the data. If your data are not normally distributed or are just ordinal measures (e.g., low, medium, high), you can use the nonparametric Spearman or the (less popular) Kendall correlation.

To do correlations, we will use the `rcorr.adjust` function from the `Rcmdr` package discussed in Sect. 3.11.2. The history of this function is a good example of why R is growing so rapidly. For over a decade I had been asking the developers of SAS and SPSS to add an option to correct their p-values for the effects of multiple testing. When I started using R, I came across the built-in `p.adjust` function that corrects a vector of p-values from any tests. I liked the way the `Hmisc` package's `rcorr` function did correlations, so I wrote a function to add `p.adjust` to it. I sent that off to John Fox, the developer of R Commander. He improved the code and added it to the package. With just a few hours of work and the efforts of several people, every R user now has access to a useful feature.

Before using the `Rcmdr` package, you must install it. (See Chap. 2, ”Installing and Updating R.”) Then you must load it with either the *Packages> Load Packages* menu item or the function call

```
library("Rcmdr")
```

The graphical user interface will start up, but we will just use programming code to run it. To correlate variables, supply them in the form of either a data frame or a matrix. To better demonstrate the impact of correcting *p*-values, I will use the smaller mydata data set:

```
> load("mydata.RData")

> rcorr.adjust( mydata[3:6] )

      q1    q2    q3    q4
q1  1.00  0.78 -0.12  0.88
q2  0.78  1.00 -0.27  0.90
q3 -0.12 -0.27  1.00 -0.03
q4  0.88  0.90 -0.03  1.00

n= 7

P
   q1     q2     q3     q4
q1        0.0385 0.7894 0.0090
q2 0.0385        0.5581 0.0053
q3 0.7894 0.5581        0.9556
q4 0.0090 0.0053 0.9556

 Adjusted p-values (Holm s method)
   q1     q2     q3     q4
q1        0.1542 1.0000 0.0450
q2 0.1542        1.0000 0.0318
q3 1.0000 1.0000        1.0000
q4 0.0450 0.0318 1.0000
```

The first set of output are the Pearson correlations themselves. We can see that the correlation of q1 and q2 is 0.78.

Next, it reports the number of valid observations: n=7. The function filters out missing values in a listwise fashion. The `rcorr` function in the `Hmisc` package does pairwise deletion, and that addition is planned for `rcorr.adjust`.

After that, we see the *p*-values. These are the same as you would get from SAS or SPSS using listwise deletion of missing values. From that set the correlation of q1 and q2 has a *p*-value of 0.0385, well under the popular significance level of 0.05.

Finally, we see the adjusted *p*-values. These take into account the number of tests we have done and corrects for them using Holm's method. Other adjustment methods are available. See `help("rcorr.adjust")` and `help("p.adjust")` for details. The adjusted *p*-value for the correlation of q1 and q2 is 0.1542! That is the more realistic value.

You add the `type="spearman"` argument to get correlations on variables that are not normally distributed or are only ordinal in scale (e.g., low, medium, high):

```
> rcorr.adjust( mydata[3:6], type="spearman" )

      q1    q2    q3   q4
q1  1.00  0.66 -0.04 0.82
q2  0.66  1.00 -0.08 0.88
q3 -0.04 -0.08  1.00 0.26
q4  0.82  0.88  0.26 1.00

n= 7


P
   q1     q2     q3     q4
q1        0.1041 0.9327 0.0237
q2 0.1041        0.8670 0.0092
q3 0.9327 0.8670        0.5768
q4 0.0237 0.0092 0.5768

 Adjusted p-values (Holm s method)
   q1     q2     q3     q4
q1        0.4165 1.0000 0.1183
q2 0.4165        1.0000 0.0554
q3 1.0000 1.0000        1.0000
q4 0.1183 0.0554 1.0000
```

17.4.1 The `cor` Function

Now let us take a look at R's built-in functions for calculating correlations. As usual, they provide more sparse output:

```
> cor( mydata[3:6],
+   method = "pearson", use = "pairwise")
           q1         q2          q3          q4
q1  1.0000000  0.7395179 -0.12500000  0.88040627
q2  0.7395179  1.0000000 -0.27003086  0.85063978
q3 -0.1250000 -0.2700309  1.00000000 -0.02613542
q4  0.8804063  0.8506398 -0.02613542  1.00000000
```

The cor function call above uses three arguments.

1. The variables to correlate. This can be a pair of vectors, a matrix, or a data frame. If the first argument is a vector, then the next argument must be another vector with which to correlate. You can label them x= and y= or just put them in the first two positions.
2. The method argument can be pearson, spearman, or kendall for those types of correlations. (Be careful not to capitalize these names when used with arguments in R.)
3. The use argument determines how the function will deal with missing data. The value pairwise.complete, abbreviated pairwise above, uses as much data as possible. That is the default approach in SAS and SPSS. The value complete.obs is the SAS/SPSS equivalent of listwise deletion of missing values. This tosses out cases that have any missing values for the variables analyzed. If each variable has just a few missing values, but the values that are missing involve different cases, you can lose a very large percentage of your data with this option. However, it does ensure that every correlation is done on the exact same cases. That can be important if you plan to use the correlation matrix in additional computations. As usual, by default R provides no results if it finds missing values. That is the use=all.obs setting. So if you omit the use argument and have missing values, cor will print only an error message telling you that it has found missing values.

Unlike the Rcmdr package's rcorr.adjust function, the built-in cor function provides only a correlation matrix. Another built-in function, cor.test, provides comprehensive output but only for two variables at a time.

```
> cor.test(mydata$q1, mydata$q2, use="pairwise")

        Pearson s product-moment correlation

data:  mydata$q1 and mydata$q2

t = 2.691, df = 6, p-value = 0.036

alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.0727632 0.9494271

sample estimates:
      cor
0.7395179
```

Note here that if you run cor.test multiple times it is up to you to correct your p-values for the number of tests you do. You can do that manually using the `p.adjust` function, but in the case of correlations it is much easier to use `rcorr.adjust` in the `Rcmdr` package as shown previously.

17.5 Linear Regression

Linear regression models the linear association between one continuous dependent variable and a set of continuous independent or predictor variables. SAS performs linear regression with PROC REG, among others. SPSS uses the REGRESSION procedure, and others.

Assumptions:

Scatter plots of the dependent variable with each independent variable shows essentially a straight line. The `plot(x,y)` function call does the scatter plots. If you have a curve, transformations such as square roots or logarithms often help straighten it out.

The spread in the data is the same at low, medium, and high values. This is called homoscedasticity. Transformations often help with this requirement also.

The model residuals (difference between the predicted values and the actual values) are normally distributed. We will use the `plot` function to generate a normal QQ plot to test this assumption.

The model residuals are independent. If they contain a relationship, such as the same subjects measured through time or classes of subjects sharing the same teacher, you would want to use a more complex model to take that into account.

When performing a single type of analysis in SAS or SPSS, you prepare your two or three commands and then submit them to get all your results at once. You could save some of the output using the SAS Output Delivery System (ODS) or SPSS Output Management System (OMS) , but that is not required. R, on the other hand, shows you very little output with each command, and everyone uses its integrated output management capabilities.

Let us look at a simple example. First, we will use the `lm` function to do a *l*inear *m*odel predicting the values of q4 from the survey questions. Although this type of data is viewed as ordinal scale by many, social scientists often view it as interval-level. With a more realistic data set, we would be working with a scale for each measure that consisted of the means of several questions, resulting in far more than just five values. The first argument to `lm` is the formula. Table 17.1 shows formulas for various types of models. The following `lm` function call includes the formula for a linear regression that includes no interactions:

```
> lm(q4 ~ q1 + q2 + q3, data = mydata100)

Call:
```

```
lm(formula = q4 ~ q1 + q2 + q3, data = mydata100)

Coefficients:

 (Intercept)           q1           q2           q3
    1.20940      0.41134      0.15791      0.09372
```

We see that the results provide only the coefficients to the linear regression model. So the model is as follows:

$$Predictedq4 = 1.20940 + 0.41134 \times q1 + 0.15791 \times q2 + 0.09372 \times q3$$

The other results that SAS or SPSS would provide, such as R-squared or tests of significance, are not displayed. Now we will run the model again and save its results in an object called myModel.

```
> myModel <- lm(q4 ~ q1 + q2 + q3, data = mydata100)
```

This time, no printed results appear. We can see the contents of myModel by entering its name just like any other R object:

```
> myModel

Call:

lm(formula = q4 ~ q1 + q2 + q3, data = mydata100)

Coefficients:

 (Intercept)           q1           q2           q3
    1.20940      0.41134      0.15791      0.09372
```

The above results are exactly the same as we saw initially. So what type of object is myModel? The mode and class functions can tell us:

```
> mode(myModel)

 [1] "list"

> class(myModel)

 [1] "lm"
```

So we see the `lm` function saved our model as a list with a class of lm. Now that we have the model stored, we can apply a series of *extractor functions* to get much more information. Each of these functions will see the class of lm and will apply the methods that it has available for that class of object.

We have used the summary function previously. With our data frame object, mydata100, the `summary` function “knew” to get frequency counts on factors and other measures, like means, on continuous variables. The following code shows what it will do with lm objects:

```
> summary(myModel)

Call:

lm(formula = q4 ~ q1 + q2 + q3, data = mydata100)

Residuals:

Overall, I found this workshop useful.
    Min      1Q  Median      3Q     Max
-1.9467 -0.6418  0.1175  0.5960  2.0533

Coefficients:

            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.20940    0.31787   3.805 0.000251 ***
q1           0.41134    0.13170   3.123 0.002370 **
q2           0.15791    0.10690   1.477 0.142942
q3           0.09372    0.11617   0.807 0.421838

---
Signif. codes:  0  ***  0.001  **  0.01  *  0.05  .  0.1    1
Residual standard error: 0.9308 on 95 degrees of freedom
  (1 observation deleted due to missingness)

Multiple R-squared: 0.3561,     Adjusted R-squared: 0.3358

F-statistic: 17.51 on 3 and 95 DF,  p-value: 3.944e-09
```

That is much the same output we would get from SAS or SPSS. The main difference is the summary of model residuals. The t-tests on the model parameters are partial tests of each parameter, conditional on the other independent variables being in the model. These are sometimes called type III tests.

From the perspective of this table it might appear that neither q2 nor q3 is adding significantly to the model. However, the next table will provide a different perspective. The very small p-value of 3.944e-09, or 0.000000003944, is quite a bit smaller than 0.05, so we would reject the hypothesis that the overall model is worthless. The significant p-value of 0.002370 for q1 makes it the only significant predictor, given the other variables in the model. We will test that below.

