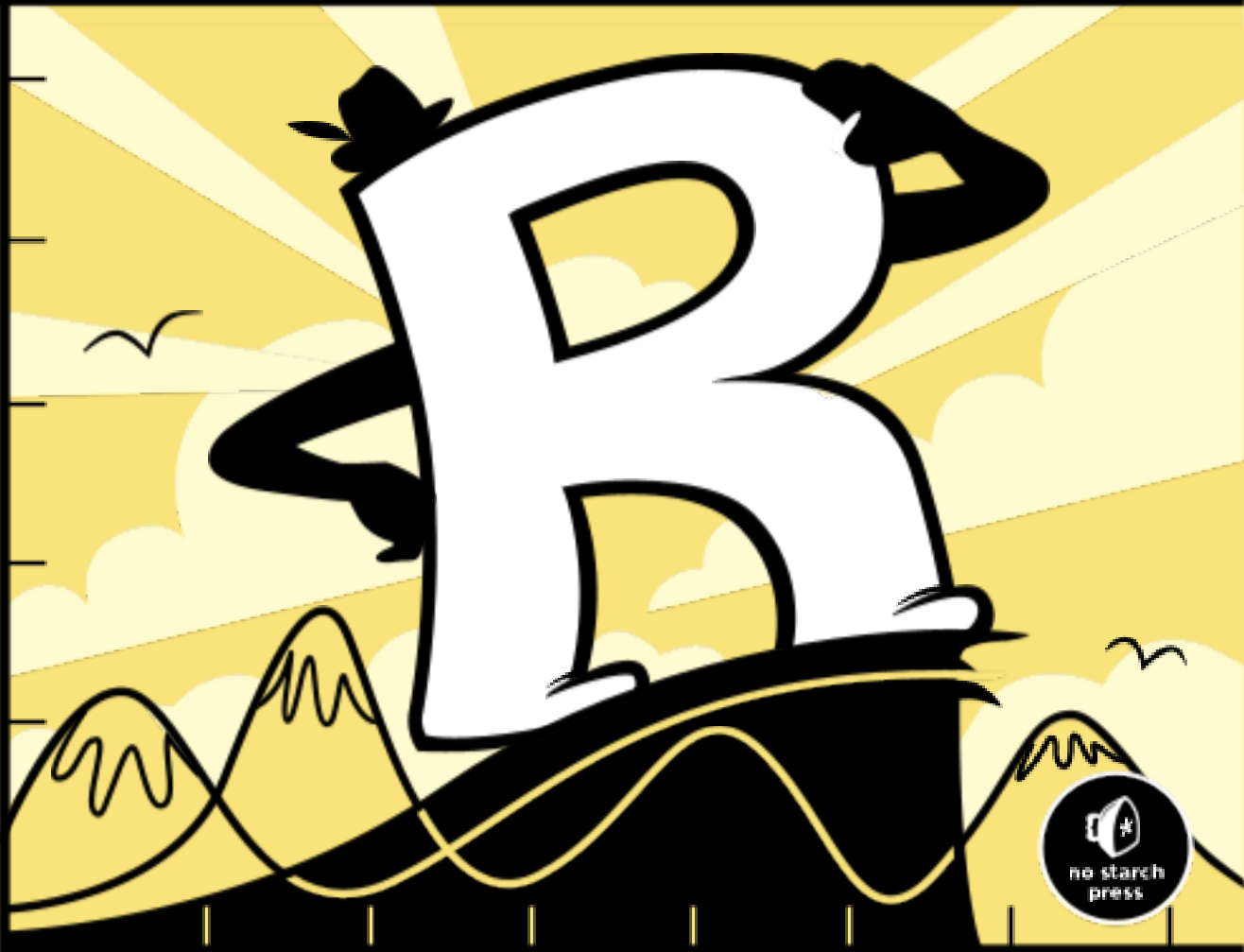


THE BOOK OF R

A FIRST COURSE
PROGRAMMING STATISTICS



The Book of R

A first Course in
Programming and Statistics

BRIEF CONTENTS

Preface.....	xxiii	
Acknowledgments.....		xxv
Introduction.....	xxvii	

PART I: THE LANGUAGE

Chapter1:GettingStarted.....	3	Chapter 2: Numerics, Arithmetic, Assignment, and Vectors	17
Chapter3:MatricesandArrays.....	39	Chapter4:Non- numericValues.....	59
Chapter5:ListsandDataFrames.....	89		
Chapter6:SpecialValues,Classes,andCoercion.....	103		
Chapter7:BasicPlotting.....	127		
Chapter8:ReadingandWritingFiles.....	147		

PART II: PROGRAMMING

Chapter9:CallingFunctions.....	165
Chapter10:ConditionsandLoops.....	179
Chapter11:WritingFunctions.....	215
Chapter12:Exceptions,Timings,andVisibility.....	241

PART III: STATISTICS AND PROBABILITY

Chapter13:ElementaryStatistics.....	261
Chapter14:BasicDataVisualization.....	289
Chapter15:Probability.....	309
Chapter16:CommonProbabilityDistributions.....	331

PART IV: STATISTICAL TESTING AND MODELING

Chapter17:SamplingDistributionsandConfidence.....	36
7 Chapter18:HypothesisTesting.....	385

Chapter19:AnalysisofVariance.....	435
Chapter20:SimpleLinearRegression.....	451
Chapter21:MultipleLinearRegression.....	485
Chapter22:LinearModelSelectionandDiagnostics.....	527

PART V: ADVANCED GRAPHICS

Chapter23:AdvancedPlotCustomization.....	575	Chapter 24: Going Further with the Grammar of Graphics	609
Chapter25: Defining Colors and Plotting in Higher Dimensions	631		
Chapter26:Interactive3DPlots.....	691		
AppendixA:InstallingRandContributedPackages.....	737		
AppendixB:WorkingwithRStudio.....	751		
ReferenceList.....	761		
Index.....	767		

CONTENTS IN DETAIL

PREFACE	xxiii
---------	-------

ACKNOWLEDGMENTS	xxv
-----------------	-----

INTRODUCTION	xxvii
--------------	-------

A Brief History of R	xxvii
About This Book	xxviii
Part I: The Language	xxviii
Part II: Programming	xxix
Part III: Statistics and Probability	xxix
Part IV: Statistical Testing and Modeling	xxix
Part V: Advanced Graphics	xxx
For Students	xxx
For Instructors	xxxi

PART I THE LANGUAGE

1	
GETTING STARTED	3
1. Obtaining and Installing R from CRAN	3
1. Opening R for the First Time	4
1. 1.2.1 Console and Editor Panes	5
2. 1.2.2 Comments	6
1.2.3 Working Directory	7
1.2.4 Installing and Loading R Packages	7
1.2.5 Help Files and Function Documentation	8
1.2.6 Third-Party Editors	11
Saving Work and Exiting R	11
1.3. 1.3.1 Workspaces	11
1.3.2 Scripts	11
Conventions	12
1.4. 1.4.1 Coding	12
1.4.2 Math and Equation References	13
1.4.3 Exercises	14
	14
Exercise 1.1	14

2	NUMERICS, ARITHMETIC, ASSIGNMENT, AND VECTORS	17
2.1	R for Basic Math	1
2.1.1	Arithmetic	7
2.1.2	Logarithms and Exponentials	1
2.1.3	E-Notation	8
	Exercise 2.1	1
2.	Assigning Objects	9
	Exercise 2.2	2
2	Vectors	0
2.3.1	Creating a Vector	2
2.	2.3.2 Sequences, Repetition, Sorting, and Lengths	1
2.	2.3.3 Subsetting and Element Extraction	2
3	Exercise	2
2.3.4	Vector-Oriented Behavior	2
2.3		2
	Exercise	3
		2
		3
3	2.4	2
	MATRICES AND ARRAYS	39
	Exercise	39
3.1	Defining a Matrix	4
3.1.1	Filling Direction	9
3.1.2	Row and Column Bindings	8
3.1.3	Matrix Dimensions	4
3.2	Subsetting	4
	3.2.1 Row, Column, and Diagonal Extractions	4
	3.2.2 Omitting and Overwriting	3
	Exercise 3.1	4
3.3	Matrix Operations and Algebra	6
3.3.1	Matrix Transpose	2
3.3.2	Identity Matrix	4
3.3.3	Scalar Multiple of a Matrix	3
3.3.4	Matrix Addition and Subtraction	4
3.3.5	Matrix Multiplication	4
3.3.6	Matrix Inversion	7
	Exercise 3.2	4
	Multidimensional Arrays	7
3.4	3.4.1 Definition	7
	3.4.2 Subsets, Extractions, and Replacements	7
		4
	Exercise 3.3	8
		4
		9
4	NON-NUMERIC VALUES	4
4.1	Logical Values	59
4.1.1	TRUE or FALSE?	59
	4.1.2 A Logical Outcome: Relational Operators	59
	Exercise 4.1	59

4.1.3 MultipleComparisons:LogicalOperators	6
Exercise 4.2	4
4.1.4 LogicalsAreNumbers!	6
4.1.5 LogicalSubsettingandExtraction	7
Exercise 4.3	6
4.2 Characters	7
4.2.1 CreatingaString	6
4.2.2 Concatenation	8
4.2.3 EscapeSequences	7
4.2.4 SubstringsandMatching	2
Exercise 4.4	7
Factors	2
4.3 4.3.1 IdentifyingCategories	7
4.3.2 DefiningandOrderingLevels	3
4.3.3 CombiningandCutting	7
	4
	7
Exercise 4.5	6
	7
	7
5	7
LISTS AND DATA FRAMES	89
5.1 Lists of Objects	89
5.1.1 DefinitionandComponentAccess	89
5.1.2 Naming	91
5.1.3 Nesting	93
Exercise 5.1	94
5.2 Data Frames	95
5.2.1 Construction	96
5.2.2 AddingDataColumnsandCombiningDataFrames	98
5.2.3 LogicalRecordSubsets	100
Exercise 5.2	101
6	
SPECIAL VALUES, CLASSES, AND COERCION	103
6.1 Some Special Values	10
6.1.1 Infinity	3
6.1.2 NaN	10
Exercise 6.1	4
6.1.3 NA	10
6.1.4 NULL	6
Exercise 6.2	10
Understanding Types, Classes, and Coercion	7
6.2 6.2.1 Attributes	10
6.2.2 ObjectClass	8
6.2.3 Is-DotObject-CheckingFunctions	11
6.2.4 As-DotCoercionFunctions	0
	11
Exercise 6.3	3
	11
	4
	11
	4
	11
	6
	11

7	BASIC PLOTTING	127
7.	Using plot with Coordinate Vectors	12
1	Graphical Parameters	7
7.	7.2.1 AutomaticPlotTypes	12
2	7.2.2 TitleandAxisLabels	9
	7.2.3 Color	12
	7.2.4 LineandPointAppearances	9
	7.2.5 PlottingRegionLimits	13
	Adding Points, Lines, and Text to an Existing Plot	0
7.	Exercise 7.1	13
	The ggplot2 Package	1
3	7.4.1 AQuickPlotwithqplot	13
	7.4.2 SettingAppearanceConstantswithGeoms	3
7.	7.4.3 AestheticMappingwithGeoms	13
	Exercise 7.2	3
4		13
		4
		13
		9
8	READING AND WRITING FILES	147
	8.1 R-ReadyDataSets	14
	8.1.1 Built-inDataSets	14
	8.1.2 ContributedDataSets	14
	8.2 ReadinginExternalDataFiles	8
	8.2.1 TheTableFormat	14
	8.2.2 SpreadsheetWorkbooks	8
	8.2.3 Web-BasedFiles	14
	8.2.4 OtherFileFormats	15
	8.3 WritingOutDataFilesandPlots	6
	8.3.1 DataSets	0
	8.3.2 PlotsandGraphicsFiles	15
	8.4 AdHocObjectRead/WriteOperations	3
	Exercise 8.1	15
		4
		15
		5
		15
		6
PART II	PROGRAMMING	15
		6
		15
		7
9	CALLING FUNCTIONS	165
	9.1 Scoping	16
	9.1.1 Environments.....	165
	9.1.2 SearchPath.....	1.66
	9.1.3 ReservedandProtectedNames.....	168
	Exercise 9.1.....	1.7..0...
		172

9.2	Argument Matching	17
9.2.1	Exact	2
9.2.2	Partial	17
9.2.3	Positional	2
9.2.4	Mixed	17
9.2.5	Dot-Dot-Dot: Use of Ellipses	3
		17
	Exercise 9.2	4
		17
10		5
		17
		19
		6
10.1	ifStatements	17
10.1.1	Stand-AloneStatement	9
10.1.2	elseStatements	18
10.1.3	Using ifelse for Element-wise Checks	0
		18
	Exercise 10.1	3
10.1.4	Nesting and Stacking Statements	18
10.1.5	TheswitchFunction	4
		18
	Exercise 10.2	5
10.2	CodingLoops	18
10.2.1	forLoops	6
		18
	Exercise 10.3	6
10.2.2	whileLoops	18
		9
	Exercise 10.4	19
10.2.3	ImplicitLoopingwithapply	1
		19
	Exercise 10.5	3
10.3	OtherControlFlowMechanisms	19
10.3.1	Declaringbreakornext	3
10.3.2	TherepeatStatement	19
		3
		19
	Exercise 10.6	9
		20
11		0
WRITING FUNCTIONS		20
		21
11.1	ThefunctionCommand	31
11.1.1	FunctionCreation	30
11.1.2	Usingreturn	41
		20
	Exercise 11.1	82
11.2	Arguments	30
11.2.1	LazyEvaluation	22
11.2.2	SettingDefaults	20
11.2.3	Checking for Missing Arguments	22
		21
11.2.4	DealingwithEllipses	21
		12
	Exercise 11.2	21
11.3	Specialized Functions	22
11.3.1	HelperFunctions	5
11.3.2	DisposableFunctions	22
11.3.3	RecursiveFunctions	7
		22
	Exercise 11.3	8
		23
		23
		3
		23

INTRODUCTION



R plays a key role in a wide variety of research and data analysis projects because it makes many modern statistical methods, both simple and advanced, readily available and easy to use. It's true, however, that a beginner to R is often new to programming in general. As a beginner, you must not only learn to use R for your specific data analysis goals but also learn to think like a programmer. This is partly why R has a bit of a reputation for being “hard”—but rest assured, that really isn't the case.

A Brief History of R

R is based heavily on the S language, first developed in the 1960s and 1970s by researchers at Bell Laboratories in New Jersey (for an overview, see, for example, Becker et al., 1988). With a view to embracing open source software, R's developers—Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand—released it in the early 1990s under the

About This Book

The Book of R is intended as a resource to help you get comfortable with R as a first programming language and with the statistical thought that underpins much of its use. The goal is to lay an introductory yet comprehensive foundation for understanding the computational nature of modern data science. The structure of the book seeks to progress naturally in content, first focusing on R as a computational and programming tool and then shifting gears to discuss using R for probability, statistics, and data exploration and modeling. You'll build your knowledge up progressively, and at the end of each chapter, you'll find a section summarizing the important code as a quick reference.

Part I: The Language

Part I, which covers the fundamental syntax and object types used across all aspects of R programming, is essential for beginners. Chapters 2 through 5 introduce the basics of simple arithmetic, assignment, and important object types such as vectors, matrices, lists, and data frames. In Chapter 6, I'll discuss the way R represents missing data values and distinguishes among different object types. You're given a primer on plotting in Chapter 7, using both built-in and contributed functionality (via the `ggplot2` package—see Wickham, 2009); this chapter lays the groundwork for graphical design discussed later in the book. In Chapter 8, I'll cover the fundamentals of reading data in from external files, essential for analysis of your own collected data.

Part II: Programming

Part II focuses on getting you familiar with common R programming mechanisms. First, I'll discuss functions and how they work in R in Chapter 9. Then, in Chapter 10, I'll cover loops and conditional statements, which are used to control the flow, repetition, and execution of your code, before teaching you how to write your own executable R functions in Chapter 11. The examples in these two chapters are designed primarily to help you understand the behavior of these mechanisms rather than to present real-world analyses. I'll also cover some additional topics, such as error handling and measuring function execution time, in Chapter 12.

Part III: Statistics and Probability

With a firm handle on R as a language, you'll shift your attention to statistical thinking in Part III. In Chapter 13, you'll look at important terminology used to describe variables; elementary summary statistics such as the mean, variance, quantiles, and correlation; and how these statistics are implemented in R. Turning again to plotting, Chapter 14 covers how to visually explore your data (with both built-in and `ggplot2` functionality) by using and customizing common statistical plots such as histograms and box-and-whisker plots. Chapter 15 gives an overview of the concepts of probability and random variables, and then you'll look at the R implementation and statistical interpretation of some common probability distributions in Chapter 16.

Part IV: Statistical Testing and Modeling

In Part IV, you're introduced to statistical hypothesis testing and linear regression models. Chapter 17 introduces sampling distributions and confidence intervals. Chapter 18 details hypothesis testing and p -values and demonstrates implementation and interpretation using R; the common ANOVA procedure is then discussed in Chapter 19. In Chapters 20 and 21, you'll explore linear regression modeling in detail, including model fitting and dealing with different types of predictor variables, inferring and predicting, and dealing with variable transformation and interactive effects. Rounding off Part IV, Chapter 22 discusses methods for selecting an appropriate linear model and assessing the validity of that model with various diagnostic tools.

Linear regression represents just one class of parametric models and is a natural starting point for learning about statistical regression. Similarly, the R syntax and output used to fit, summarize, predict from, and diagnose linear models of this kind are much the same for other regression models—so once you're comfortable with these chapters, you'll be ready to tackle the R implementation of more complicated models covered in more advanced texts with relative ease.

error occurs, this literal behavior works in your favor. Look carefully at the commands line by line and try to narrow down the instructions that caused the fault.

- Attempt the practice exercises in this book and check your responses using the suggested solutions—these are all available as R script files on the book's website, <https://www.nostarch.com/bookofr/>. Download the .zip file and extract the .R files, one for each part of the book. Open these in your R session, and you can run the lines like you would any R code to see the output. The short practice exercises are intended to be exactly that—practice—as opposed to being hard or insurmountable challenges. Everything you need to know to complete them will be contained in the preceding sections of that chapter.
- Especially in your early stages of learning, when you're away from this book, try to use R for everything, even for very simple tasks or calculations you might usually do elsewhere. This will force your mind to switch to "R mode" more often, and it'll get you comfortable with the environment quickly.

For Instructors

This book was designed from a three-day workshop, Introduction to R, that

I run at my current institution—the Department of Mathematics & Statistics at the University of Otago in New Zealand—as part of our Statistics Workshops for Postgraduates and Staff (SWoPS). Succeeded by the SWoPS class Statistical Modelling 1 run by two of my colleagues, the aim of Introduction to R is, as the title suggests, to address the programming side of things. Your coverage will naturally depend on your target audience.

Here I provide some recommendations for using the content in *The Book of R* for workshops of similar length to our SWoPS series. Particular chapters can be added or dropped depending on your target workshop duration and students' existing knowledge.

- **Programming Introduction:** Parts I and II. Selected material from

Part V, especially Chapter 23 (Advanced Plot Customization), might also suit the scope of such a course.

- **Statistics Introduction:** Parts III and IV. If a brief introduction to R is warranted beforehand, consider dropping, for example, Chapter 13 from Part III and Chapters 17 through 19 in Part IV and building an initial foundation from content in Part I.

- **Intermediate Programming and Statistics:** Parts II and IV. Consider dropping Chapters 17 through 19 from Part IV to include Part V if the audience is interested in developing plotting skills.

- **R Graphics:** Parts I and V. Depending on audience knowledge, material from Part I may be dropped so that Chapter 14 in Part II can be included (Basic Data Visualization).

If you're planning to go even further and structure a longer course around this book, the practice exercises make particularly good lecture-specific homework to keep students abreast of the skills in R and statistics as they're developed. The main points of the sections making up each chapter are relatively easy to translate into slides that can be initially structured with help from the Contents in Detail.

Part I

THE LANGUAGE

1

GETTING STARTED



R provides a wonderfully flexible programming environment favored by the researchers who do some form of data analysis as part of their work. In this chapter, I'll lay the groundwork for learning and using R, and I'll cover the basics of installing R and certain other things useful to know before you begin.

1.1 Obtaining and Installing R from CRAN

R is available for Windows, OS X, and Linux/Unix platforms. You can find the main collection of R resources online at the Comprehensive R Archive Network (CRAN). If you go to the R project website at <http://www.r-project.org/>, you can navigate to your local CRAN mirror and download the installer relevant to your operating system. Section A.1 provides step-by-step instructions for installing the base distribution of R.

1.2 Opening R for the First Time

R is an interpreted language that's strictly case- and character-sensitive, which means that you enter instructions that follow the specific syntactic rules of the language into a console or command-line interface. The software then interprets and executes your code and returns any results.

NOTE *high-level Level*
R is what's known as a programming language. It refers to the level of abstraction away from the fundamental details of computer execution. That is, a low-level language will require you to do things such as manually manage the machine's memory allotments, but with a high-level language like R, you're fortunately spared these technicalities.

When you open the base R application, you're presented with the R console; Figure 1-1 shows a Windows instance, and the left image of Figure 1-2 shows an example in OS X. This represents R's naturally incorporated *graphical user interface (GUI)* and is the typical way base R is used.

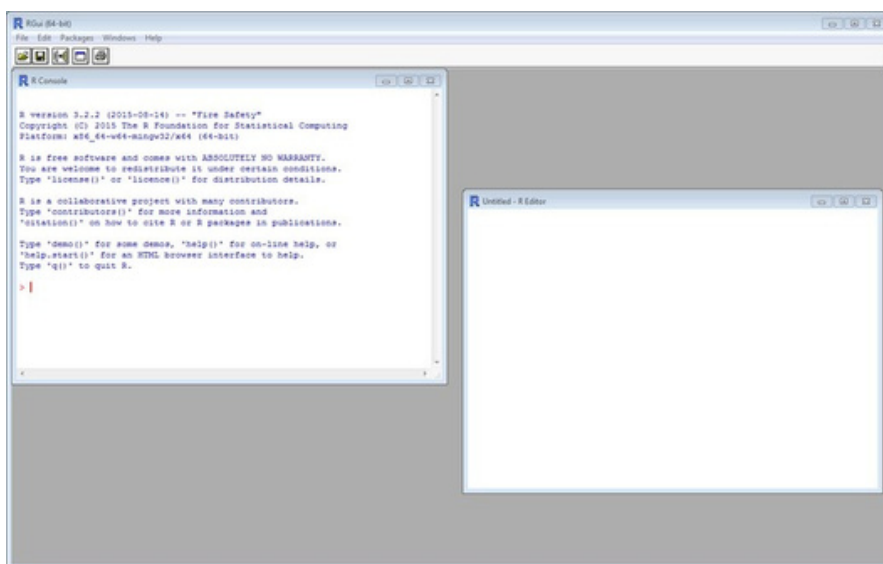


Figure 1-1: The R GUI application (default configuration) in Windows

The functional, “no-frills” appearance of the interpreter, which in my experience has struck fear into the heart of many an undergraduate, stays true to the very nature of the software—a blank statistical canvas that can be used for any number of tasks. Note that OS X versions use separate windows for the console and editor, though the default behavior in Windows is to contain these panes in one overall R window (you can change this in the GUI preferences if desired; see Section 1.2.1).

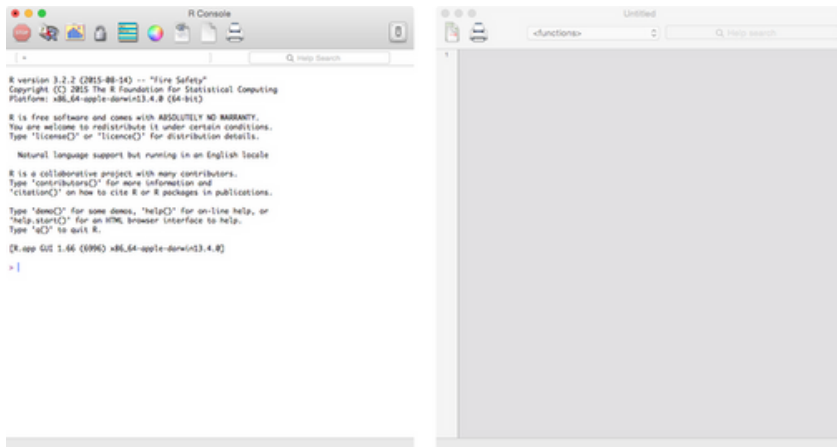


Figure 1-2: The base R GUI console pane (left) and a newly opened instance of the built-in editor (right) in OS X

NOTE

As I've just done, in some parts of the book I'll refer specifically to the R GUI functionality in Windows and OS X, given these are the two platforms most often used by beginners. As well as Linux/Unix implementations, it's possible to run R from a terminal or shell or, indeed, in the alternative batch mode. The vast majority of the code in this book is functional in all settings.

1.2.1 Console and Editor Panes

There are two main window types used for programming R code and viewing output. The console or command-line interpreter that you've just seen is where all execution takes place and where all textual and numeric output is provided. You may use the R console directly for calculations or plotting. You would typically use the console directly only for short, one-line commands.

By default, the R *prompt* that indicates R is ready and awaiting a command is a `>` symbol, after which a text cursor appears. To avoid confusion with the mathematical symbol for "greater than," `>`, some authors (including me) prefer to modify this. A typical choice is `R>`, which you can set as follows:

With the cursor placed at the prompt, you can use the keyboard up arrow (↑) and down arrow (↓) to scroll through any previously executed commands; this is useful when making small tweaks to earlier commands.

For longer chunks of code and function authoring, it's more convenient to first write your commands in an *editor* and execute them in the console only when you're done. There is a built-in R code editor for this purpose. The R *scripts* you write in the code editor are essentially just plain-text files with a `.R` extension.

You can open a new instance of the editor using the R GUI menus (for example, File → New script in Windows or File → New Document in OS X). The built-in editor features useful keystroke shortcuts (for example, CTRL-R in Windows or [⌘]-RETURN in OS X), which automatically send lines to the console. You can send the line upon which the cursor sits, a highlighted line, a highlighted part of a line, or a highlighted chunk of code. It's common to have multiple editor panes open at once when working with multiple R script files; keystroke code submissions simply operate with respect to the currently selected editor.

Aesthetics such as coloring and character spacing of both the console and editor can be tailored to a certain extent depending on operating system; you simply need to access the relevant GUI preferences. Figure 1-3 shows the R GUI preferences in Windows (Edit → GUI preferences...) and OS X (R → Preferences...). A nice feature of the OS X version of R in particular is the code-coloring and bracket-matching features of the editor, which can improve the authoring and readability of large sections of code.

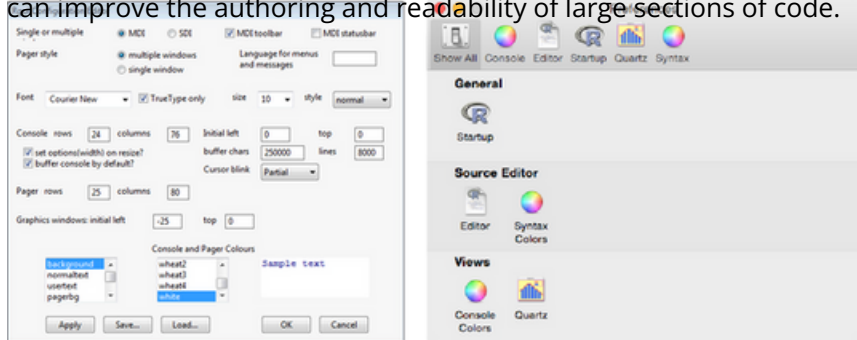


Figure 1-3: The R GUI preferences in Windows (left) and OS X (right)

1.2.2 Comments

In R, you can annotate your code with *comments*. Just preface the line with a hash mark (#), and anything that comes thereafter will be ignored by the interpreter. For example, executing the following in the console does nothing but return you to the prompt:

```
R> # This is a comment in R...
```

Comments can also appear after valid commands.

```
R> 1+1 # This works out the result of one plus one!  
[1]2
```

If you're writing large or complicated chunks of code in the editor, this kind of annotation can be helpful to others (and indeed yourself!) who want to understand what your code is doing.

1.2.3 WorkingDirectory

An active R session always has a *working directory* associated with it. Unless you explicitly specify a file path when saving or importing data files, R will use this working directory by default. To check the location of the working directory, use the `getwd()` function.

```
R> getwd()  
[1] "/Users/tdavies"
```

File paths are always enclosed in double quotation marks, and R uses forward slashes, not backslashes, when specifying folder locations.

You can change the default working directory using the function `setwd()` as follows:

```
R> setwd("/folder1/folder2/folder3/")
```

You may provide your file path relative to the current working directory or fully (in other words, from a system root drive). Either way, it's important to remember the case-sensitive nature of R; you must match the naming and punctuation of any folder names exactly or an error will be thrown.

That said, if you're happy specifying a full and correct file path each time you read or write a file (there are further details in Chapter 8), then the files of interest can reside anywhere on your computer.

1.2.4 Installing and Loading R Packages

The base installation of R comes ready with a plethora of built-in commands for numeric calculations, common statistical analyses, and plotting and visualization. These commands can be used right from the outset and needn't

be loaded or imported in any way. I'll refer to these functions as *built-in* or *ready-to-use* in this text.

Slightly more specialized techniques and data sets are contained within *packages* (also referred to as *libraries*) of code. Using contributed packages is common, and you'll be doing so throughout this book, so it's important to get comfortable with installing and loading the required libraries.

Section A.2 covers the relevant details concerning package download and installation from CRAN, but I'll provide a brief overview here.

Loading Packages

There are a small number of recommended packages that are included with the base distribution of R (listed in Section A.2.2). They don't need to be installed separately, but to use them, you do need to load them by calling `library()`. One package you'll use in this book is named MASS (Venables and Ripley, 2002). To load it (or any other installed package) and gain access to its functions and data sets, simply execute `library()` at the prompt as follows:

```
R> library("MASS")
```

Note that calling `library` provides access to a package's functionality only for the running R session. When you close R and reopen a fresh instance, you'll need to reload any packages you want to use.

Installing Packages

There are thousands of contributed packages not included with the typical R installation; to make them loadable in R, you must first download and install them from a repository (usually CRAN). The easiest way to do this is by using the `install.packages` function directly at the R prompt (for this you need an Internet connection).

For example, one such package is `ks` (Duong, 2007), which you'll use in Chapter 26. Executing the following will attempt to connect to your local CRAN mirror and download and install `ks`, as well as several packages upon which it relies (called *dependencies*):

```
R> install.packages("ks")
```

The console will show running output as the procedure completes. You need to install a package only once; thereafter it will be available for your R installation. You can then load your installed package (like `ks`) in any newly opened instance of R with a call to `library`, just as you did for `MASS`.

Section A.2.3 offers more detail on package installation.

Updating Packages

The maintainers of contributed packages periodically provide version updates to fix bugs and add functionality. Every so often, you might want to check for updates to your collection of installed packages. From the R prompt, a simple execution of the following will attempt to connect to your set package repository (defaulting to CRAN), looking for versions of all your installed packages that are later than those you currently have.

```
R> update.packages()
```

Section A.3 offers more details about updating packages and Section A.4 discusses alternate CRAN mirrors and repositories.

1.2.5 Help Files and Function Documentation

R comes with a suite of *help files* that you can use to search for particular functionality, to seek information on precisely how to use a given function and specify its *arguments* (in other words, the values or objects you supply to the function when you execute it), to clarify the role of arguments in the operations, to learn about the form of any returned objects, to provide possible examples of using the function, and to get details on how you may cite any software or data sets.

To access the help file for a given command or other object, use the help function at the console prompt or use the convenient shortcut `?.` For example, consider the ready-to-use arithmetic mean function, `mean`.

`R> ?mean`

This brings up the file in the top image of Figure 1-4.



Figure 1-4: The R help file for the function `mean` (top) and the results of a help search for the string "mean" (bottom) in OS X

If you're unsure of the precise name of the desired function, you can search the documentation across all installed packages using a character string (a statement in double quotes) passed to `help.search`, or you can use `??` as a shortcut:

```
R> ?? "mean"
```

This search brings up a list of functions, with their host packages and descriptions, whose help files contain the string of interest, as shown in the bottom image of Figure 1-4 (the highlighted entry is that of the `arithmetic::mean`).

All help files follow the general format shown in the top image of Figure 1-4; the length and level of detail in the file typically reflect the complexity of the operations carried out by the function. Most help files include the first three items listed here; the others are common but optional:

- The *Description* section provides a short statement about the operations carried out.
- The *Usage* section specifies the form of the function in terms of how it should be passed to the R console, including the natural order of the arguments and any default values (these are the arguments that are shown being set using `=`).
- In the *Arguments* section, more detail is given about what each argument does as well as the possible values that they're allowed to take on.
- The nature of the object that's returned by the function (if anything) is specified under *Value*.
- The *References* section provides relevant citations for the command or the methodology behind the function.
- The help files for related functions are linked under *See Also*.
- *Examples* provides executable code that you can copy and paste into the console, demonstrating the function in action.

There are several more possible fields in a help file—functions with slooeng-er explanations often contain a *Details* section after the *Arguments* section. Common traps or mistakes made when calling the function are

usually

placed in a *Warnings* section, and additional information can be placed in *Notes*.

Although they might seem quite technical when you're first starting out, I encourage you to keep looking at help files—even if you already know how the function works, getting comfortable with the layout and organization of function documentation is an important part of becoming a

skilled

R user.

1.2.6 Third-Party Editors

The popularity of R has led to the development of several third-party code editors, or compatible plug-ins for existing code-editing software, which can enhance the experience of coding in R.

One noteworthy contribution is RStudio (RStudio Team, 2015). This is an integrated development environment (IDE) available free for Windows, OS X, and Linux/Unix platforms at <http://www.rstudio.com/>. RStudio includes a direct-submission code editor; separate point-and-click panes for things such as file, object, and project management; and the creation of markup documents incorporating R code. Appendix B discusses RStudio and its capabilities in more detail.

Use of any third-party editor, including RStudio, is by and large a personal choice. In this book, I simply assume use of the typical base R GUI application.

1.3 Saving Work and Exiting R

So, you've spent a few hours coding in R, and it's time to go home? When saving work in R, you need to pay attention to two things: any R objects that have been created (and stored) in the active session and any R script files written in an editor.

1.3.1 Workspaces

You can use the GUI menu items (for example, under File in Windows and under Workspace in OS X) to save and load *workspace image* files. An

R workspace image contains all the information held in the R session at the time of exit and is saved as a *.RData* file. This will include all objects you've created and stored (in other words, *assigned*) within the session (you'll see how to do this in Chapter 2), including those that may have been loaded from a previous workspace file.

Essentially, loading a stored *.RData* file allows you to "pick up from where you left off." At any point in an R session, you can execute `ls()` at the prompt, which lists all objects, variables, and user-defined functions currently present in the active workspace.