Table 17.1. Example formulas in SAS, SPSS, and R. The x and y variables are continuous and the a, b, c, and d variables are factors

Model	R	SAS	SPSS
Simple regression	`y~x`	`MODEL y = x;`	`/DESIGN x.`
Regression with interaction	`y ~ x1+x2+x1:x2`	`MODEL y = x1 x2 x1*x2;`	`/DESIGN x1 x2 x1*x2.`

Regression without intercept	`y ~ -1 + x`	`MODEL y = x1/noint;`	`/DESIGN x.`
			`/INTERCEPT=EXCLUDE.`

Regression of y with all other variables	`y ~ .`	`MODEL y = _numeric_;`	
1-way analysis of variance	`y ~ a`	`MODEL y = a;`	`/DESIGN a.`

2-way analysis of variance with interaction	`y~a+b+a:b or y~a*b`	`MODEL y = a b a*b;`
		`MODEL y = a|b;`
			`/DESIGN a b a*b.`

4-way analysis of variance with all interactions	`y ~ a*b*c*d`	`MODEL y = a|b|c|d;`	`/DESIGN`
4-way analysis of variance with 2-way interactions	`y ~ (a+b+c+d)^2`	`MODEL y = a|b|c|d@2;`	
Analysis of covariance	`y ~ a x`	`MODEL y = a x;`	`/DESIGN a x.`

Analysis of variance with b nested within a	`y ~ b \%in\% a`
	`y ~ a/b`
		`MODEL y = b(a);`	`/DESIGN a b(a).`
			`/DESIGN a b WITHIN a.`

You can ask for an analysis of variance (ANOVA) table with the `anova` function:

```
> anova(myModel)

Analysis of Variance Table
Response: q4

          Df Sum Sq Mean Sq F value    Pr(>F)
q1         1 42.306  42.306 48.8278 3.824e-10 ***
q2         1  2.657   2.657  3.0661   0.08317 .
q3         1  0.564   0.564  0.6508   0.42184
Residuals 95 82.312   0.866
---

Signif. codes:  0  ***  0.001  **  0.01  *  0.05  .  0.1     1
```

These tests are what SAS and SPSS would call sequential or Type I tests. So the first test is for q1 by itself. The second is for q2, given that q1 is already in the model. The third is for q3, given that q1 and q2 are already in the model. Changing the order of the variables in the `lm` function would change these results. From this perspective, q1 is even more significant.

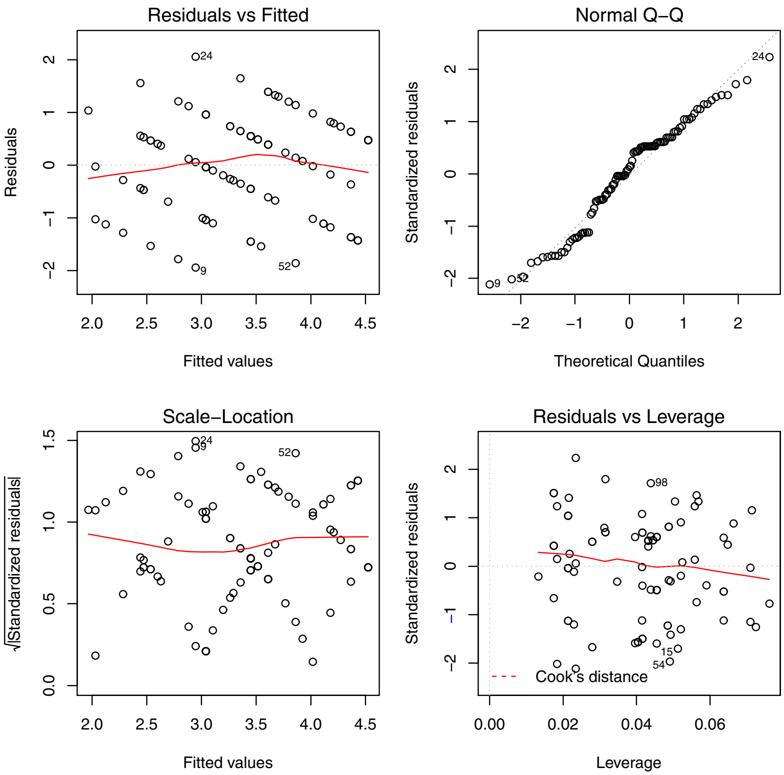


Fig. 17.1. Linear regression diagnostic plots generated automatically by the `plot` function. The odd strips of points are the result of the dependent variable having only five values

17.5.1 Plotting Diagnostics

The `plot` function also has methods for lm class objects. The single call to the `plot` function below was sufficient to generate all four of the plots shown in Fig. 17.1:

```
> plot(myModel)
```

The residuals vs. fitted plot shown in Fig. 17.1 (upper left), shows the fitted values plotted against the model residuals. If the residuals follow any particular pattern, such as a diagonal line, there may be other predictors not yet in the model that could improve it. The fairly flat lowess line looks good. The five strips of points shown result from our dependent variable having only five values.

The normal Q-Q Plot in Fig. 17.1 (upper right), shows the quantiles of the standardized residuals plotted against the quantiles you would expect if the data were normally distributed. Since these fall mostly on the straight line, the assumption of normally distributed residuals is met.

The scale-location plot in Fig. 17.1 (lower left), shows the square root of the absolute standardized residuals plotted against the fitted, or predicted, values. Since the lowess line that fits this is fairly flat, it indicates that the spread in the predictions is roughly the same across the prediction line, meeting the assumption of homoscedasticity.

Finally, the residuals vs. leverage plot in Fig. 17.1 (lower right), shows a measure of the influence of each point on the overall equation against the standardized residuals. Since no points stand out far from the pack, we can assume that there are no outliers having undue influence on the fit of the model.

17.5.2 Comparing Models

To do a stepwise model selection, first create the full model and save it, and then apply one of the following functions: `step`, `add1`, `drop1`, or, from Venable and Ripley's `MASS` package, `stepAIC`. `MASS` is named after their book *Modern Applied Statistics with S* [65]. Keep in mind that if you start with a large number of variables, stepwise methods make your p-values essentially worthless. If you can choose a model on part of your data and see that it still works on the remainder, then the p-values obtained from the remainder data should be valid.

The `anova` function can also compare two models, as long as one is a subset of the other. So to see if q2 and q3 combined added significantly to a model over q1, I created the two models and then compared them. One catch with this scenario is that with missing values present, the two models will be calculated on different sample sizes and R will not be able to compare them. It will warn you, "Models were not all fitted to the same size of data set." To deal with this problem, we will use the `na.omit` function discussed in Sect. 10.5 to remove the missing values first.

```
> myNoMissing <- na.omit(
+    mydata100[ , c("q1","q2","q3","q4") ] )

> myFullModel    <- lm( q4 ~ q1 + q2 + q3, data = myNoMissing)

> myReducedModel <- lm( q4 ~ q1,           data = myNoMissing)

> anova( myReducedModel, myFullModel)

Analysis of Variance Table

Model 1: q4 ~ q1
Model 2: q4 ~ q1 + q2 + q3
```

```
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     97 85.532
2     95 82.312  2     3.220 1.8585 0.1615
```

The *p*-value of 0.1615 tells us that variables q2 and q3 combined did not add significantly to the model that had only q1. When you compare two models using the `anova` function, you should list the reduced model first.

17.5.3 Making Predictions with New Data

You can apply a saved model to a new set of data using the `predict` function. Of course, the new data set must have the same variables, with the same names and the same types. Under those conditions, you can apply myModel to a data frame called myNewData using

```
myPredictions <- predict(myModel, myNewData)
```

Because the variables must have the *exact* same names, it is very important not to use the $ format in names. The following two statements perform what is statistically the same model; however, the one using mydata100$q4 will only work on future data frames named "mydata"!

```
myModel <- lm( mydata100$q4 ~ mydata$q1 )  # Bad idea
myModel <- lm(q4 ~ q1, data = mydata100)   # Much better
```

17.6 t-Test: Independent Groups

A t-test for independent groups compares two groups at a time on the mean of a continuous measure. SAS uses PROC TTEST for this and SPSS uses the T-TEST procedure.

The assumptions for t-tests are as follows:

The measure is interval-level continuous data. If the data are only ordinal (e.g., low, medium, high), consider the Wilcoxon rank sum test, also known as the Mann–Whitney test.
The measures are normally distributed. You can examine that with `hist(myVar)` or `qqnorm(myvar)` as shown in Chap. 15, "Traditional Graphics." If they are not, consider the Mann–Whitney or Wilcoxon test. For details, see Sect. 17.9.
The observations are independent. For example, if you measured the same subjects repeatedly, it would be important to take that into account in a more complex model.
The distributions should have roughly the same variance. See Sect. 17.7 for details.

You can perform t-tests in R using the `t.test` function:

```
> t.test( q1 ~ gender, data = mydata100)

        Welch Two Sample t-test

data:  q1 by gender

t = 5.0578, df = 96.938, p-value = 2.008e-06

alternative hypothesis:
true difference in means is not equal to 0

95 percent confidence interval:
 0.6063427 1.3895656

sample estimates:

mean in group Female   mean in group Male
            3.978723             2.980769
```

The `t.test` function call above has two arguments, the formula and the data frame to use. The formula `q4~gender` is in the form dependent~independent. For details, see Sect. 5.7.3, "Controlling Functions with Formulas."

Instead of a formula, you can specify two variables to compare such as `t.test(x,y)`, and they can be selected using any of R's many variable selection approaches. The `data` argument specifies the data frame to use only in the case of a formula. So you might think that this form works:

```
t.test( q4[which(gender ==  m ) ],
        q4[which(gender ==  f ) ] , data = mydata100)
```

However, unless the data are attached, it does not! You would have to enter `attach(mydata100)` before the command above would work. Alternatively, with an unattached data frame, you could leave off the `data` argument and use the form

```
with(mydata100,
  t.test( q4[ which(gender ==  m ) ],
          q4[ which(gender ==  f ) ] )
)
```

or, you might prefer the `subset` function:

```
t.test(
  subset(mydata100, gender == "m", select = q4),
  subset(mydata100, gender == "f", select = q4)
)
```

For more details see Chap. 9, “Selecting Variables and Observations.”

The results show that the mean for the females is 3.96 and for the males is 2.98. The *p*-value of 2.748e-06, or 0.000002748, is much smaller than 0.05, so we would reject the hypothesis that those means are the same.

Unlike SAS and SPSS, R does not provide a test for homogeneity of variance and two t-test calculations for equal and unequal variances. Instead, it provides only the unequal variance test by default. The additional argument, `var.equal = TRUE`, will ask it to perform the other test. See Sect. 17.7 for that topic.

17.7 Equality of Variance

SAS and SPSS offer tests for equality (homogeneity) of variance in their t-test and analysis of variance procedures. R, in keeping with its minimalist perspective, offers such tests in separate functions.

The Levene test for equality of variance is the most popular. Fox’s `car` package [21] contains the `levene.test` function. To use the `car` package you must first install it. See Chap. 2, ”Installing and Updating R.” Then you must load it with either the *Packages> Load Packages* menu item or the function call:

```
> library("car")
```

Now we are ready to use the `levene.test` function:

```
> levene.test(posttest, gender)

Levene s Test for Homogeneity of Variance

      Df F value Pr(>F)
group  1  0.4308 0.5131
      97
```

The `levene.test` function has only two arguments of the form (var, group). Its null hypothesis is that your groups have equal variances. If its *p*-value is smaller than 0.05, you reject that hypothesis. So in the above case, we would accept the hypothesis that the variances are equal.