Alternatively, you can use the R commands `save.image` and `load` at the console for handling workspace *.RData* files—both of these functions contain a file argument to which you pass the folder location and name of the target *.RData* file (see the corresponding help files? `save.image` and `?load` for further information on the use of these).

Note that saving a workspace image in this way doesn't retain the functionality of any contributed packages that were loaded in the previously active R session. As mentioned in Section 1.2.4, you'll need to use

library

to load any packages required for your work for each new instance of R.

The quickest way to exit the software is to `entqe(r)` at the prompt:

```
R> q()
```

Simply exiting the console will bring up a dialog asking if you'd like to save the workspace image. In this case, choosing to save doesn't open a file browser to name your file but creates (or overwrites) a "no-name" file as one with a *.RData* extension in your working directory (refer to Section 1.2.3). If an unnamed *.RData* file exists in the default working directory when a new instance of R is opened, the program will automatically load that default workspace—if that has happened, you'll be notified in the console's welcoming text.

NOTE

.RData

Alongside the file, R will automatically save a file containing a line-by-line history of all the commands executed in the console for the associated workspace in the same directory. It's this history file that allows you to scroll through the previously executed commands using the keyboard directional arrows, as noted earlier.

1.3.2 Scripts

For tasks requiring anything more than a handful of commands, you'll usually want to work in the built-in code editor. Saving your R scripts is therefore at least as important as saving a workspace, if not more so.

You save editor scripts as plain-text files with a *.R* extension (noted in Section 1.2.1); this allows your operating system to associate these files with the R software by default. To save a script from the built-in editor, ensure the editor is selected and navigate to File

Save (or press CTRL-S in Windows or ⌘-S in OS X). To open a previously saved script, select File

Open script... (CTRL-O) in Windows or File

Open Document... (⌘-O) in OS X.

Often, you won't really need to save a workspace *.RData* file if your script files are saved. Once any required commands in a saved script are reexecuted in a new R console, the objects created previously (in other words, those contained within a saved *.RData* file) are simply created once more. This can be useful if you're working on multiple problems at one time because it can be easy to mistakenly overwrite an object when relying solely on the stand-alone default workspace. Keeping your collection of R scripts separate is therefore a simple way to separate several projects without needing to worry about overwriting anything important that may have been stored previously.

R also provides a number of ways to write individual objects, such as data sets and image files of plots, to disk, which you'll look at in Chapter 8.

1.4 Conventions

There are a few conventions that I'll follow in the book in terms of the presentation of code and math.

1.4.1 Coding

As mentioned, when you code with R, you execute the code in the console, possibly after writing the script in the editor first. The following points are important to note:

- R code that's entered directly into the console for execution is shown preceded by the R> prompt and followed by any output displayed in the console. For example, this simple division of 14 by 6 from Section 2.1.1 looks like this:

```
R> 14/6
[1] 2.333333
```

If you want to copy and paste console-executed code directly from the text of the book, you'll need to omit the R> prompt.

- For code that should be written in the editor before it's executed in the console, I'll indicate as such in the text, and the code will be presented *without* the prompt. The following example comes from Section 10.2.1:

```
for(myitem in 5:7){
  cat("--BRACED AREA BEGINS--\n")
  cat("the current item is",myitem,"\n")
  cat("--BRACED AREA ENDS--\n\n")
}
```

My preferred coding style for actually arranging and indenting chunks like this will become clearer as you progress through Part II.

- There will occasionally be long lines of code (either executed directly in the console or written in the editor), which, for the sake of print, will be split and indented at an appropriate place to fit on the page. For example, take this line from Section 6.2.2:

```
R> ordfac.vec <-
factor(x=c("Small","Large","Large","Regular","Small"),
levels=c("Small","Regular","Large"),
ordered=TRUE)
```

Although this can be written out as a single line when using R, you can also break the line at a comma (in this case, the comma splits the arguments to the factor function). The broken line will be indented

to the level of the opening parenthesis of the relevant command. Both forms—single line or split—will work as is when executed in R.

Lastly, in a couple of places when the console output is lengthy and not

essential to your understanding of the immediate content, it'll be pressed for the sake of print. I'll say as much in the text, and you'll

see

the designation *--snip--* in the affected chunk of code.

1.4.2 Math and Equation References

Mathematics and equations that appear in this book (mainly in Parts III and IV) will be kept to a minimum, but in certain sections it's sometimes necessary to go into a little mathematical detail.

Important equations will be presented on their own lines as follows:

$$y=4x \text{ (1.1)}$$

Equations will be numbered in parentheses, and references to equations in the text will use these parenthesized numbers and may or may not be preceded by *Equation*. For example, you'll see equations referred to in both of the following ways:

- As per Equation (1.1), $y=8$ when $x=2$.
- Inversion of (1.1) yields $x=y/4$.

When numeric results are rounded to a certain level, they'll be noted as such according to the number of *decimal places*, abbreviated to *d.p.* Here are some examples:

- The famous geometric value π is given as $\pi=3.1416$ (rounded to 4 d.p.).
- Setting $x=1.467$ in (1.1) results in $y=5.87$ (2 d.p.).

1.4.3 Exercises

Exercise questions in the chapters appear in a rounded box:

Exercise 1.1

- Say the word *cat* aloud.
- Using nothing but your brain, find the solution to $1+1$.

These exercises are optional. If you choose to tackle them, they are intended to be completed as and when they appear in the text to help you practice and understand the specific content and code in the sections that immediately precede them.

All the data sets you'll use in this book for coding and plotting examples are available either as built-in R objects or as part of one of the contributed packages you'll install. These packages will be noted in the relevant text (for a short list of them, see Section A.2.3).

For your convenience, all code examples in this book, as well as complete suggested solutions to *all* practice exercises, are freely available as runnable *.R* script files on the book's web page at <https://www.nostarch.com/bookofr/>.

You should think of these solutions (and any accompanying commentary) as “suggested” because there are often multiple ways to perform a certain task in R, which may not necessarily be any better or worse than those supplied.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
options	Set various R options	Section 1.2.1, p. 5
#	A comment (ignored by interpreter)	Section 1.2.2, p. 6
getwd	Print current working directory	Section 1.2.3, p. 7
setwd	Set current working directory	Section 1.2.4, p. 7
library	Load an installed package	Section 1.2.4, p. 8
install.packages	Download and install package	Section 1.2.4, p. 8
update.packages	Update installed packages	Section 1.2.5, p. 9
help or ?	Function/object help file	Section 1.2.5, p. 10
help.search or ??	Search help files	Section 1.3.1, p. 12
q	Quit R	

2

NUMERICS, ARITHMETIC, ASSIGNMENT, AND VECTORS



In its simplest role, R can function as a mere desktop calculator. In this chapter, I'll discuss how to use the software for arithmetic. I'll also show how to store results so you can use them later in other calculations. Then, you'll learn about vectors, which let you handle multiple values at once. Vectors are an essential tool in R, and much of R's functionality was designed with vector operations in mind. You'll examine some common and useful ways to manipulate vectors and take advantage of vector-oriented behavior.

2.1 RforBasicMath

All common arithmetic operations and mathematical functionality are ready to use at the console prompt. You can perform addition, subtraction, multiplication, and division with the symbols `+`, `-`, `*`, and `/`, respectively. You can create exponents (also referred to as *powers* or *indices*) using `^`, and control the order of the calculations in a single command using parentheses, `()`.

2.1.1 Arithmetic

In R, standard mathematical rules apply throughout and follow the usual left-to-right order of operations: parentheses, exponents, multiplication, division, addition, subtraction (PEMDAS). Here's an example in the console:

```
R> 2+3
[1] 5
R> 14/6
[1] 2.333333
R> 14/6+5
[1] 7.333333
R> 14/(6+5)
[1] 1.272727
R> 3^2
[1] 9
R> 2^3
[1] 8
```

You can find the square root of any non-negative number with the `sqrt` function. You simply provide the desired number to `x` as shown here:

```
R> sqrt(x=9)
[1] 3
R> sqrt(x=5.311)
[1] 2.304561
```

When using R, you'll often find that you need to translate a complicated arithmetic formula into code for evaluation (for example, when replicating a calculation from a textbook or research paper). The next examples provide a mathematically expressed calculation, followed by its execution in R:

$$\frac{1023 + \frac{\times 60}{8}}{8} - 3$$

```
R> 10^2+3*60/8-
3 [1] 119.5
```

$$\frac{53(62 - \frac{\times 613}{4})}{4 + 64 - 22.251 - 22 + 1 - 4}$$

```
R> 5^3*(6-2)/(61-
3+4) [1] 8.064516
```

$$\frac{4 + 64 - 22.251 - 22 + 1 - 4}{1}$$

```
R> 2^(2+1)-4+64^((-2)^(2.25-
1/4)) [1] 16777220
```

$$\left(\frac{0.44 \times (1 - 0.44)}{34} \right)^{\frac{1}{2}}$$

```
R> (0.44*(1-0.44)/34)^(1/2)
[1] 0.08512966
```

Note that some R expressions require extra parentheses that aren't present in the mathematical expressions. Missing or misplaced parentheses are common causes of arithmetic errors in R, especially when dealing with exponents. If the exponent is itself an arithmetic calculation, it must always appear in parentheses. For example, in the third expression, you need parentheses around $2.25 - 1/4$. You also need to use parentheses if the number being raised to some power is a calculation, such as the expression $22 + 1$ in the third example. Note that R considers a negative number a calculation because it interprets, for example, -2 as $-1 * 2$. This is why you also need the parentheses around -2 in that same expression. It's important to highlight these issues early because they can easily be overlooked in large chunks of code.

2.1.2 Logarithms and Exponentials

You'll often see or read about researchers performing a *log transformation*

on certain data. This refers to rescaling numbers according to the *logarithm*. When supplied a given number x and a value referred to as a *base*, the log-arithm calculates the power to which you must raise the base to get to x .

For example, the logarithm of $x = 243$ to base 3 (written mathematically as $\log_3 243$) is 5, because $3^5 = 243$. In R, the log transformation is achieved with the function `log`. You supply `log` with the number to transform, assigned to the value `x`, and the base, assigned to `base`, as follows:

```
[1] 5
```

Here are some things to consider:

- Both x and the base must be positive.
- The log of any number x when the base is equal to x is 1.
- The log of $x = 1$ is always 0, regardless of the base.

There's a particular kind of log transformation often used in mathematics called the *natural log*, which fixes the base at a special mathematical number—*Euler's number*. This is conventionally written as e and is approximately equal to 2.718.

Euler's number gives rise to the *exponential function*, defined as e raised to the power of x , where x can be any number (negative, zero, or positive).

The exponential function, $f(x) = e^x$, is often written as `exp(x)` and represents the *inverse* of the natural log such that $\exp(\log(x)) = x$.

The R command for the exponential function is `exp`:

```
R> exp(x=3)
[1] 20.08554
```

The default behavior of `log` is to assume the natural log:

```
R>
log(x=20.08554)
[1] 3
```

You must provide the value of base yourself if you want to use a value other than *e*. The logarithm and exponential functions are mentioned here because they become important later on in the book—many statistical methods use them because of their various helpful mathematical properties.

2.1.3 E-Notation

When R prints large or small numbers beyond a certain threshold of significant figures, set at 7 by default, the numbers are displayed using the classic scientific e-notation. The e-notation is typical to most programming languages—and even many desktop calculators—to allow easier interpretation of extreme values. In e-notation, any number *x* can be expressed as *key*, which represents exactly *x* × 10^{*y*}. Consider the number 2,342,151,012,900. It could, for example, be represented as follows:

- 2.3421510129e12, which is equivalent to writing 2.3421510129 × 10¹²
- 234.21510129e10, which is equivalent to writing 234.21510129 × 10¹⁰

You could use any value for the power of *y*, but standard e-notation uses the power that places a decimal just after the first significant digit. Put simply, for a *positive* power +*y*, the e-notation can be interpreted as “move the decimal point *y* positions to the *right*.” For a *negative* power –*y*, the interpretation is “move the decimal point *y* positions to the *left*.” This is exactly how R presents e-notation:

```
[1] 2.342151e+12
R>
0.0000002533
```

In the first example, R shows only the first seven significant digits and hides the rest. Note that no information is lost in any calculations even if R hides digits; the e-notation is purely for ease of readability by the user, and the extra digits are still stored by R, even though they aren't shown. Finally, note that R must impose constraints on how extreme a number can be before it is treated as either infinity (for large numbers) or zero (for small numbers). These constraints depend on your individual system, and

I'll discuss the technical details a bit more in Section 6.1.1. However, any modern desktop system can be trusted to be precise enough by default for most computational and statistical endeavors in R.

Exercise 2.1

a. Using R, verify that

$$6a + 42$$

$$\frac{4}{3} \cdot 29.50556 - 3.62 =$$

when $a = 2.3$.

b. Which of the following squares negative 4 and adds 2 to the result?

i. $(-4)^2 + 2$

ii. $-4^2 + 2$

iii. $(-4)^{(2+2)}$

iv. $-4^{(2+2)}$

c. Using R, how would you calculate the square root of half of the average of the numbers 25.2, 15, 16.44, 15.3, and 18.6?

d. Find $\log$

e 0.3.

. Compute the exponential transform of your answer to (d).

e. Identify R's representation of

-0.00000000423546322 when

f. printing this number to the console.

2.2 Assigning Objects

So far, R has simply displayed the results of the example calculations by printing them to the console. If you want to save the results and perform further operations, you need to be able to *assign* the results of a given computation to an *object* in the current workspace. Put simply, this amounts to storing some item or result under a given name so it can be accessed later, without having to write out that calculation again. In this book, I will use the terms *assign* and *store* interchangeably. Note that some programming books refer to a stored object as a *variable* because of the ability to easily overwrite that object and change it to something different, meaning that what it represents can vary throughout a session. However, I'll use the term *object* throughout this book because we'll discuss variables in Part III as a distinctly different statistical concept.

You can specify an assignment in R in two ways: using arrow notation (`<-`) and using a single equals sign (`=`). Both methods are shown here:

```
<-
```

```
R> x <- -5
```

```
R> x
```

```
[1] -5
```

```

R> x = x + 1 # this overwrites the previous value of
x R>x
[1] -4

R> mynumber = 45.2

R> y <- mynumber*x
R>y
[1] -180.8

R> ls()
[1] "mynumber" "x"

```

As you can see from these examples, R will display the value assigned to an object when you enter the name of the object into the console. When you use the object in subsequent operations, R will substitute the value you assigned to it. Finally, if you use the `ls` command (which you saw in Section 1.3.1) to examine the contents of the current workspace, it will reveal the names of the objects in alphabetical order (along with any other previously created items).

Although `=` and `<-` do the same thing, it is wise (for the neatness of code if nothing else) to be consistent. Many users choose to stick with the `<-`, however, because of the potential for confusion in using the `=` (for example, I clearly didn't mean that x is *mathematically* equal to $x + 1$ earlier). In this book, I'll do the same and reserve `=` for setting function arguments, which begins in Section 2.3.2. So far you've used only numeric values, but note that the procedure for assignment is universal for all types and classes of objects, which you'll examine in the coming chapters.

Objects can be named almost anything as long as the name begins with a letter (in other words, not a number), avoids symbols (though underscores and periods are fine), and avoids the handful of “reserved” words such as those used for defining special values (see Section 6.1) or for controlling code flow (see Chapter 10). You can find a useful summary of these naming rules in Section 9.1.2.

Exercise 2.2

- a. Create an object that stores the value 32.
- Overwrite your object in (a) by itself divided by 2.33. Print the result to the console.
- b. Create a new object with the value -8.2×10^{-13} .
- Print directly to the console the result of multiplying (b) by (c).
- c. Print directly to the console the result of multiplying (b) by (c).
- d. Print directly to the console the result of multiplying (b) by (c).
- Print directly to the console the result of multiplying (b) by (c).

2.3 Vectors

Often you'll want to perform the same calculations or comparisons upon multiple entities, for example if you're rescaling measurements in a data set. You could do this type of operation one entry at a time, though this is clearly not ideal, especially if you have a large number of items. R provides a far more efficient solution to this problem with *vectors*.

For the moment, to keep things simple, you'll continue to work with numeric entries only, though many of the utility functions discussed here may also be applied to structures containing non-numeric values. You'll start looking at these other kinds of data in Chapter 4.

2.3.1 Creating a Vector

The vector is the essential building block for handling multiple items in R. In a numeric sense, you can think of a vector as a collection of observations or measurements concerning a single variable, for example, the heights of 50 people or the number of coffees you drink daily. More complicated data structures may consist of several vectors. The function for creating a vector is the single letter *c*, with the desired entries in parentheses separated by commas.

```
R> myvec <- c(1,3,1,42)
R> myvec
[1] 1 3 1 42
```

Vector entries can be calculations or previously stored items (including vectors themselves).

```
R> foo <- 32.1
R> myvec2 <- c(3,-3,2,3.45,1e+03,64^0.5,2+(3-1.1)/9.44,foo)
R> myvec2
[1] 3.000000 -3.000000 2.000000 3.450000 1000.000000 8.000000
[7] 2.201271 32.100000
```

myvec2 This code created a new vector assigned to the object `myvec2`. Some of the entries are defined as arithmetic expressions, and it's the result of the expression that's stored in the vector. The last element, `foo`, is an existing numeric object defined as 32.1. Let's look at another example.

```
R> myvec3 <- c(myvec,myvec2)
R> myvec3
[1] 1.000000 3.000000 1.000000 42.000000 3.000000 -3.0000
[7] 2.000000 3.450000 1000.000000 8.000000 2.201271 0 0
[13] 32.100000
```

This code creates and stores yet another vector, `myvec3`, which contains the entries of `myvec` and `myvec2` appended together in that order.

2.3.2 Sequences, Repetition, Sorting, and Lengths

Here I'll discuss some common and useful functions associated with R vectors: `seq`, `rep`, `sort`, and `length`.

Let's create an equally spaced sequence of increasing or decreasing numeric values. This is something you'll need often, for example when programming loops (see Chapter 10) or when plotting data points (see Chapter 7). The easiest way to create such a sequence, with numeric values separated by intervals of 1, is to use the colon operator.

```
R> 3:27
```

```
[1] 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

```
3:27
```

The example should be read as “from 3 to 27 (by 1).” The result is a numeric vector just as if you had listed each number manually in parentheses with `c`. As always, you can also provide either a previously stored value or

a (strictly parenthesized) calculation when using the colon operator:

```
R> foo <- 5.3
```

```
R> bar <- foo:(-47+1.5)
```

```
R> bar
```

```
[1] 5.3 4.3 3.3 2.3 1.3 0.3 -0.7 -1.7 -2.7 -3.7 -4.7
```

```
[12] -5.7 -6.7 -7.7 -8.7 -9.7 -10.7 -11.7 -12.7 -13.7 -14.7 -15.7
```

```
[23] -16.7 -17.7 -18.7 -19.7 -20.7 -21.7 -22.7 -23.7 -24.7 -25.7 -26.7
```

```
[34] -27.7 -28.7 -29.7 -30.7 -31.7 -32.7 -33.7 -34.7 -35.7 -36.7 -37.7
```

```
[45] -38.7 -39.7 -40.7 -41.7 -42.7 -43.7 -44.7
```

Sequences with `seq`

You can also use the `seq` command, which allows for more flexible creations of sequences. This ready-to-use function takes in a from value, a to value, and a value, and it returns the corresponding sequence as a numeric vector.

```
R> seq(from=3,to=27,by=3)
```

```
[1] 3 6 9 12 15 18 21 24 27
```

This gives you a sequence with intervals of 3 rather than 1. Note that these kinds of sequences will always start at the from number but will not always include the to number, depending on what you are asking R to increase (or decrease) them by. For example, if you are increasing (or decreasing) by even numbers and your sequence ends in an odd number, the final number won't be included. Instead of providing a value, however, you can specify a length.out value to produce a vector with that many numbers, evenly spaced between the from and to values.

```
R> seq(from=3,to=27,length.out=40)
```

```
[1] 3.000000 3.615385 4.230769 4.846154 5.461538 6.076923 6.692308
```

```
[8] 7.307692 7.923077 8.538462 9.153846 9.769231 10.384615 11.000000
```

```
[15] 11.615385 12.230769 12.846154 13.461538 14.076923 14.692308 15.307692
```

```
[22] 15.923077 16.538462 17.153846 17.769231 18.384615 19.000000 19.615385
[29] 20.230769 20.846154 21.461538 22.076923 22.692308 23.307692
23.923077 [36] 24.538462 25.153846 25.769231 26.384615 27.000000
```

length.out 40 By setting to, you make the program print exactly 40 evenly spaced numbers from 3 to 27.

For decreasing sequences, the use of must be negative. Here's an example:

```
R> foo <- 5.3
R> myseq <- seq(from=foo,to=(-47+1.5),by=-2.4)
R> myseq
[1] 5.3 2.9 0.5 -1.9 -4.3 -6.7 -9.1 -11.5 -13.9 -16.3 -18.7 -21.1
[13] -23.5 -25.9 -28.3 -30.7 -33.1 -35.5 -37.9 -40.3 -42.7 -45.1
```

This code uses the previously stored object as the value for and uses the parenthesized calculation as the value. Given those values (that is, with being greater than), these sequence can progress only in negative steps; directly above, we set to be . The use of length.out to create decreasing sequences, however, remains the same (it would make no sense to specify a "negative length"). For the same and to values, you can create a decreasing sequence of length 5 easily, as shown here:

```
R> myseq2 <- seq(from=foo,to=(-47+1.5),length.out=5) R> myseq2
[1] 5.3 -7.4 -20.1 -32.8 -45.5
```

There are shorthand ways of calling these functions, which you'll learn about in Chapter 9, but in these early stages I'll stick with the explicit usage.

Repetition with rep

Sequences are extremely useful, but sometimes you may want simply to repeat a certain value. You do this using .

```
R> rep(x=1,times=4)
[1] 1 1 1 1
R> rep(x=c(3,62,8.3),times=3)
[1] 3.0 62.0 8.3 3.0 62.0 8.3 3.0 62.0 8.3
R> rep(x=c(3,62,8.3),each=2)
[1] 3.0 3.0 62.0 62.0 8.3 8.3
R> rep(x=c(3,62,8.3),times=3,each=2)
[1] 3.0 3.0 62.0 62.0 8.3 8.3 3.0 3.0 62.0 62.0 8.3 8.3 3.0 3.0 62.0
[16] 62.0 8.3 8.3
```

The function is given as a single value or a vector of values as its argument, as well as a value for the arguments and each. The value for provides the number of times to repeat, and each provides the

number of times to repeat each element of `x`. In the first line directly above, you simply repeat a single value four times. The other examples first use `rep` and `times` on a vector to repeat the entire vector, then use `each` to repeat each member of the vector, and finally use both `times` and `each` at once.

If neither `times` nor `each` is specified, R's default is to treat the values of `times` and `each` as 1 so that a call of `rep(x=c(3,62,8.3))` will just return the originally supplied `x` with no changes.

As with `seq`, you can include the result of `rep` in a vector of the same data

R> foo <- 4
type, as shown in the following example:

```
c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1)) [1]
3.00 8.30 32.00 32.00 32.00 32.00 -2.00 -1.25 -0.50 0.25 1.00
```

Here, I've constructed a vector where the third to sixth entries (inclusive) are governed by the evaluation of a `rep` command—the single value 32 repeated `foo` times (where `foo` is stored as 4). The last five entries are the result of an evaluation of `seq`, namely a sequence from -2 to 1 of length `foo+1` (5).

Sorting with `sort`

Sorting a vector in increasing or decreasing order of its elements is another simple operation that crops up in everyday tasks. The conveniently named `sort` function does just that.

```
R> sort(x=c(2.5,-1,-10,3.44),decreasing=FALSE)
[1]-10.00 -1.00 2.50 3.44
R> sort(x=c(2.5,-1,-10,3.44),decreasing=TRUE)
[1] 3.44 2.50 -1.00-10.00
R> foo <- seq(from=4.3,to=5.5,length.out=8)
R> foo
[1] 4.300000 4.471429 4.642857 4.814286 4.985714 5.157143 5.328571 5.500000
R> bar <- sort(x=foo,decreasing=TRUE)
R> bar
[1] 5.500000 5.328571 5.157143 4.985714 4.814286 4.642857 4.471429 4.300000
R> sort(x=c(foo,bar),decreasing=FALSE)
[1] 4.300000 4.300000 4.471429 4.471429 4.642857 4.642857 4.814286 4.814286
[9] 4.985714 4.985714 5.157143 5.157143 5.328571 5.328571 5.500000 5.500000
```

```
sort
x decreasing
```

The function is pretty straightforward. You supply a vector to the function as the argument, and a second argument, `decreasing`, indicates the order in which you want to sort. This argument takes a type of value you have not yet met: one of the all-important *logical* values. A logical value

can be only one of two specific, case-sensitive values: TRUE or FALSE. Generally speaking, logicals are used to indicate the satisfaction or failure of a certain *condition*, and they form an integral part of all programming languages. You'll investigate logical values in R in greater detail in Section 4.1. For now, in regards to sort, you set decreasing=FALSE to sort from smallest to largest, and decreasing=TRUE sorts from largest to smallest.

Finding a Vector Length with length

I'll round off this section with the length function, which determines how many entries exist in a vector given as the argument .

```
R> length(x=c(3,2,8,1))
[1] 4
R> length(x=5:13)
[1] 9
R> foo <- 4
R> bar <- c(3,8,3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
R> length(x=bar)
[1] 11
```

Note that if you include entries that depend on the evaluation of other functions (in this case, calls to rep and seq), length tells you the number of entries *after* those inner functions have been executed.

Exercise 2.3

- a. Create a vector that stores a sequence of values from 5 to 10, progresses in steps of 0.3.
- b. Overwrite the object from (a) using the same sequence with the order reversed.
 - Repeat the vector c(-1,3,-5,7,-9) twice, with each element
- c. Repeat the vector from (b) 10 times, and store the result. Display the result from largest to smallest.
- d. Create and store a vector that contains, in any configuration, the following:
 - i. A sequence of integers from 6 to 12 (inclusive)
 - ii. A threefold repetition of the value 5.3
 - iii. The number -3
 - iv. A sequence of nine values starting at 102 and ending at the
- e. number that is the total length of the vector created in (c). Confirm that the length of the vector created in (d) is 20.

2.3.3 Subsetting and Element Extraction

In all the results you have seen printed to the console screen so far, you may have noticed a curious feature. Immediately to the left of the output there

is a square-bracketed [1]. When the output is a long vector that spans the width of the console and wraps onto the following line, another square-bracketed number appears to the left of the new line. These numbers represent the *index* of the entry directly to the right. Quite simply, the index corresponds to the *position* of a value within a vector, and that's precisely why the first value always has a [1] next to it (even if it's the only value and not part of a larger vector).

These indexes allow you to retrieve specific elements from a vector, which is known as *subsetting*. Suppose you have a vector called `myvec` in

your workspace. Then there will be exactly `length(x=myvec)` entries in `myvec`, with each entry having a specific position: 1 or 2 or 3, all the way up to `length(x=myvec)`. You can access individual elements by asking R to return

the values of `myvec` at specific locations, done by entering the name of the vector followed by the position in square brackets.

```
R> length(x=myvec)
```

```
[1] 10
```

```
R> myvec[1]
```

```
[1] 5
```

```
R> foo <- myvec[2]
```

```
R> foo
```

```
[1] -2.3
```

```
R> myvec[length(x=myvec)]
```

```
[1] -8
```

```
length(x=myvec)
```

Because results in the final index of the vector (in this

case, 10), entering this phrase in the square brackets extracts the final element, -8. Similarly, you could extract the second-to-last element by subtracting 1 from the length; let's try that, and also assign the result to a new object:

```
R> myvec.len <-
```

```
length(x=myvec) R> bar <-
```

```
myvec[myvec.len-1] <-
```

```
R> bar -
```

```
[1] 40221
```

As these examples show, the index may be an arithmetic function of other numbers or previously stored values. You can assign the result to a new object in your workspace in the usual way with the `<-` notation. Using your knowledge of sequences, you can use the colon notation with the length of

the specific vector to obtain all possible indexes for extracting a particular element in the vector:

```
R>
l:myvec.len      5678910
[1]1234
```

You can also delete individual elements by using *negative* versions of the indexes supplied in the square brackets. Continuing with the objects myvec, bar, and as defined earlier, consider the following operations:

```
myvec[-1]
[1] -2.3 4.0          4.0    4.0    6.0    8.0    10.0 40221.0    -8.0
```

This line produces the contents of without the first element. Similarly, the following code assigns to the object baz the contents of myvec without its second element:

```
R>      baz      <-
myvec[-2] R> baz
[1]544          4    6    8    10 40221    -8
```

Again, the index in the square brackets can be the result of an appropriate calculation, like so:

```
R> qux <- myvec[-(myvec.len-
1)] R> qux
[1] 5.0-2.3 4.0 4.0 4.0 6.0          8.0 10.0 -8.0
```

Using the square-bracket operator to extract or delete values from a vector does not change the original vector you are subsetting *unless* you explicitly overwrite the vector with the subsetted version. For instance, in this example, qux is a new vector defined as myvec without its second-to-last entry, but in your workspace, myvec itself *remains unchanged*. In other words, subsetting vectors in this way simply returns the requested elements, which can be assigned to a new object if you want, but doesn't alter the original object in the workspace.

Now, suppose you want to piece myvec back together from qux and bar. You can call something like this:

```
R> c(qux[-length(x=qux)],bar,qux[length(x=qux)])
[1] 5.0 -2.3 4.0 4.0 4.0 6.0      8.0    10.0 40221.0
[10] -8.0
```

As you can see, this line uses `qux[-length(x=qux)]`, the object defined earlier, and `qux[length(x=qux)]`. For to reconstruct the vector in three parts: clarity, let's examine

each part in turn.

- `qux[-length(x=qux)]`

This piece of code returns the values of `qux` except for its last element.

```
R> length(x=qux)
[1] 9
R> qux[-
length(x=qux)] [1] 5.0- 4.06.08.0 10.0
2.3 4.0 4.0
```

Now you have a vector that's the same as the first eight entries of `myvec`.

- `ec`

Earlier, you had stored `baars` the following:

```
R> bar <- myvec[myvec.len-1]
R> bar
[1] 40221
```

This is precisely the second-to-last element of `myvec`, the value that `qux` is missing. So, you'll slot this value in after `qux[-length(x=qux)]`.

- `qux[length(x=qux)]`

Finally, you just need the last element of that matches the last element of `myvec`. This is extracted from `myvec` (not `qux`, deleted as earlier) using `length(x=qux)`.

```
R> qux[length(x=qux)]
[1] -8
```

Now it should be clear how calling these three parts of code together, in this order, is one way to reconstruct `myvec`.

As with most operations in R, you are not restricted to doing things one by one. You can also subset objects using *vectors of indexes*, rather than individual indexes. Using `myvec` again from earlier, you get the following:

```
R> myvec[c(1,3,5)]
[1] 544
```

This returns the first, third, and fifth elements of `myvec` in one go.

Another common and convenient subsetting tool is the colon operator (discussed in Section 2.3.2), which creates a sequence of indexes. Here's an example:

```
R> 1:4
[1] 1234
R> foo <- myvec[1:4]
R> foo
[1] 5.0-2.3 4.0 4.0
```

This provides the first four elements of myvec (recall that the colon operator returns a numeric vector, so there is no need to explicitly wrap this using c). The order of the returned elements depends entirely upon the index vector supplied in the square brackets. For example, using foo again, consider the order of the indexes and the resulting extractions, shown here:

```
R> length(x=foo):2
[1] 4 3 2
R> foo[length(foo):2]
[1] 4.0 4.0 -2.3
```

Here you extracted elements starting at the end of the vector, working backward. You can also use to repeat an index, as shown here:

```
R> indexes <- c(4,rep(x=2,times=3),1,1,2,3:1)
R> indexes
[1] 4 2 2 2 1 1 2 3 2 1
R> foo[indexes]
[1] 4.0 -2.3 -2.3 -2.3 5.0 5.0 -2.3 4.0 -2.3 5.0
```

This is now something a little more general than strictly “subsetting”—by using an index vector, you can create an entirely new vector of any length consisting of some or all of the elements in the original vector. As shown earlier, this index vector can contain the desired element positions in any order and can repeat indexes.

You can also return the elements of a vector after deleting more than one element. For example, to create a vector after removing the first and third elements of foo, you can execute the following:

```
R> foo[-c(1,3)]
[1] -2.3 4.0
```

Note that it is not possible to mix positive and negative indexes in a single index vector.

Sometimes you’ll need to overwrite certain elements in an existing vector with new values. In this situation, you first specify the elements you want to overwrite using square brackets and then use the assignment operator to assign the new values. Here’s an example:

```
R> bar <- c(3,2,4,4,1,2,4,1,0,0,5) R>
bar
[1] 3 2 4 4 1 2 4 1 0 0 5
R> bar[1] <- 6
R> bar
[1] 6 2 4 4 1 2 4 1 0 0 5
```

This overwrites the first element of `bar`, which was originally 3, with a new value, 6. When selecting multiple elements, you can specify a single value to replace them all or enter a vector of values that's equal in length to the number of elements selected to replace them one for one. Let's try this with the same vector from earlier.