If you run this same test in SPSS, you will get a somewhat different answer. SPSS bases its test on deviations from the mean, while the `car` package’s implementation used here uses deviations from the median.

Other tests for comparing variances are R’s built-in `var.test` and `bartlett.test` functions.

17.8 t-Test: Paired or Repeated Measures

The goal of a paired t-test is to compare the mean of two correlated measures. These are often the same measure taken on the same subjects at two different times. SAS and SPSS use the same procedures for paired t-tests as for independent samples t-tests, as does R.

The paired t-test's assumptions are as follows:

The two measures are interval-level continuous data. If the data are only ordinal (e.g., low-medium-high) consider the Wilcoxon signed-rank test. For details see Sect. 17.10, "Wilcoxon Signed-Rank Test: Paired Groups."

The differences between the measures are normally distributed. You can use `hist(posttest-pretest)` or `qqnorm(posttest-pretest)` to examine that, as shown in the chapters on graphics. If they are not, consider the Wilcoxon signed-rank test. For details see Sect. 17.10.

Other than the obvious pairing, observations are independent. For example, if siblings were also in your data set, it would be important to take that into account in a more complex model.

You can perform the paired t-tests in R using the `t.test` function. This example assumes the data frame is attached.

```
> t.test(posttest, pretest, paired = TRUE)

        Paired t-test

data:  posttest and pretest
t = 14.4597, df = 99, p-value < 2.2e-16

alternative hypothesis:
  true difference in means is not equal to 0

95 percent confidence interval:
 6.11708 8.06292

sample estimates:
mean of the differences
                   7.09
```

The `t.test` function call above has three main arguments.

1. The first variable to compare.
2. The second test variable to compare. The function will subtract this from the first to get the mean difference. If these two were reversed, the p-value would be the same; however, the mean difference would be -7.09 (i.e., negative) and the confidence interval would be around that.

3. The `paired = TRUE` argument tells it that the two variables are correlated rather than independent. It is extremely important that you set this option for a paired test! If you forget this, it will perform an independent-groups t-test instead.

The results show that the mean difference of 7.09 is a statistically significant *p*-value of 2.2e-16, or 0.00000000000000022. So we would reject the hypothesis that the means are the same.

17.9 Wilcoxon–Mann–Whitney Rank Sum: Independent Groups

The Wilcoxon rank sum test, also known as the Mann–Whitney U test, compares two groups on the mean rank of a dependent variable that is at least ordinal (e.g., low, medium, high).

In SAS, you could run this test with PROC NPAR1WAY. SPSS uses the similar NPAR TESTS procedure, perhaps in conjunction with the SPSS Exact Tests module.

The Wilcoxon–Mann–Whitney test's assumptions are as follows:

The distributions in the two groups must be the same, other than a shift in location; that is, the distributions should have the same variance and skewness. Examining a histogram is a good idea if you have at least 30 subjects. A box plot is also helpful regardless of the number of subjects in each group.

Observations are independent. For example, if you measured the same subjects repeatedly, it would be important to take that into account in a more complex model.

The `wilcox.test` function works very much like the `t.test` function:

```
> wilcox.test(q1 ~ gender, data = mydata100)

        Wilcoxon rank sum test with continuity correction

data:  q1 by gender

W = 1841.5, p-value = 6.666e-06

alternative hypothesis: true location shift is not equal to 0
```

The `wilcox.test` function call above has two main arguments.

1. The formula `q4~gender` is in the form dependent~independent. For details, see Sect. 5.7.3, "Controlling Functions with Formulas."

2. Instead of a formula, you can specify two variables to compare, such as `wilcox.test(x,y)`, and they can be selected using any of R's many variable selection approaches. See the examples in the Sect. 17.6.
3. The `data` argument specifies the data frame to use, if (and only if) you are using a formula.

The *p*-value of 6.666e-06 or 0.000006666, is less than 0.05, so we would reject the hypothesis that the males and females have the same distribution on the q4 variable. The median is the more popular measure of location to report on for a nonparametric test, and we can apply that function using the `aggregate` function. Recall that `aggregate` requires its group argument to be a list or data frame:

```
> aggregate( q1, data.frame(gender),
+   median, na.rm = TRUE)

  gender x

1 Female 4
2   Male 3
```

We see that the median of the females is 4 and that of the males is 3.

17.10 Wilcoxon Signed-Rank Test: Paired Groups

The goal of a Wilcoxon signed-rank test is to compare the mean rank of two correlated measures that are at least ordinal (e.g., low, medium, high) in scale. These are often the same measure taken on the same subjects at two different times.

In SAS you can perform this test by creating a difference score and then running that through PROC UNIVARIATE. In SPSS, you would use the NPTESTS procedure.

The assumptions of the Wilcoxon signed-rank test are as follows:

The scale of measurement is at least ordinal.
The difference between the two measures has a symmetric distribution. If not, you can use the sign test in Sect. 17.11.
Other than the obvious pairing, the observations are independent. For example, if your data also contained siblings, you would want a more complex model to make the most of that information.

This test works very much like the `t.test` function with the `paired` argument:

```
> wilcox.test( posttest, pretest, paired = TRUE)
```

```
        Wilcoxon signed rank test with continuity correction

data:  posttest and pretest

V = 5005, p-value < 2.2e-16

alternative hypothesis: true location shift is not equal to 0
```

The `wilcox.test` function call above has three main arguments.

1. The first variable to compare.
2. The second variable to compare. The function will subtract this from the first and then convert the difference to ranks.
3. The `paired = TRUE` argument, which tells it that the variables are correlated. Be careful, as without this argument, it will perform the Wilcoxon rank-sum test for independent groups. That is a completely different test and it would be inappropriate for correlated data.

The *p*-value of 2.2e-16 is less than 0.05, so we would reject the hypothesis that the location difference is zero. As Dalgaard [16] pointed out, with a sample size of 6 or fewer, it is impossible to achieve a significant result with this test.

The median is the more popular measure of location to report for a nonparametric test, and we can calculate them with the `median` function.

```
> median(pretest)

 [1] 75

> median(posttest)

 [1] 82
```

17.11 Sign Test: Paired Groups

The goal of the sign test is to determine if one set of paired variables is consistently larger or smaller than the other.

In SAS you can perform this test by creating a difference score and then running that through PROC UNIVARIATE. In SPSS, you would use the NPTESTS procedure.

The assumptions of the sign test are as follows:

The measurement scale of each pair is at least ordinal. That is, you can tell which of each pair is greater.

Other than the obvious pairing, the observations are independent. For example, if your data also contained siblings, you would want a more complex model to make the most of that information.

While this test does away with the Wilcoxon rank-sum test's assumption that the distribution of the differences is symmetric, it does so at a substantial loss of power. Choose this test with care!

Base R does not include a built-in version of the sign test. Instead, it provides the `pbinom` function to let you do one based on the binomial distribution. However, Arnholt's `PASWR` package [3] does include a `SIGN.test` function. That package is designed to accompany Probability and Statistics with R, by Ugarte, et al. [62]. To use `PASWR`, you must first install it (Chap. 2, "Installing and Updating R"). Then you must load it with either the *Packages> Load Packages* menu item or the function call

```
library("PASWR")
```

Here is how the `SIGN.test` function works:

```
> SIGN.test(posttest, pretest, conf.level=.95)

        Dependent-samples Sign-Test

data:  posttest and pretest
S = 96, p-value < 2.2e-16
alternative hypothesis: true median difference is not equal to 0
95 percent confidence interval:
 5 8
sample estimates:
median of x-y
            6

                  Conf.Level L.E.pt U.E.pt
Lower Achieved CI     0.9431      5      8
Interpolated CI       0.9500      5      8
Upper Achieved CI     0.9648      5      8
> sum(posttest > pretest)
[1] 96
```

The `SIGN.test` function call above has three main arguments.

1. The first variable to compare.
2. The second variable to compare. The function will subtract this from the first and then work with the sign and the median of the differences.
3. Optionally, the `conf.int` argument specifies the confidence interval it could calculate on the median x-y difference.

The `S = 96` is the number of positive signs. That is, for 96 pairs of values the posttest score was higher than the pretest score. The p-value of less than 2.2e-16 is much smaller than 0.05, so you would conclude the result is significant. You can report either the number of positive signs or the median difference of 6. The 95% confidence interval places the population median between 5 and 8.

17.12 Analysis of Variance

An analysis of variance (ANOVA or AOV) tests for group differences on the mean of a continuous variable divided up by one or more categorical factors. Both SAS and SPSS have ANOVA and GLM procedures that perform this analysis.

Its assumptions are as follows:

The measure is interval-level continuous data. If the data are only ordinal (e.g., low-medium-high), consider the Kruskal–Wallis test for a single factor (i.e., one-way ANOVA).
The measure is normally distributed. You can examine that assumption with `hist(myVar)` or `qqnorm(myvar)`, as shown in the chapters on graphics. If they are not, consider the Kruskal–Wallis test.
The observations are independent. For example, if each group contains subjects that were measured repeatedly over time or who are correlated (e.g., same family), you would want a more complex model to make the most of that information.
The variance of the measure is the same in each group.

We can get the group means using the `aggregate` function:

```
> aggregate( posttest,
+   data.frame(workshop),
+   mean, na.rm = TRUE)

  workshop      x
1        R 86.258
2      SAS 79.625
3     SPSS 81.720
4    Stata 78.947
```

Similarly, we can get variances by applying the `var` function:

```
> aggregate( posttest,
+   data.frame(workshop),
+   var, na.rm = TRUE)

  workshop      x
1        R 24.998
2      SAS 37.549
3     SPSS 19.543
4    Stata 73.608
```

You can see that the variance for the Stata group is quite a bit higher than the rest. Levene's test will provide a test of significance for that:

```
> levene.test(posttest, workshop)

Levene s Test for Homogeneity of Variance

      Df F value  Pr(>F)
group  3    2.51 0.06337 .
      95

---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1    1
```

The Levene test's null hypothesis is that the variances do not differ. Since it calculated a *p*-value of 0.06337, we can conclude that the differences in variance are not significant.

The `aov` function calculates the *a*nalysis *o*f *v*ariance.

```
> myModel <- aov( posttest ~ workshop, data = mydata100 )
```

The `aov` function call above has two arguments.

1. The formula `posttest~workshop` is in the form dependent~independent. The independent variable must be a factor. See Sect. 5.7.3, "Controlling Functions with Formulas," for details about models with more factors, interactions, nesting, and so forth.
2. The `data` argument specifies the data frame to use for the formula. If you do not supply this argument, you can use any other valid form of variable specification that tells it in which data frame your variables are stored (e.g., `mydata$postest~mydata$workshop`).