```
R> bar[c(2,4,6)] <- c(-2,-0.5,-1)
R> bar
[1] 6.0 -2.0 4.0 -0.5 1.0 -1.0 4.0 1.0 0.0 0.0 5.0
```

Here you overwrite the second, fourth, and sixth elements with -2, -0.5, and -1, respectively; all else remains the same. By contrast, the following code overwrites elements 7 to 10 (inclusive), replacing them all with :

```
R> bar[7:10] <- 100
R> bar
[1] 6.0 -2.0 4.0 -0.5 1.0 -1.0 100.0 100.0 100.0 100.0 5.0
```

Finally, it's important to mention that this section has focused on just one of the two main methods, or "flavors," of vector element extraction in R. You'll look at the alternative method, using logical flags, in Section 4.1.5.

Exercise 2.4

- a. Create and store a vector that contains the following, in this order:
 - A sequence of length 5 from 3 to 6 (inclusive)
 - A two-fold repetition of the vector `c(2,-5.1,-33)`
 - The value $7 - 42 + 2$
- b. Extract the first and last elements of your vector from (a), storing them as a new object.
- c. Store as a third object the values returned by omitting the first and last values of your vector from (a).
- d. Use only (b) and (c) to reconstruct (a).
- e. Overwrite (a) with the same values sorted from smallest to largest.
- f. Use the colon operator as an index vector to reverse the order of (e), and confirm this is identical to using `sort` on (e) with `decreasing=TRUE`.
- g. Create a vector from (c) that repeats the third element of (c) three times, the sixth element four times, and the last element once.

- h. Create a new vector as a copy of (e) by assigning (e) as is to a newly named object. Using this new copy of (e), overwrite the first, the fifth to the seventh (inclusive), and the last element with the values 99 to 95 (inclusive), respectively.

2.3.4 Vector-Oriented Behavior

Vectors are so useful because they allow R to carry out operations on multiple elements simultaneously with speed and efficiency. This *vector-oriented*, *vectorized*, or *element-wise* behavior is a key feature of the language, one that you will briefly examine here through some examples of rescaling measurements.

Let's start with this simple example:

```
R> foo <- 5.5:0.5
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
R> foo-c(2,4,6,8,10,12)
[1] 3.5 0.5 -2.5 -5.5 -8.5 -11.5
```

This code creates a sequence of six values between 5.5 and 0.5, in increments of 1. From this vector, you subtract another vector containing 2, 4, 6, 8, 10, and 12. What does this do? Well, quite simply, R matches up the elements according to their respective positions and performs the operation on each corresponding pair of elements. The resulting vector is obtained by subtracting the first element of `c(2,4,6,8,10,12)` from the first element of `foo` ($5.5 - 2 = 3.5$), then by subtracting the second element of `c(2,4,6,8,10,12)` from the second element of `foo` ($4.5 - 4 = 0.5$), and so on. Thus, rather than

inelegantly cycling through each element in turn (as you could do by hand or by explicitly using a loop), R permits a fast and efficient alternative using vector-oriented behavior. Figure 2-1 illustrates how you can understand this type of calculation and highlights the fact that the positions of the elements are crucial in terms of the final result; elements in differing positions have no effect on one another.

The situation is made more complicated when using vectors of different lengths, which can happen in two distinct ways. The first is when the length of the longer vector can be evenly divided by the length of the shorter vector. The second is when the length of the longer vector *cannot* be divided by the length of the shorter vector—this is usually unintentional on the user's part. In both of these situations, R essentially attempts to replicate, or *recycle*, the shorter vector by as many times as needed to match the length of the longer vector, before completing the specified operation. As an example, suppose you wanted to alternate the entries of `foo` shown earlier as negative

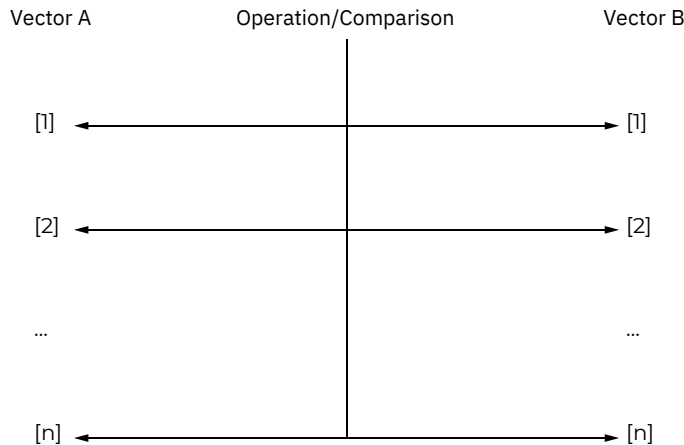


Figure 2-1: A conceptual diagram of the element-wise behavior of a comparison or operation carried out on two vectors of equal length in R. Note that the operation is performed by matching up the element positions.

and positive. You could explicitly multiply `foo` by `c(1,-1,1,-1,1,-1)`, but you don't need to write out the full latter vector. Instead, you can write the following:

```
R> bar <- c(1,-1)
R> foo*bar
[1] 5.5-4.5 3.5-2.5      1.5 -0.5
```

Here `bar` has been applied repeatedly throughout the length of `foo` until completion. The left plot of Figure 2-2 illustrates this particular example. Now let's see what happens when the vector lengths are not evenly divisible.

```
R> baz <- c(1,-1,0.5,-0.5)
R> foo*baz
[1] 5.50 -4.50 1.75 -1.25 1.50 -0.50
Warning message:
In foo * baz :
  longer object length is not a multiple of shorter object length
```

Here you see that R has matched the first four elements of `foo` with the entirety of `baz`, but it's not able to fully repeat the vector again. The repetition has been attempted, with the first two elements of `baz` being matched with the last two of the longer `foo`, though not without a protest from R, which notifies the user of the unevenly divisible lengths (you'll look at warnings in more detail in Section 12.1). The plot on the right in Figure 2-2 illustrates this example.

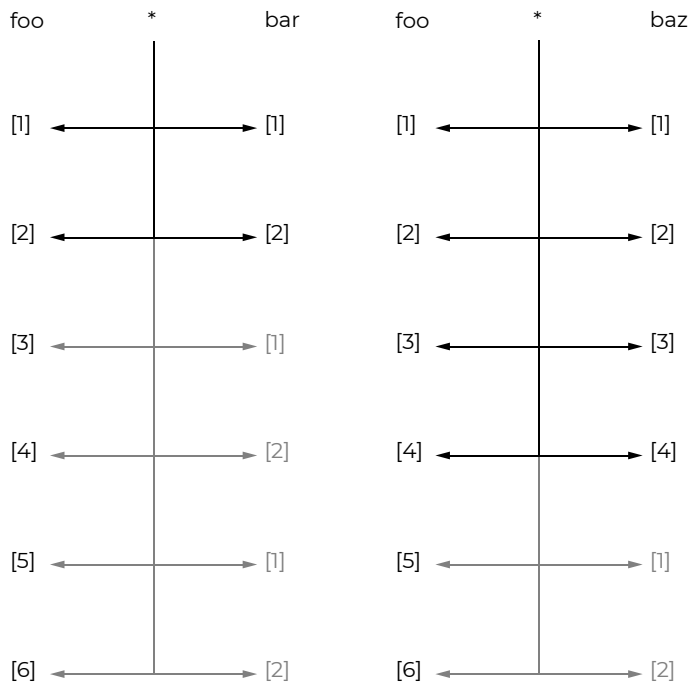


Figure 2-2: An element-wise operation on two vectors of differing lengths. Left: foo multiplied by bar; lengths are evenly divisible. Right: foo multiplied by baz; lengths are not evenly divisible, and a warning is issued.

As I noted in Section 2.3.3, you can consider single values to be vectors of length 1, so you can use a single value to repeat an operation on all the values of a vector of any length. Here's an example, using the same vector:

```
R> qux <- 3
R> foo+qux
[1] 8.5 7.5 6.5 5.5 4.5 3.5
```

This is far easier than executing `foo+c(3,3,3,3,3)` or the more general `foo+rep(x=3,times=length(x=foo))`. Operating on vectors using a single value in this fashion is quite common, such as if you want to rescale or translate a set of measurements by some constant amount. Another benefit of vector-oriented behavior is that you can use vectorized functions to complete potentially laborious tasks. For example, if you want to sum or multiply all the entries in a numeric vector, you can just use a built-in function.

Recall `foo` , shown earlier:

```
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
```

You can find the sum of these six elements with

```
R>
sum(foo)
[1] 18
```

and their product with

```
R>
prod(foo)
[1] 162.4219
```

Far from being just convenient, vectorized functions are faster and more efficient than an explicitly coded iterative approach like a loop. The main takeaway from these examples is that much of R's functionality is designed specifically for certain data structures, ensuring neatness of code as well as optimization of performance.

Lastly, as mentioned earlier, this vector-oriented behavior applies in the same way to overwriting multiple elements. Again using `foo`, examine the following:

```
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
R> foo[c(1,3,5,6)] <-
c(-99,99) R> foo
[1]-99.0 4.5 99.0 2.5-99.0          99.0
```

You see four specific elements being overwritten by a vector of length 2, which is recycled in the same fashion you're familiar with. Again, the length of the vector of replacements must evenly divide the number of elements being overwritten, or else a warning similar to the one shown earlier will be issued when R cannot complete a full-length recycle.

Exercise 2.5

- Convert the vector `c(2,0.5,1,2,0.5,1,2,0.5,1)` to a vector of only 1s, using a vector of length 3.
- The conversion from a temperature measurement in degrees Fahrenheit `F` to Celsius `C` is performed using the following equation:

$$C = \frac{5}{9}(F - 32)$$

Use vector-oriented behavior in R to convert the temperatures 45, 77, 20, 19, 101, 120, and 212 in degrees Fahrenheit to degrees Celsius.

- c. Use the vector `c(2,4,6)` and the vector `c(1,2)` in conjunction with
- d. `rep` and `*` to produce the vector `c(2,4,6,4,8,12)`.
- . Overwrite the middle four elements of the resulting vector

from

(c) with the two recycled values -0.1 and -100, in that order.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
<code>+</code> , <code>*</code> , <code>-</code> , <code>/</code> , <code>^</code>	Arithmetic	Section 2.1, p. 17
<code>sqrt</code>	Square root	Section 2.1.1, p. 18
<code>log</code>	Logarithm	Section 2.1.2, p. 19
<code>exp</code>	Exponential	Section 2.1.2, p. 19
<code><-</code> , <code>=</code>	Object assignment	Section 2.2, p. 21
<code>c</code>	Vector creation	Section 2.3.1, p. 23
<code>;</code> , <code>seq</code>	Sequence creation	Section 2.3.2, p. 24
<code>rep</code>	Value/vector repetition	Section 2.3.2, p. 25
<code>sort</code>	Vector sorting	Section 2.3.2, p. 26
<code>length</code>	Determine vector length	Section 2.3.2, p. 27
<code>[]</code>	Vector subsetting/extraction	Section 2.3.3, p. 28
<code>sum</code>	Sum all vector elements	Section 2.3.4, p. 36
<code>prod</code>	Multiply all vector elements	Section 2.3.4, p. 36

3

MATRICES AND ARRAYS



By now, you have a solid handle on using vectors in R. A *matrix* is simply several vectors stored together. Whereas the size of a vector is described by its length, the size of a matrix is specified by a number of rows and a number of columns. You can also create higher-dimensional structures that are referred to as *arrays*. In this chapter, we'll begin by looking at how to work with matrices before increasing the dimension to form arrays.

3.1 Defining a Matrix

The matrix is an important mathematical construct, and it's essential to many statistical methods. You typically describe a matrix A as an $m \times n$ matrix; that is, A will have exactly m rows and n columns. This means A will have a total of mn entries, with each entry $a_{i,j}$ having a unique position given by its specific row ($i = 1, 2, \dots, m$) and column ($j = 1, 2, \dots, n$).

You can therefore express a matrix as follows:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix}$$

To create a matrix in R, use the `matrix()` command, providing the entries of the matrix to the argument `data` as a vector:

```
R> A <- matrix(data=c(-3,2,893,0.17),nrow=2,ncol=2)
R> A
     [,1] [,2]
[1,] -3893.00
[2,] 2 0.17
```

You must make sure that the length of this vector matches exactly with the number of desired rows (`nrow`) and columns (`ncol`). You can elect not to supply `nrow` and `ncol` when calling `matrix`, in which case R's default behavior is to return a single-column matrix of the entries in `data`. For example, `matrix(data=c(-3,2,893,0.17))` would be identical to `matrix(data=c(-3,2,893,0.17),nrow=4,ncol=1)`.

3.1.1 Filling Direction

It's important to be aware of how R fills up the matrix using the entries from `data`. Looking at the previous example, you can see that the 2 × 2 matrix `A`

has been filled in a *column-by-column* fashion when reading the data

entries

from left to right. You can control how R fills in data using the argument `byrow`, as shown in the following examples.

```
R> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=FALSE)
     [,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
```

Here, `matrix` fills the matrix in a column-by-column fashion, reading the data argument vector from left to right. This is R's default handling of the `matrix` function, so if the `byrow` argument isn't supplied, the software will assume `byrow=FALSE`. Figure 3-1 illustrates this behavior.

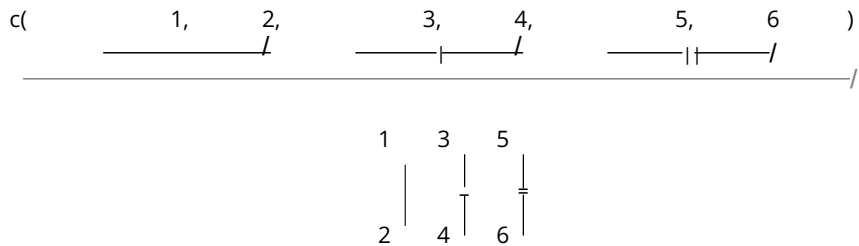


Figure 3-1: Filling a 2 × 3 matrix in a column-wise fashion with `byrow=FALSE`(R default)

Now, let's repeat the same line of code but set `byrow=TRUE`.

```
R>
matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=TRUE)
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
```

The resulting 2 × 3 structure has now been filled in a row-wise fashion, as shown in Figure 3-2.

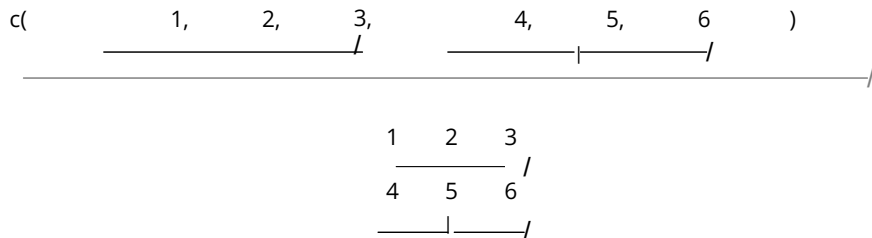


Figure 3-2: Filling a 2 × 3 matrix in a row-wise fashion with `byrow=TRUE`

3.1.2 Row and Column Bindings

If you have multiple vectors of equal length, you can quickly build a matrix by binding together these vectors using the built-in R functions, `rbind` and `cbind`. You can either treat each vector as a row (by using the command `rbind`) or treat each vector as a column (using the command `cbind`). Say you have the two vectors `1a:n3d` `4:6`. You can reconstruct the 2 × 3 matrix

```
inFigure3-2using asfollows:
R> rbind(1:3,4:6)
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
```

Here, has bound together the vectors as two rows of a matrix, with the top-to-bottom order of the rows matching the order of the vectors supplied. This same matrix could be constructed as follows, using `cbind`:

```
R> cbind(c(1,4),c(2,5),c(3,6))
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
```

Here, you have three vectors each of length 2. You use `cbind` to glue together these three vectors in the order they were supplied, and each vector becomes a column of the resulting matrix.

3.1.3 Matrix Dimensions

Another useful `dim` provides the dimensions of a matrix stored in your workspace.

```
R> mymat <-
rbind(c(1,3,4),5:3,c(100,20,90),11:13) R> mymat
[,1] [,2] [,3]
[1,] 1 3 4
[2,] 5 4 3
[3,] 100 20 90
[4,] 11 12 13

R> dim(mymat)
[1] 4 3
R> nrow(mymat)
[1] 4
R> ncol(mymat)
[1] 3
R> dim(mymat)[2]
[1] 3

mymat rbind
```

Having defined a matrix using `rbind`, you can confirm its dimensions with `dim`, which returns a vector of length 2; always supplies the number of rows first, followed by the number of columns. You can also use two related functions: `nrow` (which provides the number of rows only) and `ncol` (which provides the number of columns only). In the last command shown, you use `dim` and your knowledge of vector subsetting to extract the same result that `ncol` would give you.

3.2 Subsetting

Extracting and subsetting elements from matrices in R is much like extracting elements from vectors. The only complication is that you now have an additional dimension. Element extraction still uses the square-bracket

operator, but now it must be performed with both a row *and* a column position, given strictly in the order of `[row,column]`. Let's start by creating a 3

× 3

matrix, which I'll use for the examples in this section.

```
R> A <- matrix(c(0.3,4.5,55.3,91.0,1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
R> A [,1] [,2] [,3]
[1,] 0.3 91.0 -4.2
[2,] 4.5 0.1 8.2
[3,] 55.3 105.5 27.9
```

A

To tell R to “look at the third row of and give me the element from the second column,” you execute the following:

```
R> A[3,2]
[1] 105.5
```

As expected, you're given the element at position `[3,2]`.

3.2.1 Row, Column, and Diagonal Extractions

To extract an entire row or column from a matrix, you simply specify the desired row or column number and leave the other value blank. It's important to note that *you must still include* the comma that separates the row and column numbers—this is how R distinguishes between a request for a row and a request for a column. The following returns the second column of A:

```
R>
A[,2]      0.1 105.5
[1] 91.0
```

The following examines the first row:

```
R> A[1,]
[1] 0.3 91.0 -4.2
```

Note that whenever an extraction (or deletion, covered in a moment) results in a single value, single row, or single column, R will always return stand-alone vectors comprised of the requested values. You can also perform more complicated extractions, for example requesting whole rows or columns, or multiples rows or columns, where the result must be returned as a new matrix of the appropriate dimensions. Consider the following subsets:

```
R> A[2:3,]
      [,1] [,2] [,3]
[1,]  4.5  0.1  8.2
[2,] 55.3 105.5 27.9
```

```
R> A[,c(3,1)]
      [,1] [,2]
[1,]  8.2  0.3
```

```
[1,] -4.2 0.3 [2,]
8.2 4.5 [3,] 27.9
55.3
```

```
R> A[c(3,1),2:3]
      [,1] [,2]
[1,] 105.5 27.9
[2,] 91.0-4.2
```

The first command returns the second and third rows of A, and the second command returns the third and first columns of A. The last command accesses the third and first rows of A, in that order, and from those rows it returns the second and third column elements.

You can also identify the values along the diagonal of a square matrix (that is, a matrix with an equal number of rows and columns) using the `diag` command.

```
R> diag(x=A)
[1] 0.3 0.127.9
```

This returns a vector with the elements along the diagonal of A, at `starting`.

3.2.2 Omitting and Overwriting

To delete or omit elements from a matrix, you again use square brackets, but this time with negative indexes. The following provides A without its second column:

```
R> A[,-2]
      [,1] [,2]
[1,] 0.3-4.2
[2,] 4.5 8.2 [3,]
55.3 27.9
```

The following removes the first row from A and retrieves the third and second column values, in that order, from the remaining two rows:

```
R> A[-1,3:2]
      [,1] [,2]
[1,] 8.2 0.1 [2,]
27.9 105.5
```

The following produces A without its first row and second column:

```
R> A[-1,-2]
      [,1] [,2]
[1,] 4.5 8.2
[2,] 55.3 27.9
```

Lastly, this deletes the first row and then deletes the second and third columns from the result:

```
R> A[-1,-c(2,3)]
[1] 4.555.3
```

Note that this final operation leaves you with only the last two elements of the first column of A, so this result is returned as a stand-alone vector rather than a matrix.

To overwrite particular elements, or entire rows or columns, you identify the elements to be replaced and then assign the new values, as you did with vectors in Section 2.3.3. The new elements can be a single value, a vector of the same length as the number of elements to be replaced, or a vector whose length evenly divides the number of elements to be replaced. To illustrate this, let's first create a copy of A and call it B.

```
R> B<-A
R> B
      [,1] [,2] [,3]
[1,] 0.3 91.0-4.2 [2,]
4.5 0.1 8.2 [3,] 55.3
105.5 27.9
```

The following overwrites the second row of B with the sequence 1,2, and:

```
R> B[2,] <- 1:3
R> B
      [,1] [,2] [,3]
[1,] 0.3 91.0-4.2
[2,] 1.0 2.0 3.0
[3,] 55.3 105.5 27.9
```

The following overwrites the second column elements of the first and third rows with :

```
R> B[c(1,3),2] <- 900
R> B
      [,1] [,2] [,3]
[1,] 0.3 900-4.2
[2,] 1.0 2 3.0
[3,] 55.3 900 27.9
```

Next, you replace the third column of B with the values in the third row of.

```
R> B[,3] <- B[3,]
R> B
      [,1] [,2] [,3]
[1,] 0.3 900-4.2
[2,] 1.0 2 3.0
[3,] 55.3 900 27.9
```

```
[1,] 0.3 900 55.3
[2,] 1.0 2900.0
[3,] 55.3 900 27.9
```

To try R's vector recycling, let's now overwrite the first and third column elements of rows 1 and 3 (a total of four elements) with the two values -7 and.

```
R> B[c(1,3),c(1,3)] <- c(-7,7)
R> B
[1,] [2,] [3,]
[1,] -7 900 -7
[2,] 1 2 900
[3,] 7 900 7
```

The vector of length 2 has replaced the four elements *in a column-wise fashion*. The replacement vector `c(-7,7)` overwrites the elements at positions (1,1) and (3,1), in that order, and is then repeated to overwrite (1,3) and (3,3), in that order.

To highlight the role of index order on matrix element replacement, consider the following example:

```
R> B[c(1,3),2:1] <- c(65,-65,88,-88)
R> B [1,] [2,] [3,]
[1,] 88 65 -7
[2,] 1 2 900
[3,] -88 -65 7
```

The four values in the replacement vector have overwritten the four specified elements, again in a column-wise fashion. In this case, because I specified the first and second columns in reverse order, the overwriting proceeded accordingly, filling the second column before moving to the first. Position (1,2) is matched with 65, followed by (3,2) with -65; then (1,1) becomes 88, and (3,1) becomes -88.

If you just want to replace the diagonal of a square matrix, you can avoid explicit indexes and directly overwrite the values using the `diag` command.

```
R> diag(x=B) <- rep(x=0,times=3)
R> B
[1,] [2,] [3,]
[1,] 0 65 -7
[2,] 1 0 900
[3,] -88 -65 0
```

Exercise 3.1

- a. Construct and store a 4×2 matrix that's filled row-wise with the values 4.3, 3.1, 8.2, 8.2, 3.2, 0.9, 1.6, and 6.5, in that order.
- b. Confirm the dimensions of the matrix from (a) are 3×2 if you remove any one row.
- c. Overwrite the second column of the matrix from (a) with that same column sorted from smallest to largest.
- d. What does R return if you delete the fourth row and the first column from (c)? Use matrix to ensure the result is a single-column matrix, rather than a vector.
- e. Store the bottom four elements of (c) as a new 2×2 matrix.
- f. Overwrite, in this order, the elements of (c) at positions (4,2), (1,2), (4,1), and (1,1) with -12 of the two values on the diagonal of (e).

3.3 Matrix Operations and Algebra

You can think of matrices in R from two perspectives. First, you can use these structures purely as a computational tool in programming to store and operate on results, as you've seen so far. Alternatively, you can use matrices for their mathematical properties in relevant calculations, such as the use of matrix multiplication for expressing regression model equations. This distinction is important because the mathematical behavior of matrices is not always the same as the more generic data handling behavior. Here I'll briefly describe some special matrices, as well as some of the most common mathematical operations involving matrices, and the corresponding functionality in R. If the mathematical behavior of matrices isn't of interest to you, you can skip this section for now and refer to it later as needed.

3.3.1 Matrix Transpose

For any matrix A , its *transpose*, A^T , is the $n \times m$ matrix obtained by writing either its columns as rows or its rows as columns. Here's an example:

```
[ ] 2 6 2 5 2 I f A A
=>= 51 614, then
```

In R, the transpose of a matrix is found with the function `t`. Let's create a new matrix and then transpose it.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> A [,1] [,2] [,3]
[1,] 2 5 2
[2,] 6 1 4
R> t(A)
[,1] [,2]
[1,] 2 6
[2,] 5 1
[3,] 2 4
```

If you “transpose the transpose” of A , you’ll recover the original matrix.

```
R> t(t(A))
[,1] [,2] [,3]
[1,] 2 5 2 [2,] 6 1 4
```

3.3.2 Identity Matrix

The *identity matrix* written as I_m is a particular kind of matrix used in mathematics. It’s a square $m \times m$ matrix with ones on the diagonal and zeros elsewhere.

Here’s an example:

```
1 0 0
I = 0 1 0 3
```

001 You can create an identity matrix of any dimension using the standard matrix function, but there’s a quicker approach using `diag`. Earlier, I used `diag` on an existing matrix to extract or overwrite its diagonal elements. You can also use it as follows:

```
R> A <- diag(x=3)
R> A
[,1] [,2] [,3]
[1,] 1 0 0
[2,] 0 1 0
[3,] 0 0 1
```

`diag` Here you see can be used to easily produce an identity matrix. To clarify, the behavior of `diag` depends on what you supply to it as its argument. If, as earlier, is a matrix, `diag` will retrieve the diagonal elements of the matrix. If is a single positive integer, as is the case here, then `diag` will produce the identity matrix of the corresponding dimension. You can more uses of `diag` on

3.3.3 Scalar Multiple of a Matrix

A scalar value is just a single, univariate value. Multiplication of any matrix A by a scalar value a results in a matrix in which every individual element is multiplied by a.

Here's an example:

```
[[1]25241042×614=1228
```

R will perform this multiplication in an element-wise manner, as you might expect. Scalar multiplication of a matrix is carried out using the standard arithmetic operator.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> a<-2
R> a*A
[,1] [,2] [,3]
[1,] 4 10 4
[2,] 12 2 8
```

3.3.4 Matrix Addition and Subtraction

Addition or subtraction of two matrices of equal size is also performed in an element-wise fashion. Corresponding elements are added or subtracted from one another, depending on the operation.

Here's an example:

```
26 28.1 42151
```

```
. - -
3 8 . 2 = 2 7 . 2 - 4 -
```

698 2-. -413.8 You can add or subtract any two equally sized matrices with the stan-

d and symbols.

```
R> A <- cbind(c(2,5,2),c(6,1,4))
R> A
[,1] [,2]
[1,] 2 6
[2,] 5 1
[3,] 2 4
R> B <- cbind(c(-2,3,6),c(8.1,8.2,-9.8))
R> B
[,1] [,2]
[1,] -2 8.1
[2,] 3 8.2
[3,] 6 -9.8
R> A-B
```

```

      [,1] [,2]
[1,] 4-2.1 [2,]
2- 7.2 [3,]
-413.8

```

3.3.5 Matrix Multiplication

× In n o r d n e d r p t o × m q, u i l t t i m p l u y s t t w o m a t r i c e s A a n d B o f s i z e m
 b e t r u e t h a t n = p. T h e r e s u l t i n g m a t r i x A
 · B w i l l h a v e t h e s i z e m × q. T h e
 e l e m e n t s o f t h e p r o d u c t a r e c o m p u t e d i n a r o w - b y - c o l u m n f a s h i o n, w h e r e
 t h e v a l u e a t p o s i t i o n (A B)_{i, j} i s c o m p u t e d b y e l e m e n t - w i s e m u l t i p l i c a t i o n o f
 t h e e n t r i e s i n r o w i o f A b y t h e e n t r i e s i n c o l u m n j o f B, s u m m i n g t h e r e s u l t.
 H e r e ' s a n e x a m p l e:

```

[] 252614
3 - 3 . -
1 [2×3+5631
1 × - + - + +
[= x x x x
15 ](1)21×2+(33)952(11)32×5- + x x - + x x
( 1) 41 6( 3=) 1(1) 45

```

Note that, in general, multiplication of appropriately sized matrices (denoted, say, with C and D) is not commutative; that is, CD ≠ DC. Unlike addition, subtraction, and scalar multiplication, matrix multiplication is not a simple element-wise calculation, and the standard * operator cannot be used. Instead, you must use R's matrix product operator, written with percent symbols as %*%. Before you try this operator, let's first store the two example matrices and check to make sure the number of columns in the first matrix matches the number of rows in the second matrix using dim.

```

R> A <- rbind(c(2,5,2),c(6,1,4))
R> dim(A)
[1] 2 3
R> B <- cbind(c(3,-1,1),c(-3,1,5)) R>
dim(B)
[1] 3 2

```

This confirms the two matrices are compatible for multiplication, so you can proceed.

```

R> A%*%B
      [,1] [,2]
[1,] 3 9
[2,] 21 3

```

You can show that matrix multiplication is noncommutative using the same two matrices. Switching the order of multiplication gives you an entirely different result.

```
R> B%%A
      [,1] [,2] [,3]
[1,] -12 12 -6
[2,]  4 -4  2
[3,] 32 10 22
```

3.3.6 Matrix Inversion

Some square matrices can be *inverted*. The inverse of a A is denoted matrix A^{-1} . An invertible matrix satisfies the following equation:

$$AA^{-1} = I$$

Here's an example of a matrix and its inverse:

```
[,] 3 1 1-0.542=
-2 1.5
```

Matrices that are not invertible are referred to as *singular*. Inverting a matrix is often necessary when solving equations with matrices and has important practical ramifications. There are several different approaches to matrix inversion, and these calculations can become extremely computationally expensive as you increase the size of a matrix. We won't go into too much detail here, but if you're interested, see Golub and Van Loan (1989) for formal discussions.

For now, I'll just show you the R function `solve` as one option for inverting a matrix.

```
R> A <- matrix(data=c(3,4,1,2),nrow=2,ncol=2)
R> A
      [,1] [,2]
[1,]  3  1
[2,]  4  2
R> solve(A)
      [,1] [,2]
[1,] 1-0.5
[2,] -2 1.5
```

You can also verify that the product of these two matrices (using matrix multiplication rules) results in the 2×2 identity matrix.

```
R> A%%solve(A)
      [,1] [,2]
[1,] 1 0
[2,] 0 1
```

Exercise 3.2

a. Calculate the following:

$$\begin{matrix} * 2.12 & 1020 \\ 24+, & -3040 / 7 \end{matrix}$$

b. $\begin{matrix} 76 & 5060 \end{matrix}$ -Store these two matrices:

$$\begin{matrix} 1 \\ 2B= & A & 3= \\ & & 4 \end{matrix}$$

7 8 Which of the following multiplications are possible? For those that are, compute the result.

- i. A
 - ii. B
 - iii. $A \cdot B$
 - iv. $(A \cdot A) \cdot B$
 - v. $(A \cdot A) \cdot B$
 - vi. $(B \cdot B) + (A \cdot A) - 100I - 13$
- For
- $$\begin{matrix} 2 & 0 & 0 & 0 & x \\ 0 & 3 & 0 & 0 \\ A & 0 & 0 & 0 & 1 \end{matrix}$$

3.4 Multidimensional Arrays

Just as a column vector (a "triangle" of elements) is the result of increasing dimensionality (as a "triangle" of elements), a matrix can be increased to get more complex data structures. In R, vectors and matrices can be considered special cases of the more general *array*, which is how I'll refer to these types of structures when they have more than two dimensions. So, what's the next step up from a matrix? Well, just as a matrix is considered to be a collection of vectors of equal length, a three-dimensional array can be considered to be a collection of equally dimensioned matrices,

providing you with a rectangular prism of elements. You still have a fixed number of rows and a fixed number of columns, as well as a new third dimension called a *layer*. Figure 3-3 illustrates a three-row, four-column, two-layer(3 4 × 2)×array.

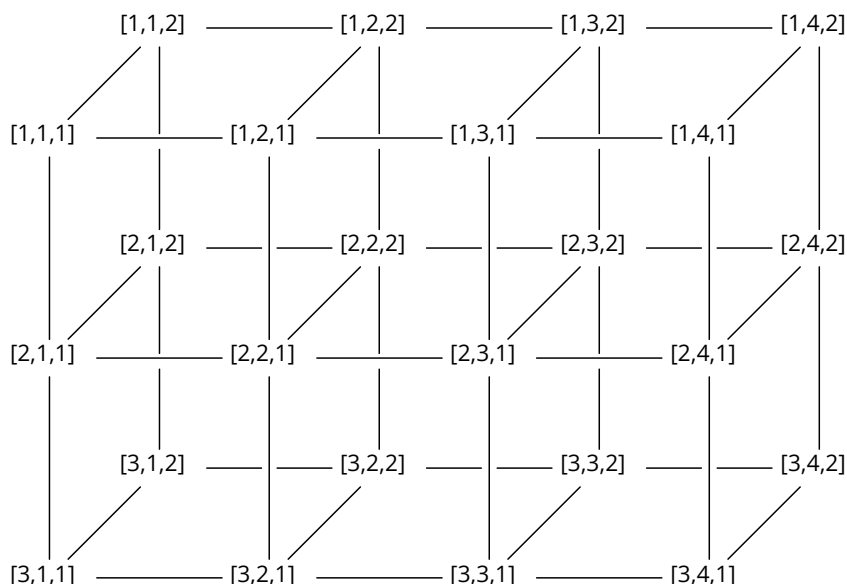


Figure 3-3: A conceptual diagram of a 3 × 4 × 2 array. The index of each element is given at the corresponding position. These indexes are provided in the strict order of [row,column,layer].

3.4.1 Definition

To create these data structures in R, use the array function and specify the individual elements in the data argument as a vector. Then specify size in the dim argument as another vector with a length corresponding to the number of dimensions. Note that array fills the entries of each layer with the elements in data in a strict column-wise fashion, starting with the first layer. Consider the following example:

```
R> AR <- array(data=1:24,dim=c(3,4,2))
```

```
R>AR
```

```
, , 1
```

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

```
[1,] 1 4 7 10
```

```
[2,] 2 5 8 11
```

```
[3,] 3 6 9 12
```

```
„2
```

```

      [,1] [,2] [,3] [,4]
[1,] 13 16 19 22 [2,] 14
17 20 23 [3,] 15 18 21 24

```

This gives you an array of the same size as in Figure 3-3—each of the two layers constitutes a 3×4 matrix. In this example, note the order of the dimensions supplied to `dim`: `c(rows,columns,layers)`. Just like a single matrix, the product of the dimension sizes of an array will yield the total number of elements. As you increase the dimension further, the `dim` vector must be extended accordingly. For example, a four-dimensional array is the next step up and can be thought of as *blocks* of three-dimensional arrays. Suppose you had a four-dimensional array comprised of three copies of AR, the three-dimensional array just defined. This new array can be stored in R as follows (once again, the array is filled column-wise):

```

R> BR <- array(data=rep(1:24,times=3),dim=c(3,4,2,3))
R>BR
„1,1

[,1] [,2] [,3] [,4]
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12

„2,1

[,1] [,2] [,3] [,4]
[1,] 13 16 19 22
[2,] 14 17 20 23
[3,] 15 18 21 24

„1,2

[,1] [,2] [,3] [,4]
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12

„2,2

[,1] [,2] [,3] [,4]
[1,] 13 16 19 22
[2,] 14 17 20 23
[3,] 15 18 21 24

„1,3

```

```

      [,1] [,2] [,3] [,4]
[1,]  1  4  7 10 [2,]  2  5  8
11 [3,]  3  6  9 12

```

```

,,2,3

```

```

      [,1] [,2] [,3] [,4]
[1,] 13 16 19 22
[2,] 14 17 20 23
[3,] 15 18 21 24

```

```

BR

```

With you now have three copies of AR. Each of these copies is split into its two layers so R can print the object to the screen. As before, the rows are indexed by the first digit, the columns by the second digit, and the layers by the third digit. The new fourth digit indexes the blocks.

3.4.2 Subsets, Extractions, and Replacements

Even though high-dimensional objects can be difficult to conceptualize, R indexes them consistently. This makes extracting elements from these structures straightforward now that you know how to subset matrices—you just have to keep using commas in the square brackets as separators of the dimensions being accessed. This is highlighted in the examples that follow.

Suppose you want the second row of the second layer of the previously created array AR. You just enter these exact dimensional locations of AR in square brackets.