We can see some results by printing myModel:

```
> myModel

Call:
   aov(formula = posttest ~ workshop, data = mydata100)

Terms:
                workshop Residuals
Sum of Squares   875.442  3407.548
Deg. of Freedom        3        95
Residual standard error: 5.989067

Estimated effects may be unbalanced
1 observation deleted due to missingness
```

Of course, the summary function has methods for ANOVA models; it prints the same result.

```
> anova(myModel)

Analysis of Variance Table

Response: posttest
          Df Sum Sq Mean Sq F value    Pr(>F)
workshop   3  875.4   291.8  8.1356 7.062e-05 ***
Residuals 95 3407.5    35.9

---
Signif. codes:  0  ***  0.001  **  0.01  *  0.05  .  0.1    1
```

Given the p-value of 7.062e-05 or 0.00007062, we would reject the hypothesis that the means are all the same. However, which ones differ? The pairwise.t.test function provides all possible tests and corrects them for multiple testing using the Holm method by default. In our case, we are doing six t-tests, so the best (smallest) p-value is multiplied by 6, the next best by 5, and so on. This is also called a sequential Bonferroni correction.

```
> pairwise.t.test(posttest, workshop)
        Pairwise comparisons using t tests with pooled SD

data:  posttest and workshop

      R       SAS     SPSS
SAS   0.00048 -       -
SPSS  0.02346 0.44791 -
Stata 0.00038 0.71335 0.39468

P value adjustment method: holm
```

We see that the posttest scores are significantly different for R compared to the other three. The mean scores for the SAS, SPSS, and Stata workshops do not differ significantly among themselves.

An alternative comparison approach is to use Tukey's honestly significant difference (HSD) test. The TukeyHSD function call below uses only two arguments: the model and the factor whose means you would like to compare:

```
> TukeyHSD(myModel, "workshop")

  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = posttest ~ workshop, data = mydata100)
```

```
$workshop

                diff      lwr      upr   p adj
SAS-R       -6.63306 -10.8914 -2.37472 0.00055
SPSS-R      -4.53806  -8.7481 -0.32799 0.02943
Stata-R     -7.31070 -11.8739 -2.74745 0.00036
SPSS-SAS     2.09500  -2.3808  6.57078 0.61321
Stata-SAS   -0.67763  -5.4871  4.13185 0.98281
Stata-SPSS -2.77263  -7.5394  1.99416 0.42904
```

The “diff” column provides mean differences. The “lwr” and “upr” columns provide lower and upper 95% confidence bounds, respectively. Finally, the “p adj” column provides *p*-values adjusted for the number of comparisons made. The conclusion is the same: the R group’s posttest score differs significantly from the others’ posttest scores, but the others’ scores do not differ significantly among themselves.

We can graph these results using the `plot` function (Fig. 17.2).

```
> plot( TukeyHSD(myModel, "workshop") )
```

The `plot` function also provides appropriate diagnostic plots (not shown). These are the same plots shown and discussed in Sect. 17.5, “Linear Regression.”

```
> plot(myModel)
```

You can perform much more complex ANOVA models in R, but they are beyond the scope of this book. A good book on ANOVA is Pinheiro and Bates’ *Mixed Effects Models in S and S-Plus* [45].

17.13 Sums of Squares

In ANOVA, SAS and SPSS provide partial (type III) sums of squares and F-tests by default. SAS also provides sequential sums of squares and F-tests by default. SPSS will provide those if you ask for them. R provides sequential ones in its built-in functions. For one-way ANOVAs or for two-way or higher ANOVAS with equal cell sizes, there is no difference between sequential and partial tests. However, in two-way or higher models that have unequal cell counts (unbalanced models), these two sums of squares lead to different F-tests and *p*-values.

The R community has a very strongly held belief that tests based on partial sums of squares can be misleading. One problem with them is that they test the main effect after supposedly partialing out significant interactions. In many circumstances, that does not make much sense. See the [64] for details.

If you are sure you want type III sums of squares, Fox's car package will calculate them using its Anova function. Notice that the function begins with a capital letter "A"! To use the car package, you must first install it (Chap. 2, "Installing and Updating R"). Then you must load it with either the *Packages> Load Packages* menu item or the function call

```
> library("car")

Attaching package:  car
        The following object(s) are masked from
        package:Hmisc : recode
```

You can see from the above message that the car package also contains a recode function that is now blocking access to (masking) the recode function in the Hmisc package. We could have avoided that message by detaching Hmisc first, but this is a good example of function masking and it does not affect the function we need.

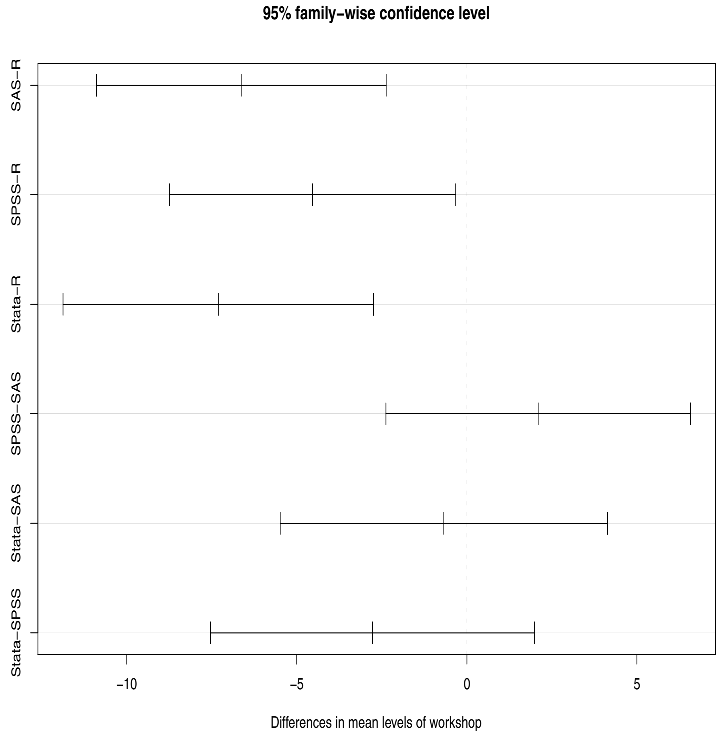


Fig. 17.2. A plot of the Tukey HSD test results showing the R group differing from the other three workshops

We use the Anova function with the type="III" argument to request the sums of squares we need. Since our model is a one-way ANOVA, the results are identical to those from the lowercase anova function that is built in to R, except that the *p*-values are expressed in scientific notation and the intercept, the grand mean in this case, is tested:

```
> Anova(myModel, type="III")

Anova Table (Type III tests)

Response: posttest

            Sum Sq Df   F value    Pr(>F)
 (Intercept) 230654  1 6430.4706 < 2.2e-16 ***
workshop       875  3    8.1356 7.062e-05 ***
Residuals     3408 95

---
Signif. codes:  0  ***  0.001  **  0.01  *  0.05  .  0.1    1
```

To get reasonable type III tests when doing more complex ANOVA models, it is important to set the contrasts option. The function call

```
options( contrasts=c("contr.sum", "contr.poly") )
```

is an example of this.

17.14 The Kruskal–Wallis Test

The nonparametric equivalent to a one-way ANOVA is the Kruskal–Wallis test. The Kruskal–Wallis test compares groups on the mean rank of a variable that is at least ordinal (e.g., low, medium, high). SAS offers it in PROC NPAR1WAY and SPSS has it in NPAR TESTS.

Its assumptions are as follows:

The distributions in the groups must be the same, other than a shift in location. This is often misinterpreted to mean that the distributions do not matter at all. That is not the case. They do not need to be normally distributed, but they do need to generally look alike. Otherwise, the test can produce a significant result if, for example, the distributions are skewed in opposite directions but are centered in roughly the same place.

The distributions should have roughly the same variance. Since the test requires no particular distribution, there is no single test for this assumption. Box plots are a good way to examine this assumption.

The observations are independent. For example, if each group contains subjects that were measured repeatedly over time or who are correlated (e.g., same family), you would want a more complex model to make the most of that information.

What follows is an example that uses the `kruskal.test` function to compare the different workshops on the means of the q4 variable:

```
> kruskal.test(posttest~workshop)

        Kruskal-Wallis rank sum test

data:  posttest by workshop

Kruskal-Wallis chi-squared=21.4448, df=3, p-value=8.51e-05
```

The `kruskal.test` function call above has two arguments.

1. The formula `posttest~workshop` is in the form dependent~independent. For details, see Sect. 5.7.3.
2. The `data` argument specifies the data frame to use for the formula. If you do not supply this argument, you can use any other valid form of variable specification that tells it in which data frame your variables are stored (e.g., `mydata$postest~mydata$workshop`).

The *p*-value of 8.51e-05, or 0.0000851, is smaller than the typical cutoff of 0.05, so you would reject the hypothesis that the groups do not differ. The next question would be, which of the groups differ? The `pairwise.wilcox.test` function answers that question. The only arguments we use below are the measure and the factor, respectively:

```
> pairwise.wilcox.test(posttest, workshop)

        Pairwise comparisons using Wilcoxon rank sum test

data:  posttest and workshop

      R      SAS    SPSS
SAS   0.0012 -      -
SPSS  0.0061 0.4801 -
Stata 0.0023 0.5079 0.4033

P value adjustment method: holm

Warning messages:

1: In wilcox.test.default(xi, xj, ...) :
  cannot compute exact p-value with ties
```

So we see that the R workshop group differs significantly from the other three and that the other three do not differ among themselves. The median is the more popular measure to report with a nonparametric test. We can use the aggregate function to apply the median function:

```
> aggregate( posttest,
+   data.frame(workshop),
+   median, na.rm = TRUE)

  workshop    x
1        R 86.0
2      SAS 78.5
3     SPSS 81.0
4    Stata 78.0
```

17.15 Example Programs for Statistical Tests

The R examples require installing the car, Gmodels and Hmisc packages. See details in Chap. 2, "Installing and Updating R."

17.15.1 SAS Program for Statistical Tests

```
* Filename: Statistics.sas ;

LIBNAME myLib  C:\myRfolder ;
DATA  temp;
  SET myLib.mydata100;
  * pretend q2 and q1 are the same score
    measured at two times & subtract;
  myDiff=q2-q1; run;

* Basic stats in compact form;
PROC MEANS; VAR q1--posttest; RUN;

* Basic stats of every sort;
PROC UNIVARIATE; VAR q1--posttest; RUN;

* Frequencies & percents;
PROC FREQ; TABLES workshop--q4; RUN;

* Chi-square;
PROC FREQ;
  TABLES workshop*gender/CHISQ; RUN;
```

```
* ---Measures of association---;

* Pearson correlations;
PROC CORR; VAR q1-q4; RUN;

* Spearman correlations;
PROC CORR SPEARMAN; VAR q1-q4; RUN;

* Linear regression;
PROC REG;
  MODEL q4=q1-q3;
  RUN;

* ---Group comparisons---;

* Independent samples t-test;
PROC TTEST;
  CLASS gender;
  VAR q1; RUN;

* Nonparametric version of above
  using Wilcoxon/Mann-Whitney test;
PROC NPAR1WAY;
  CLASS gender;
  VAR q1; RUN;

* Paired samples t-test;
PROC TTEST;
  PAIRED pretest*posttest; RUN;

* Nonparametric version of above using
  both Signed Rank test and Sign test;
PROC UNIVARIATE;
  VAR myDiff;
  RUN;

*Oneway Analysis of Variance (ANOVA);
PROC GLM;
  CLASS workshop;
  MODEL posttest=workshop;
  MEANS workshop / TUKEY; RUN;

*Nonparametric version of above using
  Kruskal-Wallis test;
PROC npar1way;
```

```
  CLASS workshop;
  VAR posttest; RUN;
```

17.15.2 SPSS Program for Statistical Tests

```
* SPSS Program for Basic Statistical Tests.
* Filename: Statistics.sps.

CD C:\myRfolder .
GET FILE= mydata100.sav .
DATASET NAME DataSet2 WINDOW=FRONT.

* Descriptive stats in compact form.
DESCRIPTIVES VARIABLES=q1 to posttest
  /STATISTICS=MEAN STDDEV VARIANCE
   MIN MAX SEMEAN .

* Descriptive stats of every sort.
EXAMINE VARIABLES=q1 to posttest
  /PLOT BOXPLOT STEMLEAF NPPLOT
  /COMPARE GROUP
  /STATISTICS DESCRIPTIVES EXTREME
  /MISSING PAIRWISE.

* Frequencies and percents.
FREQUENCIES VARIABLES=workshop TO q4.

* Chi-squared.
CROSSTABS
  /TABLES=workshop  BY gender
  /FORMAT= AVALUE TABLES
  /STATISTIC=CHISQ
  /CELLS= COUNT ROW
  /COUNT ROUND CELL .

* ---Measures of association---.

* Person correlations.
CORRELATIONS
  /VARIABLES=q1 TO q4.

* Spearman correlations.
NONPAR CORR
  /VARIABLES=q1 to q4
  /PRINT=SPEARMAN.
```

```
* Linear regression.
REGRESSION
  /MISSING LISTWISE
  /STATISTICS COEFF OUTS R ANOVA
  /CRITERIA=PIN(.05) POUT(.10)
  /NOORIGIN
  /DEPENDENT q4
  /METHOD=ENTER q1 q2 q3.

REGRESSION
  /DEPENDENT q4
  /METHOD=ENTER q1 q2 q3.

* ---Group comparisons---.

* Independent samples t-test.
T-TEST
  GROUPS = gender( m  f )
  /VARIABLES = q1.

* Nonparametric version of above using
* Wilcoxon/Mann-Whitney test.
NPTESTS
  /INDEPENDENT TEST (posttest)
  GROUP (gender) MANN_WHITNEY.

* Paired samples t-test.
T-TEST
  PAIRS = pretest  WITH posttest (PAIRED).

* Nonparametric version of above using
* Wilcoxon Signed-Rank test and Sign test.
NPTESTS
  /RELATED TEST(pretest posttest) SIGN WILCOXON.

* Oneway analysis of variance (ANOVA).
UNIANOVA posttest  BY workshop
  /POSTHOC = workshop ( TUKEY )
  /PRINT = ETASQ HOMOGENEITY
  /DESIGN = workshop .

* Nonparametric version of above using
  Kruskal Wallis test.
NPAR TESTS
```

```
  /K-W=posttest BY workshop(1 3).
```

17.15.3 R Program for Statistical Tests

```
# Filename: Statistics.R

setwd("c:/myRfolder")
load("mydata100.Rdata")
attach(mydata100)
options(linesize = 63)

head(mydata100)

# ---Frequencies & Univariate Statistics---

# Deducer s frequencies() function

# The easy way using the Hmisc package.
library("Hmisc")
describe(mydata100)

# R s built-in function.
summary(mydata100)

# The flexible way using built-in functions.

table(workshop)
table(gender)

# Proportions of valid values.
prop.table( table(workshop) )
prop.table( table(gender) )

# Rounding off proportions.
round( prop.table( table(gender) ), 2 )

# Converting proportions to percents.
round( 100* ( prop.table( table(gender) ) ) )

# Frequencies & Univariate
summary(mydata100)

# Means & Std Deviations
options(width=63)
sapply( mydata100[3:8], mean,  na.rm = TRUE)
sapply( mydata100[3:8], sd,    na.rm = TRUE)

# ---Crosstabulations---

# The easy way, using the gmodels package.
library("gmodels")
CrossTable(workshop, gender,
  chisq = TRUE, format = "SAS")

# The flexible way using built-in functions.

# Counts
myWG <- table(workshop, gender)
myWG     # Crosstabulation format.
myWGdata <- as.data.frame(myWG)
myWGdata # Summary or Aggregation format.

chisq.test(myWG)

# Row proportions.
prop.table(myWG, 1)

# Column proportions.
prop.table(myWG, 2)

# Total proportions.
prop.table(myWG)

# Rounding off proportions.
round( prop.table(myWG, 1), 2 )

# Row percents.
round( 100* ( prop.table(myWG, 1) ) )

# Adding Row and Column Totals.
addmargins(myWG, 1)
addmargins(myWG, 2)
addmargins(myWG)

# ---Correlation & Linear Regression---

# The rcorr.adjust function from the R Commander package
library("Rcmdr")
load("mydata.RData")
rcorr.adjust( mydata[3:6] )
```

```
# Spearman correlations.
rcorr.adjust( mydata[3:6], type = "spearman" )

# The built-in cor function.
cor( mydata[3:6],
  method = "pearson", use = "pairwise")

# The built-in cor.test function
cor.test(mydata$q1, mydata$q2, use = "pairwise")

# Linear regression.
lm( q4 ~ q1 + q2 + q3, data = mydata100)

myModel <- lm( q4 ~ q1 + q2 + q3, data = mydata100 )
myModel
summary(myModel)
anova(myModel) #Same as summary result.

# Set graphics parameters for 4 plots (optional).
par( mfrow = c(2, 2), mar = c(5, 4, 2, 1) + 0.1 )

plot(myModel)

# Set graphics parameters back to default settings.
par( mfrow = c(1, 1), mar = c(5, 4, 4, 2) + 0.1 )

# Repeat the diagnostic plots and route them
# to a file.
postscript("LinearRegDiagnostics.eps")
par( mfrow = c(2,2), mar = c(5, 4, 2, 1) + 0.1 )
plot(myModel)
dev.off()
par( mfrow = c(1, 1), mar = c(5, 4, 4, 2) + 0.1 )

myNoMissing <- na.omit(mydata100[ ,c("q1","q2","q3","q4")] )
myFullModel    <- lm( q4 ~ q1 + q2 + q3, data = myNoMissing)
myReducedModel <- lm( q4 ~ q1,           data = myNoMissing)
anova( myReducedModel, myFullModel)


# ---Group Comparisons---

# Independent samples t-test.
t.test( q1 ~ gender, data = mydata100)
```

```
# Same test; requires attached data.
t.test( q1[gender == "Male"],
        q1[gender == "Female"] )

# Same test using with() function.
with(mydata100,
  t.test( q4[ which(gender == "m") ],
          q4[ which(gender == "f") ] )
)

# Same test using subset() function.
t.test(
  subset(mydata100, gender == "m", select = q4),
  subset(mydata100, gender == "f", select = q4)
)

# Paired samples t-test.
t.test(posttest, pretest, paired = TRUE)

# Equality of variance.
library("car")
levene.test(posttest, gender)

var.test(posttest~gender)

# Wilcoxon/Mann-Whitney test.
wilcox.test( q1 ~ gender, data = mydata100)
# Same test specified differently.
wilcox.test( q1[gender ==  Male ],
             q1[gender ==  Female ] )
aggregate( q1, data.frame(gender),
  median, na.rm = TRUE)

# Wilcoxon signed rank test.
wilcox.test( posttest, pretest, paired = TRUE)
median(pretest)
median(posttest)

# Sign test.
library("PASWR")
SIGN.test(posttest, pretest, conf.level = .95)

# Analysis of Variance (ANOVA).
aggregate( posttest,
  data.frame(workshop),
  mean, na.rm = TRUE)

aggregate( posttest,
  data.frame(workshop),
  var, na.rm = TRUE)

library("car")
levene.test(posttest, workshop)

myModel <- aov(posttest ~ workshop,
  data = mydata100)
myModel

anova(myModel)
summary(myModel)  # same as anova result.

# type III sums of squares
library("car")
Anova(myModel, type = "III")

pairwise.t.test(posttest, workshop)

TukeyHSD(myModel, "workshop")
plot( TukeyHSD(myModel, "workshop") )

# Repeat TukeyHSD plot and route to a file.
postscript("TukeyHSD.eps")
plot( TukeyHSD(myModel, "workshop") )
dev.off()

# Set graphics parameters for 4 plots (optional).
par( mfrow = c(2, 2), mar = c(5, 4, 2, 1) + 0.1 )
plot(myModel)
# Set graphics parameters back to default settings.
par( mfrow = c(1, 1), mar = c(5, 4, 4, 2) + 0.1 )

# Nonparametric oneway ANOVA using
# the Kruskal-Wallis test.
kruskal.test(posttest ~ workshop)

pairwise.wilcox.test(posttest, workshop)

aggregate( posttest,
  data.frame(workshop),
  median, na.rm = TRUE)
```

18

Conclusion

As we have seen, R differs from SAS and SPSS in many ways. R has a host of features that the other programs lack such as functions whose internal workings you can see and change, fully integrated macro and matrix capabilities, the most extensive selection of analytic methods available, and a level of flexibility that extends all the way to the core of the system. A detailed comparison of R with SAS and SPSS is contained in Appendix B.

Perhaps the most radical way in which R differs from SAS and SPSS is R's object orientation. Although objects in R can be anything: data, functions, even operators, it may be helpful to focus on objects as data; R is data oriented. R has a rich set of data structures that can handle data in standard rectangular data sets as well as vectors, matrices, arrays and lists. It offers the ability for you to define your own type of data structures and then adapt existing functions to them by adding new analytic methods. These data structures contain attributes – more data, about the data – that help R's functions (procedures, commands) to automatically choose the method of analysis or graph that best fits that data. R's output is also in the form of data that other functions can easily analyze. That ability is so seamless that functions can nest within other functions until you achieve the result you desire.

R is also controlled by data. Its arguments are often not static parameter values or keywords as they are in SAS or SPSS. Instead, they are vectors of data: character vectors of variable names, numeric values indicating column widths of text files to read, logical values of observations to select, and so on. You even generate patterns of variable names the same way you would patterns of any other data values. Being controlled by data is what allows macro substitution to be an integral part of the language. This data, or object, orientation is a key factor that draws legions of developers to R.

If you are a SAS or SPSS user who has happily avoided the complexities of output management, macros, and matrix languages, R's added functionality may seem daunting to learn at first. However, the way R integrates all these features may encourage you to expand your horizons. The added power of R and its free price make it well worth the effort.