```

R> AR[2,,2]
[1] 14 17 20 23

```

The desired elements have been extracted as a vector of length 4. If you want specific elements from this vector, say the third and first, in that order, you can call the following:

```

R> AR[2,c(3,1),2]
[1] 20 14

```

Again, this literal method of subsetting makes dealing with even high-dimensional objects in R manageable.

An extraction that results in multiple vectors will be presented as columns in the returned matrix. For example, to extract the first rows of both layers of ,yAoR enter this:

```

R>
AR[1,,]
[,1] [,2]
[1,] 1 13

```

```
[2, 4 1
] 7 6
[3, 10 1
] 9
```

The returned object has the first rows of each of the two matrix layers. However, it has returned each of these vectors as a *column* of the single returned matrix. As this example shows, when multiple vectors are extracted from an array, they will be returned as columns by default. This means extracted rows will not necessarily be returned as rows. Turning to the object BR, the following gives you the single element of the second row and first column of the matrix in the first layer of the three- dimensional array located in the third block.

```
R> BR[2,1,1,3]
[1]2
```

Again, you just need to look at the position of the index in the square brackets to know which values you are asking R to return from the array. The following examples highlight this:

```
R> BR[1,,,1]
      [,1] [,2]
[1,] 1 13 [2,]
4 16 [3,] 7 19
[4,] 10 22
```

This returns all the values in the first row of the first block. Since I left the column and layer indexes blank in this subset [1,,,1], the command has returned values for all four columns and both layers in that block of BR.

BR Next, the following line returns all the values in the second layer of the array ,composedofthreematrices:

```
R> BR[, ,2]
, , 1
      [,1] [,2] [,3] [,4]
[1,] 13 16 19 22
[2,] 14 17 20 23
[3,] 15 18 21 24
, , 2
      [,1] [,2] [,3] [,4]
[1,] 13 16 19 22
[2,] 14 17 20 23
[3,] 15 18 21 24
```

```
, , 3
```

```
      [,1] [,2] [,3] [,4]  
[1,] 13 16 19 22 [2,] 14  
17 20 23 [3,] 15 18 21 24
```

This last example highlights a feature noted earlier, where multiple vectors from AR were returned as a matrix. Broadly speaking, if you have an extraction that results in multiple d-dimensional arrays, the result will be an array of the next-highest dimension, $d + 1$. In the last example, you extracted multiple (two-dimensional) matrices, and they were returned as a three-dimensional array. This is demonstrated again in the next example:

```
R> BR[3:2,4,,]  
,,1
```

```
      [,1] [,2]  
[1,] 12 24 [2,]  
11 23
```

```
,,2
```

```
      [,1] [,2]  
[1,] 12 24 [2,]  
11 23
```

```
,,3
```

```
      [,1] [,2]  
[1,] 12 24 [2,]  
11 23
```

This extracts the elements at rows 3 and 2 (in that order), column 4, for all layers and for all array blocks. Consider the following final example:

```
R> BR[2,,1,]  
      [,1] [,2] [,3]  
[1,] 2 2 2 [2,] 5 5  
5 [3,] 8 8 8 [4,] 11  
11 11
```

Here you've asked R to return the entire second rows of the first layers of all the arrays stored in BR.

Deleting and overwriting elements in high-dimensional arrays follows the same rules as for stand-alone vectors and matrices. You specify the

dimension positions the same way, using negative indexes (for deletion) or using the assignment operator for overwriting. You can use the array function to create one-dimensional arrays (vectors) and two-dimensional arrays (matrices) should you want to (by setting the dim argument to be of length 1 or 2, respectively). Note, though, that vectors in particular may be treated differently by some functions if created with array instead of c (see the help file ?array for technical details). For this reason, and to make large sections of code more readable, it's more conventional in R programming to use the specific vector- and matrix-creation functions c and matrix.

Exercise 3.3

- a. Create and store a three-dimensional array with six layers of a 4×2 matrix, filled with a decreasing sequence of values between 4.8 and 0.1 of the appropriate length.
- b. Extract and store as a new object the fourth- and first-row elements, in that order, of the second column only of all layers of (a).
- c. Use a fourfold repetition of the second row of the matrix formed
- d $\times \text{in2} (\times \text{b2}) \times \text{to3}$.fill a new array of dimensions 2
Create a new array comprised of the results of deleting the sixth layer of (a).
- e. Overwrite the second and fourth row elements of the second column of layers 1, 3, and 5 of (d) with

Important Code in This Chapter

Function/operator	Brief description	First occurrence
matrix	Create a matrix	Section 3.1, p. 40
rbind	Create a matrix (bind rows)	Section 3.1.2, p. 41
cbind	Create a matrix (bind columns)	Section 3.1.2, p. 42
dim	Get matrix dimensions	Section 3.1.3, p. 42
nrow	Get number of rows	Section 3.1.3, p. 42
ncol	Get number of columns	Section 3.1.3, p. 42
[,]	Matrix/array subsetting	Section 3.2, p. 43
diag	Diagonal elements/identity	Section 3.2.1, p. 44
t	matrix Matrix transpose	Section 3.3.1, p. 47
*	Scalar matrix multiple	Section 3.3.3, p. 49
+, -	Matrix addition/subtraction	Section 3.3.4, p. 49
%*%	Matrix multiplication	Section 3.3.5, p. 50
%*%	Matrix inversion	Section 3.3.6, p. 51
solve	Create an array	Section 3.4.1, p. 53
array		

4

NON-NUMERIC VALUES



So far, you've been working almost exclusively with numeric values. But statistical programming also requires non-numeric values. In this chapter, we'll consider three important non-numeric data types: logicals, characters, and factors. These data types play an important role in effective use of R, especially as we get into more complex R programming in Part II.

4.1 Logical Values

Logical values (also simply called *logicals*) are based on a simple premise: a logical-valued object can only be either TRUE or FALSE. These can be interpreted as yes/no, one/zero, satisfied/not satisfied, and so on. This is a concept that appears across all programming languages, and logical values have many important uses. Often, they signal whether a condition has been satisfied or whether a parameter should be switched on or off.

You encountered logical values briefly when you used the sort function in Section 2.3.2 and the matrix function in Section 3.1. When using sort, setting `decreasing=TRUE` returns a vector ordered from largest to smallest, and

decreasing=FALSE sorts the vector the other way around. Similarly, when constructing a matrix, byrow=TRUE fills the matrix entries row-wise; otherwise, the matrix is filled column-wise. Now, you'll take a more detailed look at ways to use logicals.

4.1.1 TRUE or FALSE?

Logical values in R are written fully as TRUE and FALSE, but they are frequently abbreviated as T or F. The abbreviated version has no effect on the execution of the code, so, for example, using decreasing=T is equivalent to decreasing=TRUE in the sort function. (But do not create objects named T or F if you want to make use of this convenience—see Section 9.1.3.)

Assigning logical values to an object is the same as assigning numeric values.

```
R> foo <- TRUE
R> foo
[1] TRUE
R> bar <- F
R> bar
[1] FALSE
```

This gives you one object with the value TRUE and one with the value FALSE. Similarly, vectors can be filled with logical values.

```
R> baz <- c(T,F,F,F,T,F,T,T,F,T,F)
R> baz
[1] TRUE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE FALSE
R> length(x=baz)
[1] 12
```

Matrices (and other higher-dimensional arrays) can be created with these values too. Using foo and baz from earlier, you could construct something like this:

```
R> qux <-
matrix(data=baz,nrow=3,ncol=4,byrow=foo) R> qux
[,1] [,2] [,3] [,4]
[1,] TRUE FALSE FALSE FALSE
[2,] TRUE FALSE TRUE TRUE
[3,] TRUE FALSE TRUE FALSE
```

4.1.2 A Logical Outcome: Relational Operators

Logicals are commonly used to check relationships between values. For example, you might want to know whether some number a is greater than a predefined threshold b. For this, you use the standard *relational operators* shown in Table 4-1, which produce logical values as results.

Table 4-1: Relational Operators

Operator	Interpretation
=	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Typically, these operators are used on numeric values (though you'll look at some other possibilities in Section 4.2.1). Here's an example:

```
R> 1==2
[1] FALSE
R> 1>2
[1] FALSE
R> (2-1)<=2
[1] TRUE
R> 1!=(2+3)
[1] TRUE
```

The results should be unsurprising: 1 being equal to 2 is FALSE and 1 being greater than 2 is also FALSE, while the result of 2-1 being less than or equal to 2 is TRUE and it is also TRUE that 1 is not equal to 5 (2+3). These kinds of operations are much more useful when used on numbers that are variable in some way, as you'll see shortly.

You're already familiar with R's element-wise behavior when working with vectors. The same rules apply when using relational operators. To illustrate this, let's first create two vectors and double-check that they're of equal length.

```
R> foo <- c(3,2,1,4,1,2,1,-1,0,3)
R> bar <- c(4,1,2,1,1,0,0,3,0,4)
R> length(x=foo)==length(x=bar)
[1] TRUE
```

Now consider the following four evaluations:

```
R> foo==bar
[1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE
R> foo<bar
[1] TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
R> foo<=bar
[1] TRUE FALSE TRUE FALSE TRUE FALSE FALSE TRUE TRUE TRUE
R> foo<=(bar+10)
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

The first line checks whether the entries in `foo` are equal to the corresponding entries in `bar`, which is true only for the fifth and ninth entries. The returned vector will contain a logical result for each pair of elements, so it will be the same length as the vectors being compared. The second line compares `foo` and `bar` in the same way, this time checking whether the entries in `foo` are less than the entries in `bar`. Contrast this result with the third comparison, which asks whether entries are less than *or equal to* one another. Finally, the fourth line checks whether `foo`'s members are less than or equal to `bar`, when the elements of `bar` are increased by 10. Naturally, the results are all TRUE.

Vector recycling also applies to logicals. Let's use `foo` from earlier, along with a shorter vector, `baz`.

```
R>      baz      <-
foo[c(10,3)] R> baz
[1] 3 1
```

Here you create a vector of length 2 comprised of the 10th and 3rd elements of `foo`. Now consider the following:

```
R> foo>baz
[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE FALSE FALSE TRUE
      baz
```

Here, the two elements of `baz` are recycled and checked against the 10 elements of `foo`. Elements 1 and 2 of `foo` are checked against 1 and 2 of `baz`, elements 3 and 4 of `foo` are checked against 1 and 2 of `baz`, and so on. You can also check all the values of a vector against a single value. Here's an example:

```
R> foo<3
[1] FALSE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE FALSE
```

This is a typical operation when handling data sets in R. Now let's rewrite the contents of `foo` and `bar` as 5

× 2 column-filled matrices.

```
R> foo.mat <- matrix(foo,nrow=5,ncol=2)
R> foo.mat
[,1] [,2]
[1,] 3 2
[2,] 2 1
[3,] 1 -1
[4,] 4 0
[5,] 1 3
R> bar.mat <- matrix(bar,nrow=5,ncol=2)
R> bar.mat
[,1] [,2]
[1,] 4 0
```

```
[2,    1    0
]      2    3
[3,    1    0
]      1    4
[4,
```

] The same element-wise behavior applies here; if you compare the matrices, you get a matrix of the same size filled with logicals.

```
R> foo.mat<=bar.mat
[,1] [,2]
[1,] TRUE FALSE
[2,] FALSE FALSE
[3,] TRUE TRUE
[4,] FALSE TRUE
[5,] TRUE TRUE
R> foo.mat<3
[,1] [,2]
[1,] FALSE TRUE
[2,] TRUE TRUE
[3,] TRUE TRUE
[4,] FALSE TRUE
[5,] TRUE FALSE
```

This kind of evaluation also applies to arrays of more than two dimensions.

There are two useful functions you can use to quickly inspect a collection of logical values: `any` and `all`. When examining a vector, `any` returns `TRUE` if any of the logicals in the vector are `TRUE` and returns `FALSE` otherwise. The function `all` returns a `TRUE` only if *all* of the logicals are `TRUE`, and returns `FALSE` otherwise. As a quick example, let's work with two of the logical vectors formed by the comparisons of `foo` and `bar` from the beginning of this section.

```
R> qux <- foo==bar
R> qux
[1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE
R> any(qux)
[1] TRUE
R> all(qux)
[1] FALSE
```

Here, the `any` function returns `TRUE` because `qux` contains two `TRUE`s, and the rest are `FALSE`—so the result of `any` is of course `TRUE`, but the result of `all` is `FALSE`. Following the same rules, you get this:

```
R> quux <- foo<=(bar+10)
R> quux
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
R> any(quux)
[1] TRUE
```

```
[1] TRUE
R> all(quux)
[1] TRUE
```

The `any` and `all` functions do the same thing for matrices and arrays of logical values.

Exercise 4.1

- a. Store the following vector of 15 values as an object in your workspace: `c(6,9,7,3,6,7,9,6,3,6,6,7,1,9,1)`. Identify the following elements:
 - i. Those equal to 6
 - ii. Those greater than or equal to 6
 - iii. Those less than 6 + 2
 - iv. Those not equal to 6
- b. Create a new vector from the one used in (a) by deleting its first two elements. With this new vector, fill a 2 × 3 array. Examine the array for the following entries:
 - i. Those less than or equal to 6 divided by 2, plus 4
 - ii. Those less than or equal to 6 divided by 2, plus 4, *after* increasing every element in the array by 2
- c. Confirm the specific locations of elements equal to 0 in the 10 × 10 identity matrix `I10` (see Section 3.3).
- d. Check whether *any* of the values of the logical arrays created in (b) are TRUE. If they are, check whether they are *all* TRUE. By extracting the diagonal elements of the logical matrix created in (c), use `any` to confirm there are no TRUE entries.

4.1.3 Multiple Comparisons: Logical Operators

Logicals are especially useful when you want to examine whether multiple conditions are satisfied. Often you'll want to perform certain operations only if a number of different conditions have been met.

The previous section looked at relational operators, used to compare the literal values (that is, numeric or otherwise) of stored R objects. Now you'll look at *logical operators*, which are used to compare two TRUE or FALSE objects. These operators are based on the statements AND and OR. Table 4-2 summarizes the R syntax and the behavior of logical operators. The AND and OR operators each have a “single” and “element-wise” version—you'll see how they're different in a moment.

Table 4-2: Logical Operators Comparing Two Logical Values

Operator	Interpretation	Results
&	AND (element-wise)	TisTRUE & TRUE TRUE
		TisTRUE & FALSE FALSE
		FisTRUE & TRUE FALSE
		FisTRUE & FALSE FALSE
&&	AND (single comparison)	&Sameas above
		TisTRUE TRUE TRUE
	OR (element-wise)	TisTRUE FALSE TRUE
		FisTRUE TRUE TRUE
	OR (single comparison)	FisTRUE FALSE FALSE
		above
	NOT	Same
		TRUE FALSE
!		FALSE TRUE
		as is

The result of using any logical operator is a logical value. An AND comparison is true only if *both* logicals are TRUE. An OR comparison is true if at least one of the logicals is TRUE. The NOT operator (!) simply returns the opposite of the logical value it's used on. You can combine these operators to examine multiple conditions at once.

```
R> FALSE ||
((T&&TRUE) || FALSE) [1] TRUE
R> !TRUE&&TRUE
[1] FALSE
R> (T&&(TRUE | F))&&FALSE
[1] FALSE
R> (6<4) || (3!=1)
[1] TRUE
```

As with numeric arithmetic, there is an order of importance for logical operations in R. An AND statement has a higher precedence than an OR statement. It's helpful to place each comparative pair in parentheses to preserve the correct order of evaluation and make the code more readable. You can see this in the first line of this code, where the innermost comparison is the first to be carried out: T&&TRUE results in TRUE; this is then provided as one of the logical values for the next bracketed comparison where TRUE||FALSE results in TRUE. The final comparison is then FALSE||TRUE, and the result, TRUE, is printed to the console. The second line reads as NOT TRUE AND TRUE, which of course returns FALSE. In the third line, once again the innermost pair is evaluated first: TRUE||F is TRUE; T&&TRUE is TRUE; and finally TRUE&&FALSE

is FALSE. The fourth and final example evaluates two distinct conditions in parentheses, which are then compared using a logical operator. Since $6 < 4$ is FALSE and $3 \neq 1$ is TRUE, that gives you a logical comparison of FALSE||TRUE and a final result of TRUE.

In Table 4-2, there is a short (&, |) and long (&&, ||) version of the AND and OR operators. The short versions are meant for element-wise comparisons, where you have two logical vectors and you want multiple logicals as a result. The long versions, which you've been using so far, are meant for comparing two individual values and will return a single logical value. This is important when programming conditional checks in R in an if statement, which you'll look at in Chapter 10. It's possible to compare a single pair of logicals using the short version—though it's considered better practice to use the long versions when a single / result is needed.

Let's look at some examples of element-wise comparisons. Suppose you have two vectors of equal length, and :

```
R> foo <- c(T,F,F,F,T,F,T,T,T,F,T,F)
R> foo
[1] TRUE FALSE FALSE FALSE TRUE FALSE TRUE TRUE TRUE FALSE TRUE FALSE
```

and

```
R> bar <- c(F,T,F,T,F,F,F,F,T,T,T,T)
R> bar
[1] FALSE TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
```

The short versions of the logical operators match each pair of elements by position and return the result of the comparison.

```
R> foo&bar
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE
R> foo|bar
[1] TRUE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Using the long version of the operators, on the other hand, means R carries out the comparison only on the first pair of logicals in the two vectors.