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3_18, © Springer Science+Business Media, LLC 2011

We have discussed how R compares to SAS and SPSS and how you can do the very same things in each. However, what we have not covered literally fills many volumes. I hope this will start you on a long and successful journey with R.

I plan to improve this book as time goes on, so if there are changes you would like to see in the next edition, please feel free to contact me at muenchen.bob@gmail.com. Negative comments are often the most useful, so do not worry about being critical.

Have fun working with R!

Appendix A: Glossary of R Jargon

Below is a selection of common R terms defined first using SAS/SPSS terminology (or plain English when possible) and then more formally using R terminology. Some definitions using SAS/SPSS terminology are quite loose given the fact that they have no direct analog of some R terms. Definitions using R terminology are often quoted (with permission) or paraphrased from *S Poetry* by Burns [10].

Apply

The process of having a command work on variables or observations. Determines whether a procedure will act as a typical command or as a function instead. More formally, the process of targeting a function on rows or columns. Also the `apply` function is one of several functions that controls this process.

Argument

Option that controls what procedures or functions do. Confusing because in R, functions do what both procedures and functions do in SAS/SPSS. More formally, input(s) to a function that control it. Includes data to analyze.

Array

A matrix with more than two dimensions. All variables must be of only one type (e.g., all numeric or all character). More formally, a vector with a dim attribute. The dim controls the number and size of dimensions.

Assignment function

Assigns values like the equal sign in SAS/SPSS. The two-key sequence, "`<-`", that places data or results of procedures or transformations into a variable or data set. More formally, the two-key sequence, "`<-`", that gives names to objects.

Atomic object

A variable whose values are all of one type, such as all numeric or all character. More formally, an object whose components are all of one mode. Modes allowed are numeric, character, logical, or complex.

Attach

The process of adding a data set or add-on module to your path. Attaching a data set appears to copy the variables into an area that lets you use them by a simple name like "gender" rather than by compound name like "mydata$gender." Done using the `attach` function. More formally, the process of adding a database to your search list. Also a function that does this.

Attributes

Traits of a data set like its variable names and labels. More formally, traits of objects such as names, class, or dim. Attributes are often stored as character vectors. You can view attributes and set using the `attr` function.

Class

An attribute of a variable or data set that a procedure may use to change its options automatically. More formally, the class attribute of an object determines which method of a generic function is used when the object is an argument in the function call. The `class` function lets you view or set an object's class.

Component

Like a variable in a data set, or one data set in a zipped set of data sets. More formally, an item in a list. The length of a list is the number of components it has.

CRAN

The Comprehensive R Archive Network at `http://cran.r-project.org/`. Consists of a set of sites around the world called mirrors that provide R and its add-on packages for you to download and install.

Data frame

A data set. More formally, a set of vectors of equal length bound together in a list. They can be different modes or classes (e.g., numeric and character). Data frames also contain *attributes* such as variable and row names.

Database

One data set or a set of them, or an add-on module. More formally, an item on the search list or something that might be. Can be an R data file or a package.

Dim

A variable whose values are the number of rows and columns in a data set. It is stored in the data set itself. More formally, the attribute that describes the *dim*ensions of an array. The `dim` function retrieves or changes this attribute.

Element

A specific value for a variable. More formally, an item in a vector.

Extractor function

A procedure that does useful things with a saved model such as make predictions or plot diagnostics. More formally, a function that has methods that apply to modeling objects.

Factor

A categorical variable and its value labels. Value labels may be nothing more than "1," "2,". . . , if not assigned explicitly. More formally, the type of object that represents a categorical variable. It stores its labels in its levels attribute. The `factor` function creates regular factors and the `order` function creates ordinal factors.

Function

A procedure or a function. When you apply it down through cases, it is just like a SAS/SPSS procedure. However, you can also apply it across rows like a SAS/SPSS function. More formally, an R program that is stored as an object.

Generic function

A procedure or function that has different default options or arguments set depending on the kind of data you give it. More formally, a function whose behavior is determined by the class of one or more of its arguments. The class of the relevant argument(s) determines which method the generic function will use.

Index

The order number of a value in a variable or the order number of a variable in a data set. More formally, the order number of an element in a vector or the component in a list or data frame. In our practice data set gender is the second variable, so its index value is 2. Since index values are used as subscripts, these two terms are often used interchangeably. See also *subscript*.

Install

The process of installing an R add-on module. More formally, adding a package into your library. You install packages just once per version of R. However, to use it you must load it from the library every time you start R.

Label

A procedure that creates variable labels. Also, a parameter that sets value labels using the `factor` or `ordered` procedures. More formally, a function from the `Hmisc` package that creates variable labels. Also an argument that sets factor labels using the `factor` or `ordered` functions.

Length

The number of observations/cases in a variable, including missing values, or the number of variables in a data set. More formally, a measure of objects. For vectors, it is the number of its elements (including NAs). For lists or data frames, it is the number of its components.

Levels

The values that a categorical variable can have. Actually stored as a part of the variable itself in what appears to be a very short character variable (even when the values themselves are numbers). More formally, an attribute to a factor object that is a character vector of the values the factor can have. Also an argument to the `factor` and `ordered` functions that can set the levels.

Library

Where a given version of R stores its base packages and the add-on modules you have installed. Also a procedure that loads a package from the library into working memory. You must do that in every R session before using a package. More formally, a directory containing R packages that is set up so that the `library` function can attach it.

List

Like a zipped collection of data sets that you can analyze easily without unzipping. More formally, a set of objects of any class. Can contain vectors, data frames, matrices, and even other lists. The `list` function can create lists and many functions store their output in lists.

Load

Bringing a data set (or collection of data sets) from disk to memory. You must do this before you can use data in R. More formally, bringing an R data file into your workspace. The `load` function is one way to perform this task.

Matrix

A data set that must contain only one type of variable, e.g., all numeric or character. More formally, a two-dimensional array, that is, a vector with a dim attribute of length 2.

Method

The analyses or graphs that a procedure will perform by default, for a particular kind of variable. The default settings for some procedures depend on the scale of the variables you provide. For example, `summary(temperature)` provides mean temperature while with a factor such as `summary(gender)` it counts males and females. More formally, a function that provides the calculation of a generic function for a specific class of object. The `mode` function will display an object's mode.

Mode

A variable's type such as numeric or character. More formally, a fundamental property of an object. Can be numeric, character, logical, or complex. In other words, what *type* means to SAS and SPSS, *mode* means to R.

Modeling function

A procedure that performs estimation. More formally, a function that tests association or group differences and usually accepts a formula (e.g., `y~x`) and a `data =` argument.

Model objects

A model created by a modeling function.

NA

A missing value. Stands for *N*ot *A*vailable. See also NaN.

Names

Variable names. They are stored in a character variable that is a part of a data set or variable. Since R can use an index number instead, names are optional. More formally, an attribute of many objects that labels the elements or components of the object. The `names` function extracts or assigns variable names.

NaN

A missing value. Stands for *N*ot a *N*umber. Something that is undefined mathematically such as zero divided by zero.

NULL

An object you can use to drop variables or values. When you assign it to an object, R deletes the object. E.g., `mydata$x <- NULL` causes

R to drop the variable x from the data set mydata. NULL has a zero length and no particular mode.

Numeric

A variable that contains only numbers. More formally, the atomic mode that represents real numbers. This contains storage modes double, single, and integer.

Object

A data set, a variable, or even the equivalent of a SAS/SPSS procedure. More formally, almost everything in R. If it has a mode, it is an object. Includes data frames, vectors, matrices, arrays, lists, and functions.

Object-oriented programming

A style of software in which the output of a procedure depends on the type of data you provide it. R has an object orientation.

Option

A statement that sets general parameters, such as the width of each line of output. More formally, settings that control some aspect of your R session, such as the width of each line of output. Also a function that queries or changes the settings.

Package

An add-on module and related files such as help or practice data sets bundled together. May come with R or be written by its users. More formally, a collection of functions, help files, and, optionally, data objects.

R

A free and open source language and environment for statistical computing and graphics.

R-PLUS

A commercial version of R from XL Solutions, Inc.

Revolution R

A commercial version of R from Revolution Analytics, Inc.

Ragged array

The layers of an array are like stacked matrices. Each matrix layer must have the same number of rows and columns. However, it is far more common that subsets of a data set have different numbers of rows (e.g., different number of subjects per group). The data would then be too “ragged” to store in an array. Therefore, the data would instead be stored in a typical data set (an R data frame) and be viewed as a *ragged array.* A vector that stores group data, each with a different length is also viewed as a ragged array, see `help("tapply")`.

Revolution R Enterprise

A commercial version of R.

Replacement

A way to replace values. More formally, when you use subscripts on the left side of an assignment to change the values in an object, for example, setting 9 to missing: `x[x == 9]  <-  NA`

Row Names

A special type of ID variable. The row names are stored in a character variable that is a part of a data set. By default, row names are the character strings “1,” “2,” etc. More formally, an attribute of a data frame that labels its rows. The `row.names` function displays or changes row names.

S

The language from which R evolved. R can run many S programs, but S cannot use R packages.

S3, S4

Used in the R help files to refer to different versions of S. The differences between them are of importance mainly to advanced programmers.

Script

The equivalent of a SAS or SPSS program file. An R program.

Search list

Somewhat like an operating system search path for R objects. More formally, the collection of databases that R will search, in order, for objects.

S-PLUS

The commercial version of S. Mostly compatible with R but will not run R packages.

Subscript

Selecting (or replacing) values or variables using index numbers or variable names in square brackets. In our practice data, gender is the second variable so we can select it using `mydata[ ,2]` or `mydata[ ,"gender"]`. For two-dimensional objects, the first subscript selects rows, the second selects columns. If empty, it refers to all rows/columns.

Tags

Value names (not labels). Names of vector elements or list components, especially when used to supply a set of argument names (the tags) and their values. For examples see `help("list")`, `help("options")`, and `help("transform")`.

Vector

A variable. It can exist on its own in memory or it can be part of a data set. More formally, a set of values that have the same mode, i.e., an atomic object.

Workspace

Similar to your SAS work library, but in memory rather than on a temporary hard drive directory. Also similar to the area where your SPSS files reside before you save them. It is the area of your computer’s main memory where R does all its work. When you exit R, objects you created in your workspace will be deleted unless you save them to your hard drive first.

Appendix B: Comparison of Major Attributes of SAS and SPSS to those of R

Aggregating data

SAS & SPSS – One pass to aggregate, another to merge (if needed, SAS only), a third to use. SAS/SPSS functions, SUMMARY, or AGGREGATE offer only basic statistics.

R – A statement can mix both raw and aggregated values. Can aggregate on all statistics and even functions you write.

Controlling procedures or functions

SAS & SPSS – Statements such as CLASS and MODEL, options and formulas control the procedures.

R – You can control functions by manipulating the data's structure (its class), setting function options (arguments), entering formulas, and using apply and extraction functions.

Converting data structures

SAS & SPSS – In general, all procedures accept all variables; you rarely need to convert variable type.

R – Original data structure plus variable selection method determines structure. You commonly use conversion functions to get data into acceptable form.

Cost

SAS & SPSS – Each module has its price. Academic licenses cannot be used to help any outside organization, not even for pro bono work.

R – R and all its packages are free. You can work with any clients and even sell R if you like.

Data size

SAS & SPSS – Most procedures are limited only by hard disk size.

R – Most functions must fit the data into the computer's smaller random access memory.

Data structure

SAS & SPSS – Rectangular data sets only in main language.

R – Vector, factor, data frame, matrix, list, etc. You can even make up your own data structures.

Graphical user interfaces

SAS & SPSS – SAS Enterprise Guide, Enterprise Miner, and SPSS Modeler use the flowchart approach. SPSS offers a comprehensive menu and dialog box approach.

R – R has several. Red-R uses a flowchart similar to Enterprise Guide or Enterprise Miner. R Commander and Deducer look much like SPSS. Rattle uses a ribbon interface like Microsoft Office. While not yet as comprehensive as the SAS or SPSS GUIs, they are adequate for many analyses.

Graphics

SAS & SPSS – SAS/GRAPH's traditional graphics are powerful but cumbersome to use. Its newer SG graphics are easy to use and are high quality but are relatively inflexible. SPSS Graphics Production Language (GPL) is not as easy to program but is far more flexible than SAS/GRAPH.

R – Traditional graphics are extremely flexible but hard to use with groups. The `lattice` package does graphics in a similar way to SAS/GRAPH's new SG plots. They are easy to use and good quality, but not as flexible as those in the `ggplot2` package. The `ggplot2` package is more flexible than SAS/GRAPH and easy to use. Its functionality is very close to SPSS's GPL while having a simpler syntax.

Help and documentation

SAS & SPSS – Comprehensive, clearly written, and aimed at beginner to advanced users.

R – Can be quite cryptic; aimed at intermediate to advanced users.

Macro language

SAS & SPSS – A separate language is used mainly for repetitive tasks or adding new functionality. User-written macros run differently from built-in procedures.

R – R does not have a macro language as its language is flexible enough to not require one. User-written functions run the same way as built-in ones.

Managing data sets

SAS & SPSS – Relies on standard operating system commands to copy, delete, and so forth. Standard search tools can find data sets since they are in separate files.

R – Uses internal environments with its own commands to copy, delete, and so forth. Standard search tools cannot find multiple data frames if you store them in a single file.

Matrix language

SAS & SPSS – A separate language used only to add new features.

R – An integral part of R that you use even when selecting variables or observations.

Missing data

SAS & SPSS – When data is missing, procedures conveniently use all of the data they can. Some procedures offer listwise deletion as an alternative.

R – When data is missing, functions often provide no results (NA) by default; different functions require different missing value options.

Output management system

SAS & SPSS – Most users rarely use output management systems for routine analyses.

R – People routinely get additional results by passing output through additional functions.

Publishing results

SAS & SPSS – See it formatted immediately in any style you choose.

Quick cut and paste to word processor maintains fonts, table status, and style.

R – Process output with additional procedures that route formatted output to a file. You do not see it formatted as lined tables with proportional fonts until you import it to a word processor or text formatter.

Selecting observations

SAS & SPSS – Uses logical conditions in IF, SELECT IF, WHERE.

R – Uses wide variety of selection by index value, variable name, logical condition, string search (mostly the same as for selecting variables).

Selecting variables

SAS & SPSS – Uses simple lists of variable names in the form of: `x, y, z;  a to z;  a--z`. However, all of the variables for an analysis or graph must be in a single data set.

R – Uses wide variety of selection methods: by index value, variable name, logical condition, string search (mostly the same as for selecting observations). Analyses and graphs can freely combine variables from different data frames or other data structures.

Statistical methods

SAS & SPSS – SAS is slightly ahead of SPSS but both trail well behind R. SAS/IML Studio and SPSS can run R programs within SPSS programs.

R – Most new methods appear in R around 5 years before SAS and SPSS.

Tables

SAS & SPSS – Easy to build and nicely formatted, but limited in what they can display.

R – Can build table of the results of virtually all functions. However, formatting is extra work.

Variable labels

SAS & SPSS – Built in. Used by all procedures.

R – Added on. Used by few procedures and actually breaks some standard procedures.

Appendix C Automating your R setup

SAS has its autoexe.sas file that exists to let you automatically set options and run SAS code. R has a similar file called .Rprofile. This file is stored in your initial working directory, which you can locate with the `getwd()` function.

We will look at some useful things to automate in an .Rprofile.

Setting Options

In your .Rprofile, you can set options just as you would in R. I usually set my console width to 64 so the output fits training examples better. I also ask for 5 significant digits and tell it to mark significant results with stars. The latter is the default, but since many people prefer to turn that feature off, I included it. You would turn them off with a setting of FALSE.

```
options(width = 64, digits = 5, show.signif.stars = TRUE)
```

Enter `help("options")` for a comprehensive list of parameters you can set using the `options` function.

Setting the random number seed is a good idea if you want to generate numbers that are random but repeatable. That is handy for training examples in which you would like every student to see the same result. Here I set it to the number 1234.

```
set.seed(1234)
```

The `setwd` function sets the working directory, the place all your files will go if you don't specify a path.

```
setwd("c:/myRfolder")
```

Since I included the "/" in the working directory path, it will go to the root level of my hard drive. That works in most operating systems. Note that it must be a forward slash, even in Windows, which usually uses backward slashes in filenames. If you leave the slash off completely, it will set it to be a folder within your normal working directory.

Creating Objects

I also like to define the set of packages that I install whenever I upgrade to a new version of R. With these stored in myPackages, I can install them all with a single command. For details, see Chap. 2, "Installing and Updating R". This is the list of some of the packages used in this book.

```
myPackages <- c("car","hexbin","ggplot2",
  "gmodels","gplots", "Hmisc",
  "reshape2","Rcmdr","prettyR")
```

Loading Packages

You can have R load load your favorite packages automatically too. This is particularly helpful when setting up a computer to run R with a graphical user interface like R Commander. Loading packages at startup does have some disadvantages though. It slows down your startup time, takes up memory in your workspace, and can create conflicts when different packages have functions with the same name. Therefore, you do not want to load too many this way.

Loading packages at startup requires the use of the `local` function. The `getOption` function gets the names of the original packages to load and stores them in a character vector I named myOriginal. I then created a second character vector, myAutoLoads, containing the names of the packages I want to add to the list. I then combined them into one character vector, myBoth. Finally, I used the `options` function to change the default packages to the combined list of both the original list and my chosen packages:

```
local({
   myOriginal <- getOption("defaultPackages")

   # edit next line to be your list of favorites.
   myAutoLoads <- c("Hmisc","ggplot2")

   myBoth <- c(myOriginal,myAutoLoads)

   options(defaultPackages = myBoth)
 })
```

Running Functions

If you want R to run any functions automatically, you create your own single functions that do the required steps. To have R run a function before all others, name it "`.First`." To have it run the function after all others, name it "`.Last`." Notice that utility functions require a prefix of "`utils::`" or R will not find them while it is starting up. The `timestamp` function is one of those. It returns the time and date. The `cat` function prints messages. Its name comes from the UNIX command, cat. It is short for *catenate* (a synonym for concatenate). In essence, we will use it to concatenates the timestamp to your console output.

```
.First <- function()
  {
    cat("\n             Welcome to R!\n")
    utils::timestamp()
    cat("\n")
  }
```

You can also have R run any functions before exiting the package. I have it turn off my graphics device drivers with the `graphics.off` function to ensure that no files gets left open.

I like to have it save my command history in case I later decide I should have saved some of the commands to a script file. Below I print a farewell message and then save the history to a file named myLatest.Rhistory.

```
.Last <- function()
  {
    graphics.off() #turns off graphics devices just in case.
    cat("\n\n  myCumulative.Rhistory has been saved." )
    cat("\n\n  Goodbye!\n\n")
    utils::savehistory(file = "myCumulative.Rhistory")
  }
```

Warning: Since the `.First` and `.Last` functions begin with a period, they are invisible to the `ls` function by default. The command:

```
ls(all.names = TRUE)
```

will show them to you. Since they are functions, if you save a workspace that contains them, they will continue to operate whenever you load that workspace, even if you delete the .Rprofile! This can make it *very* difficult to debug a problem until you realize what is happening. As usual, you can display them by typing their names and run them by adding empty parentheses to them:

```
\verb+.First()+.
```

If you need to delete them from the workspace, `rm` will do it with no added arguments:

```
rm(.First,.Last)
```

Example .Rprofile

The following is the .Rprofile with all of the above commands combined. You do not have to type this in, it is included in the book's programs and data files at http://RforSASandSPSSusers.com

```
#     Startup Settings


# Place any R commands below.
```

```
options(width=64, digits=5, show.signif.stars = TRUE)
set.seed(1234)
setwd("c:/myRfolder")
myPackages <- c("car","hexbin","ggplot2",
  "gmodels","gplots", "Hmisc",
  "reshape","ggplot2","Rcmdr")
utils::loadhistory(file = "myCumulative.Rhistory")

# Load packages automatically below.

 local({
   myOriginal <- getOption("defaultPackages")

   # Edit next line to include your favorites.
   myAutoLoads <- c("Hmisc","ggplot2")
   myBoth <- c(myOriginal,myAutoLoads)
   options(defaultPackages = myBoth)
 })

# Things put here are done first.
.First <- function()
  {
    cat("\n             Welcome to R!\n")
    utils::timestamp()
    cat("\n")
  }


# Things put here are done last.
.Last <- function()
  {
    graphics.off()
    cat("\n\n  myCumulative.Rhistory has been saved." )
    cat("\n\n  Goodbye!\n\n")
    utils::savehistory(file = "myCumulative.Rhistory")
  }
```

References

[1] M. Almiron, E. S. Almeida, and M. Miranda. The reliability of statistical functions in four software packages freely used in numerical computation. *Brazilian Journal of Probability and Statistics*, Special Issue on Statistical Image and Signal Processing, (in press).

[2] C. Alzola and F. Harrell Jr. *An Introduction to S and the Hmisc and Design Libraries*. http://biostat.mc.vanderbilt.edu/twiki/pub/Main/RS/sintro.pdf, 2006.

[3] A. T. Arnholt. *PASWR: Probability and Statistics with R*. http://CRAN.R-project.org/package=PASWR. R package version 1.1.

[4] T. Baier. *SWordInstaller: SWord: Use R in Microsoft Word (Installer)*. http://CRAN.R-project.org/package=SWordInstaller, 2009. R package version 1.0-2.

[5] T. Baier and E. Neuwirth. Excel :: Com :: R. *Computational Statistics*, 22(1), 2007.

[6] D. Bates and M. Maechler. *Matrix: Sparse and Dense Matrix Classes and Methods*. http://CRAN.R-project.org/package=Matrix, 2011. R package version 0.999375-47.

[7] R. A. Becker, A. R. Wilks, R. Brownrigg, and T. P. Minka. *maps: Draw Geographical Maps*. http://CRAN.R-project.org/package=maps, 2010. R package version 2.1-5.

[8] R. S. Bivand, E. J. Pebesma, and V. Gómez-Rubio. *Applied spatial data analysis with R*. Springer, Berlin Heidelberg New York, 2008.

[9] B. Bolker. *Software Comparison*. http://finzi.psych.upenn.edu/R/Rhelp02a/archive/97802.html, 2007. R–Help Archive.

[10] P. Burns. *S poetry*. lib.stat.cmu.edu/S/Spoetry/Spoetry/Spoetry.pdf, 1998.

[11] D. Carr, N. Lewin-Koh, and M. Maechler. *hexbin: Hexagonal Binning Routines*. http://CRAN.R-project.org/package=hexbin, 2008. R package version 1.17.0.

[12] J. M. Chambers. *Software for Data Analysis: Programming with R*. Springer, Berlin Heidelberg New York, 2008.

R.A. Muenchen, *R for SAS and SPSS Users*, Statistics and Computing,
DOI 10.1007/978-1-4614-0685-3, © Springer Science+Business Media, LLC 2011

[13] W. S. Cleveland. *Visualizing Data*. Hobart, Summit, 1993.

[14] R core members, S. DebRoy, R. Bivand, et al. *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...*. http://CRAN.R-project.org/package=foreign, 2009.

[15] D. B. Dahl. *xtable: Export tables to LaTeX or HTML*. http://CRAN.R-project.org/package=xtable, 2009. R package version 1.5-5.

[16] P. Dalgaard. *Introductory Statistics with R (Statistics and Computing)*. Springer, Berlin Heidelberg New York, 2nd edition, 2008.

[17] M. Elff. *memisc: Tools for Management of Survey Data, Graphics, Programming, Statistics, and Simulation*. http://CRAN.R-project.org/package=memisc, 2010. R package version 0.95-31.

[18] I. Fellows. *Deducer*. http://CRAN.R-project.org/package=Deducer, 2010. R package version 0.4-2.

[19] R Foundation for Statistical Computing. *R: Regulatory Compliance and Validation Issues A Guidance Document for the Use of R in Regulated Clinical Trial Environments*. www.r-project.org/doc/R-FDA.pdf, 2008.

[20] J. Fox. *Rcmdr: R Commander*. http://socserv.socsci.mcmaster.ca/jfox/Misc/Rcmdr/, 2009. R package version 1.4-7.

[21] J. Fox, D. Bates, D. Firth, M. Friendly, G. Gorjanc, S. Graves, R. Heiberger, G. Monette, H. Nilsson, D. Ogle, B. Ripley, S. Weisberg, and A. Zeleis. *car: Companion to Applied Regression*. http://CRAN.R-project.org/package=car, 2009. R package version 1.2-12.

[22] M. Friendly. *Visualizing Categorical Data: Data, Stories, and Pictures*. SAS, Cary, 2000.

[23] R. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. H. Yang, and J. Zhang. Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5:R80, 2004.

[24] G. Grolemund and H. Wickham. *lubridate: Make dealing with dates a little easier*. http://CRAN.R-project.org/package=lubridate, 2010. R package version 0.2.3.

[25] G. Grolemund and H. Wickham. Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3), 2011.

[26] K. W. Haemer. Double scales are dangerous. *The American Statistician*, 2(3):24, 1948.

[27] R. M. Heiberger and E. Neuwirth. *R Through Excel: A Spreadsheet Interface for Statistics, Data Analysis, and Graphics (Use R)*. Springer, Berlin Heidelberg New York, 2009.

[28] M. Helbig and S. Urbanek. *JGR: JGR - Java Gui for R*. http://www.rosuda.org/JGR. R package version 1.6-3.

[29] T. Hengl. *A Practical Guide to Geostatistical Mapping*. http://spatial-analyst.net/book/, 2nd edition, 2010.

[30] J. M. Hilbe. *COUNT: Functions, Data, and Code for Count Data.* http://CRAN.R-project.org/package=COUNT, 2010. R package version 1.1.1.

[31] O. Jones, R. Maillardet, and A. Robinson. *Introduction to Scientific Programming and Simulation Using R.* Chapman & Hall/CRC, 2009.

[32] F. Harrell Jr. *Hmisc: Harrell Miscellaneous.* http://biostat.mc.vanderbilt.edu/s/Hmisc, 2008. R package version 3.5-2.

[33] K. B. Keeling and R. J. Pavur. A comparative study of the reliability of nine statistical software packages. *Computational Statistics & Data Analysis*, 51(8):3811 – 3831, 2007.

[34] R. Koenker. *quantreg: Quantile Regression.* http://CRAN.R-project.org/package=quantreg, 2010. R package version 4.53.

[35] M. Kuhn, S. Weston, N. Coulter, P. Lenon, and Z. Otles. *odfWeave: Sweave processing of Open Document Format (ODF) files.* http://CRAN.R-project.org/package=odfWeave, 2010. R package version 0.7.17.

[36] E. Lecoutre. The R2HTML package. *R News*, 3(3):33–36, 2003.

[37] F. Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat 2002, proceedings in Computational Statistics*, pages 575–580, 2002.

[38] J. Lemon and P. Grosjean. *prettyR: Pretty descriptive stats.* 2009. R package version 1.4.

[39] T. Lumley. *biglm: Bounded Memory Linear and Generalized Linear Models.* http://CRAN.Rproject.org/package=biglm, 2009. R package version 0.7.

[40] D. Meyer, A. Zeileis, and K. Hornik. *vcd: Visualizing Categorical Data.* http://CRAN.R-project.org/package=vcd, 2009. R package version 1.2-3.

[41] R. A. Muenchen and J. M. Hilbe. *R for Stata Users.* Springer, Berlin Heidelberg New York, 2010.

[42] P. Murrell. *R Graphics.* Chapman & Hall/CRC, Boca Raton, FL, 2005.

[43] E. Neuwirth. *RExcel.* http://rcom.univie.ac.at/download/RExcel.distro/RExcelInst.latest.exe, 2010. Version 3.1.12.

[44] E. J. Pebesma and R. S. Bivand. Classes and methods for spatial data in r. *R News*, 5 (2)(12), 2005.

[45] J. C. Pinheiro and D. M. Bates. *Mixed Effects Models in S and S-Plus.* Springer, Berlin Heidelberg New York, 2002.

[46] R Development Core Team. *R Data Import/Export.* Vienna, 2008.

[47] B. Ripley and from 1999 to Oct 2002 M. Lapsley. *RODBC: ODBC Database Access.* http://CRAN.R-project.org/package=RODBC, 2010. R package version 1.3-2.

[48] P. Roebuck. *matlab: MATLAB emulation package.* http://CRAN.R-project.org/package=matlab, 2008. R package version 0.8-2.

[49] D. Sarkar. *Lattice Multivariate Data Visualization with R.* Springer, Berlin Heidelberg New York, 2007.

[50] D. Sarkar. *lattice: Lattice Graphics*. http://CRAN.R-project.org/package=lattice, 2009. R package version 0.17-22.

[51] P. Spector. *Data Manipulation with R (Use R)*. Springer, Berlin Heidelberg New York, 2008.

[52] SPSS. *SPSS Statistics-R Integration Package*. Chicago, Illinois, 2008.

[53] H. P. Suter. *xlsReadWrite: Read and Write Excel Files (.xls)*. http://CRAN.R-project.org/package=xlsReadWrite, 2010. R package version 1.5.3.

[54] D. F. Swayne, D. Temple Lang, A. Buja, and D. Cook. GGobi: Evolving from XGobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 43:423–444, 2003.

[55] R Development Core Team. *R: A Language and Environment for Statistical Computing*. http://www.R-project.org, Vienna, 2008.

[56] R Development Core Team. *R Installation and Administration Manual*. http://cran.r-project.org/doc/manuals/R-admin.pdf, Vienna, 2008.

[57] Red-R Development Team. *Red-R Visual Programming for R*. http://www.red-r.org/, 2011. Version 1.8.

[58] Tinn-R Development Team. *Tinn-R: A editor for R language and environment statistical computing*. http://www.sciviews.org/Tinn-R/, 2004.

[59] D. Temple Lang. The omegahat environment: New possibilities for statistical computing. *Journal of Computational and Graphical Statistics*, 9(3), 2000.

[60] D. Temple Lang, D. Swayne, H. Wickham, and M. Lawrence. *rggobi: Interface between R and GGobi*. http://www.ggobi.org/rggobi, 2008. R package version 2.1.10.

[61] S. Theussl and A. Zeileis. Collaborative Software Development Using R-Forge. *The R Journal*, 1(1):9–14, 2009.

[62] M. Ugarte, A. Militino, and A. Arnholt. *Probability and Statistics with R*. CRC Press, 2008.

[63] S. Urbanek and M. Theus. High interaction graphics for R. *proceedings of the SSC 2003 Conference*, 2003.

[64] W. N. Venables. *Exegeses on Linear Models*. http://www.stats.ox.ac.uk/pub/MASS3/Exegeses.pdf, 1998.

[65] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, Berlin Heidelberg New York, 4th edition, 2002.

[66] W. N. Venables, D. M. Smith, and R Development Core Team. *An Introduction to R*. Springer, Berlin Heidelberg New York, 2007.

[67] G. R. Warnes, B. Bolker, L. Bonebakker, R. Gentleman, W. Huber, A. Liaw, T. Lumley, M. Maechler, A. Magnusson, S. Moeller, M. Schwartz, and B. Venables. *gplots: Various R Programming Tools for Plotting Data*. http://CRAN.R-project.org/package=gplots, 2009. R package version 2.7.1.

[68] G. R. Warnes, B. Bolker, T. Lumley, and R. C. Johnson. *gmodels: Various R Programming Tools for Model Fitting*. http://CRAN.R-project.org/package=gmodels, 2009. R package version 2.15.0.

[69] N. B. Weidmann, D. Kuse, and K. S. Gleditsch. *cshapes: CShapes Dataset and Utilities*. http://CRAN.R-project.org/package=cshapes, 2010. R package version 0.2-7.

[70] H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(12), 2007.

[71] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, Berlin Heidelberg New York, 2009.

[72] H. Wickham. *stringr: Make it easier to work with strings.* 2010. R package version 0.4.

[73] H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1), 2011.

[74] H. Wickham, D. Cook, H. Hofmann, and A. Buja. tourr: An r package for exploring multivariate data with projections. *Journal of Statistical Software*, 40(2):1–18, 2011.

[75] R. Wicklin. Rediscovering SAS/IML software: Modern data analysis for the practicing statistician. *Proceedings of the SAS Global Forum 2010 Conference*, 2010.

[76] L. Wilkinson. *The Grammar of Graphics*. Springer, Berlin Heidelberg New York, 2nd edition, 2005.

[77] G. Williams. Rattle: A data mining gui for r. *The R Journal*, 1(2):45–55, 2009.

[78] G. Williams. *Data Mining with Rattle and R*. Springer, Berlin Heidelberg New York, 2011.

Index