```
R>
foo&&bar
[1] FALSE
R>
foo||bar
[1] TRUE
```

Notice that the last two results match the first entries of the vectors you got using the short versions of the logical operators.

Exercise 4.2

- a. Store the vector `c(7,1,7,10,5,9,10,3,10,8)` as `foo`. Identify the elements greater than 5 OR equal to 2.
- b. Store the vector `c(8,8,4,4,5,1,5,6,6,8)` as `bar`. Identify the elements less than or equal to 6 AND not equal to 4.
 - Identify the elements that satisfy (a) in `foo` AND satisfy (b) in `bar`.
- c. Store a third vector called `baz` that is equal to the element-wise sum of `foo` and `bar`. Determine the following:
 - d. i. The elements of `baz` greater than or equal to 14 but not equal to 15
 - ii. The elements of the vector obtained via an element-wise division of `baz` by `foo` that are greater than 4 OR less than or equal to 2
- e. Confirm that using the long version in all of the preceding exercises performs only the first comparison (that is, the results each match the first entries of the previously obtained vectors).

4.1.4 Logicals Are Numbers!

Because of the binary nature of logical values, they're often represented with TRUE as 1 and FALSE as 0. In fact, in R, if you perform elementary numeric operations on logical values, TRUE is treated like 1, and FALSE is treated like 0.

```
R> TRUE+TRUE
[1] 2
R> FALSE-TRUE
[1] -1
R>
T+T+F+T+F+F+T
[1] 4
```

These operations turn out the same as if you had used the digits 1 and 0. In some situations when you'd use logicals, you can substitute the numeric values.

```
R> 1&&1
[1] TRUE
R> 1||0
[1] TRUE
R> 0&&1
[1] FALSE
```

Being able to interpret logicals as zeros and ones means you can use a variety of functions to summarize a logical vector, and you'll explore this further in Part III.

4.1.5 Logical Subsetting and Extraction

Logicals can also be used to extract and subset elements in vectors and other objects, in the same way as you've done so far with index vectors. Rather

than entering explicit indexes in the square brackets, you can supply logical *flag* vectors, where an element is extracted if the corresponding entry in the flag vector is TRUE. As such, logical flag vectors should be the same length as the vector that's being accessed (though recycling does occur for shorter flag vectors, as a later example shows).

At the beginning of Section 2.3.3 you defined a vector of length 10 as follows:

```
R> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
```

If you wanted to extract the two negative elements, you could either enter `myvec[2,10]` or you could do the following using logical flags:

```
R> myvec[c(F,T,F,F,F,F,F,F,T)]  
[1] -2.3 -8.0
```

This particular example may seem far too cumbersome for practical use. It becomes useful, however, when you want to extract elements based on whether they satisfy a certain condition (or several conditions). For example, you can easily use logicals to find negative elements in `myvec` by applying the condition `<0`.

```
R> myvec<0  
[1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
```

This is a perfectly valid flag vector that you can use to subset `myvec` to get the same result as earlier.

```
R>  
myvec[myvec<0]  
[1] -2.3 -8.0
```

As mentioned, R recycles the flag vector if it's too short. To extract every second element from `myvec`, starting with the first, you could enter the following:

```
R> myvec[c(T,F)]  
[1] 5 4          4      8 40221
```

You can do more complicated extractions using relational and logical operators, such as:

```
R> myvec[(myvec>0)&
(myvec<1000)] [1]54446810
```

This returns the positive elements that are less than 1,000. You can also overwrite specific elements using a logical flag vector, just as with index vectors.

```
R> myvec[myvec<0] <-
-200 R> myvec
[1] 5 -200 4 4 4 6 8 10 40221 -200
```

This replaces all existing negative entries with -200. Note, though, that you cannot directly use negative logical flag vectors to delete specific elements; this can be done only with numeric index vectors.

As you can see, logicals are therefore very useful for element extraction. You don't need to know beforehand which specific index positions to return, since the conditional check can find them for you. This is particularly valuable when you're dealing with large data sets and you want to inspect records or recode entries that match certain criteria.

In some cases, you might want to convert a logical flag vector into a numeric index vector. This is helpful when you need the explicit indexes of elements that were flagged TRUE. The R function which takes in a logical vector as the argument `axnd` and returns the indexes corresponding to the positions of any and all entries.

```
R> which(x=c(T,F,F,T,T))
[1]145
```

You can use this to identify the index positions of `myvec` that meet a certain condition; for example, those containing negative numbers:

```
R> which(x=myvec<0)
[1] 210
```

The same can be done for the other `myvec` selections you experimented with. Note that a line of code such as `myvec[which(x=myvec<0)]` is redundant because that extraction can be made using the condition by itself, that is, via `myvec[myvec<0]`, without using `which`. On the other hand, using `which` lets you delete elements based on logical flag vectors. You can simply use `which` to identify the numeric indexes you want to delete and render them negative. To omit the negative entries of `myvec`, you could execute the following:

```
R> myvec[-
which(x=myvec<0)] [1]5444 6 8 10 40221
```

The same can be done with matrices and other arrays. In Section 3.2, you stored a 3 × 3 matrix as follows:

```
R> A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
R> A [,1] [,2] [,3]
[1,] 0.3 91.0 -4.2
[2,] 4.5 0.1 8.2
[3,] 55.3 105.5 27.9
```

To extract the second and third column elements of the first row of A using numeric indexes, you could execute A[1,2:3]. To do this with logical flags, you could enter the following:

```
R> A[c(T,F,F),c(F,T,T)] [1]
91.0 -4.2
```

Again, though, you usually wouldn't explicitly specify the logical vectors. Suppose for example you want to replace all elements in A that are less than 1 with

–7. Performing this using numeric indexes is rather fiddly. It's much easier to use the logical flag matrix created with the following:

```
R> A<1
      [,1] [,2] [,3]
[1,] TRUE FALSE TRUE
[2,] FALSE TRUE FALSE
[3,] FALSE FALSE FALSE
```

You can supply this logical matrix to the square bracket operators, and the replacement is done as follows:

```
R> A[A<1] <- -7
R> A
      [,1] [,2] [,3]
[1,] -7.0 91.0 -7.0
[2,] 4.5 -7.0 8.2 [3,]
55.3 105.5 27.9
```

This is the first time you've subsetting a matrix without having to list row or column positions inside the square brackets, using commas to separate out dimensions (see Section 3.2). This is because the flag matrix has the same number of rows and columns as the target matrix, thereby providing all the relevant structural information.

If you use which to identify numeric indexes based on a logical flag structure, you have to be a little more careful when dealing with two-dimensional objects or higher. Suppose you want the index positions of the elements that are greater than 25. The appropriate logical matrix is as follows.

```
R> A>25
      [,1] [,2] [,3]
[1,] FALSE TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] TRUE TRUE TRUE
```

Now, say you ask R the following:

```
R>
which(x=A>25)
[1]3469
```

This returns the four indexes of the elements that satisfied the relational check, but they are provided as scalar values. How do these correspond to the row/column positioning of the matrix?

The answer lies in R's default behavior for the which function, which essentially treats the multidimensional object as a single vector (laid out column after column) and then returns the vector of corresponding indexes. Say the matrix A was arranged as a vector by stacking the columns first through third, using `c(A[,1],A[,2],A[,3])`. Then the indexes returned make more sense.

```
R> which(x=c(A[,1],A[,2],A[,3])>25)
[1]3469
```

With the columns laid out end to end, the elements that return TRUE are the third, fourth, sixth, and ninth elements in the list. This can be difficult to interpret, though, especially when dealing with higher-dimensional arrays. In this kind of situation, you can make which return dimension-specific indexes using the optional argument `arr.ind` (array indexes). By default, this argument is set to FALSE, resulting in the vector converted indexes. Setting `arr.ind` to TRUE, on the other hand, treats the object as a matrix or array rather than a vector, providing you with the row and column positions of the elements you requested.

```
R>
which(x=A>25,arr.ind=T)
row col
[1,] 3 1
[2,] 1 2
[3,] 3 2
[4,] 3 3
```

The returned object is now a matrix, where each row represents an element that satisfied the logical comparison and each column provides the position of the element. Comparing the output here with A, you can see these positions do indeed correspond to elements where `A>25`.

Both versions of the output (with `arr.ind=T` or `arr.ind=F`) can be useful—the correct choice depends on the application.

Exercise 4.3

- a. Store this vector of 10 values: `foo <- c(7,5,6,1,2,10,8,3,8,2)`. Then, do the following:
 - i. Extract the elements greater than or equal to 5, storing the result as `bar`.
 - ii. Display the vector containing those elements from `foo` that remain after omitting all elements that are greater than or equal to 5.
- b. Use `bar` from (a)(i) to construct a 2×3 matrix called `baz`, filled in a row-wise fashion. Then, do the following:
 - i. Replace any elements that are equal to 8 with the squared value of the element in row 1, column 2 of `baz` itself.
 - ii. Confirm that all values in `baz` are now less than or equal to 25 AND greater than 4.
- c. Create a $3 \times 2 \times 3$ array called `qux` using the following vector of 18 values: `c(10,5,1,4,7,4,3,3,1,3,4,3,1,7,8,3,7,3)`. Then, do the following:
 - i. Identify the dimension-specific index positions of elements that are either 3 OR 4.
 - ii. Replace all elements in `qux` that are less than 3 OR greater than or equal to 7 with the value 100.
- d. Return to `foo` from (a). Use the vector `c(F,T)` to extract every second value from `foo`. In Section 4.1.4, you saw that in some situations, you can substitute 0 and 1 for TRUE and FALSE. Can you perform the same extraction from `foo` using the vector `c(0,1)`? Why or why not? What does R return in this case?

4.2 Characters

Character strings are another common data type, and are used to represent text. In R, strings are often used to specify folder locations or software options (as shown briefly in Section 1.2); to supply an argument to a function; and to annotate stored objects, provide textual output, or help clarify plots and graphics. In a simple way, they can also be used to define different groups making up a categorical variable, though as you'll see in Section 4.3, are better suited for that.

NOTE *There are three different string formats in the R environment. The default string format is called an extended regular expression; the other variants are named Perl and literal regular expressions. The intricacies of these variants are beyond the scope of this book, so any mention of character strings from here on refers to an extended regular expression. For more technical details about other string formats, enter `?regex` at the prompt.*

4.2.1 Creating a String

Character strings are indicated by double quotation marks, ". To create a string, just enter text between a pair of quotes.

```
R> foo <- "This is a character string!" R>
foo
[1] "This is a character string!"
R> length(x=foo)
[1] 1
```

R treats the string as a single entity. In other words, foo is a vector of length 1 because R counts only the total number of distinct strings rather than individual words or characters. To count the number of individual characters, you can use the nchar function. Here's an example using foo:

```
R> nchar(x=foo)
[1] 27
```

Almost any combination of characters, including numbers, can be a valid character string.

```
R> bar <- "23.3"
R> bar
[1] "23.3"
```

Note that in this form, the string has no numeric meaning, and it won't be treated like the number 23.3. Attempting to multiply it by 2, for example, results in an error.

```
R> bar*2
Error in bar * 2 : non-numeric argument to binary operator
```

* This error occurs because R is expecting to operate on two numeric values (not one number and one string, which makes no sense).

Strings can be compared in several ways, the most common comparison being a check for equality.

```
R> "alpha"=="alpha"
[1] TRUE
R> "alpha"!="beta"
[1] TRUE
R> c("alpha","beta","gamma")=="beta"
[1] FALSE TRUE FALSE
```

Other relational operators work as you might expect. For example, R considers letters that come later in the alphabet to be greater than earlier

letters, meaning it can determine whether one string of letters is greater than another with respect to alphabetical order.

```
R> "alpha"<="beta"
[1] TRUE
R>
"gamma">"Alpha"
[1] TRUE
```

Furthermore, uppercase letters are considered greater than lowercase letters.

```
R> "Alpha">"alpha"
[1] TRUE
R> "beta">="bEtA"
[1] FALSE
```

Most symbols can also be used in a string. The following string is valid, for example:

```
R> baz <- "&_ 3 **%.? $symbolic non$en$e ,;"
R> baz
[1] "&_ 3 **%.? $symbolic non$en$e ,;"
```

One important exception is the backslash`\`, also called an *escape*. When a backslash is used within the quotation marks of a string, it initiates some simple control over the printing or display of the string itself. You'll see how this works in a moment in Section 4.2.3. First let's look at two useful functions for combining strings.

4.2.2 Concatenation

There are two main functions used to *concatenate* (or glue together) one or more strings: `cat` and `paste`. The difference between the two lies in how their contents are returned. The first function, `cat`, sends its output directly to the console screen and doesn't formally *return* anything. The `paste` function concatenates its contents and then returns the final character string as a usable

R object. This is useful when the result of a string concatenation needs to be passed to another function or used in some secondary way, as opposed to just being displayed. Consider the following vector of character strings:

```
R> qux <- c("awesome","R","is")
R> length(x=qux)
[1]3
R> qux
[1]"awesome""R" "is"
```

As with numbers and logicals, you can also store any number of strings in a matrix or array structure if you want.

When calling `cat` or `paste`, you pass arguments to the function in the order you want them combined. The following lines show identical usage yet different types of output from the two functions:

```
R> cat(qux[2],qux[3],"totally",qux[1],"!")
R is totally awesome !
R> paste(qux[2],qux[3],"totally",qux[1],"!") [1]
"R is totally awesome !"
```

Here, you've used the three elements of `qux` as well as two additional strings, "totally" and "!", to produce the final concatenated string. In the output, note that `cat` has simply concatenated and printed the text to the screen. This means you cannot directly assign the result to a new variable and treat it as a character string. For `paste`, however, the `[1]` to the left of the output and the presence of the " quotes indicate the returned item is a vector containing a character string, and this can be assigned to an object and used in other functions.

NOTE *There's a slight difference between OS X and Windows in the default handling of string concatenation when using the R GUI. After calling `cat` in Windows, the new R prompt awaiting your next command appears on the same line as the printed string, in which case you can just hit ENTER to move to the next line, or use an escape sequence, which you'll look at in Section 4.2.3. In OS X, the new prompt appears on the next line as usual.*

These two functions have an optional argument, `sep`, that's used as a separator between strings as they're concatenated. You pass `sep` a character string, and it will place this string between all other strings you've provided to or .Forexample:

```
R> paste(qux[2],qux[3],"totally",qux[1],"!",sep="---")
[1] "R---is---totally---awesome---!"
R> paste(qux[2],qux[3],"totally",qux[1],"!",sep="")
[1] "Ristotallyawesome!"
cat
```

The same behavior would occur for `.` Note that if you don't want any separation, you set `sep=""`, an empty string, as shown in the second example.

The empty string separator can be used to achieve correct sentence spacing; note the gap between awesome and the exclamation mark in the previous code when you first used `paste` and `cat`. If the `sep` argument isn't included, R will insert a space between strings by default.

For example, using manual insertion of spaces where necessary, you can write the following:

```
R> cat("Do you think ",qux[2]," ",qux[3]," ",qux[1],"?",sep="") Do
you think R is awesome?
```

Concatenation can be useful when you want to neatly summarize the results from a certain function or set of calculations. Many kinds of R objects

can be passed directly to paste or cat; the software will attempt to automatically *coerce* these items into character strings. This means R will convert the input into a string so the values can be included in the final concatenated string. This works particularly well with numeric objects, as the following examples demonstrate:

```
R>a<-3
R> b <- 4.4
R> cat("The value stored as 'a' is ",a,".",sep="")
The value stored as 'a' is 3.
R> paste("The value stored as 'b' is ",b,".",sep="")
[1] "The value stored as 'b' is 4.4."
R> cat("The result of a+b is ",a,"+",b,"=",a+b,".",sep="")
The result of a+b is 3+4.4=7.4.
R> paste("Is ",a+b," less than 10? That's totally ",a+b<10,".",sep="") [1]
"Is 7.4 less than 10? That's totally TRUE."
```

Here, the values of the non-string objects are placed where you want them in the final string output. The results of calculations can also appear as fields, as shown with the arithmetic `a+b` and the logical comparison `a+b<10`. You'll see more details about coercion from one kind of value to another in Section 6.2.4.

4.2.3 Escape Sequences

In Section 4.2.1, I noted that a stand-alone backslash doesn't act like a normal character within a string. The `\` is used to invoke an *escape sequence*. An escape sequence lets you enter characters that control the format and spacing of the string, rather than being interpreted as normal text. Table 4-3 describes some of the most common escape sequences, and you can find a full list by entering `?Quotes` at the prompt.

Table 4-3: Common Escape Sequences for Use in Character Strings

Escape sequence	Result
<code>\n</code>	Starts a newline
<code>\t</code>	Horizontal tab
<code>\b</code>	Invokes a backspace
<code>\\</code>	Used as a single backslash
<code>\"</code>	Includes a double quote

Escape sequences add flexibility to the display of character strings, which can be useful for summaries of results and plot annotations. You enter the sequence precisely where you want it to take effect. Let's look at an example.

```
R> cat("here is a string\nsplit\ntto neww\b\n\n\tlines") here
is a string
split tonew

lines
```

\ Since the signal for an escape is and the signal to begin and end a string is , if you want either of these characters to be included in a string, you must also use an escape to have them be interpreted as a normal character.

```
R> cat("I really want a backslash: \\nand a double quote: \") I really
want a backslash: \
and a double quote: "
```

These escape sequences mean that you can't use a stand-alone backslash in file path strings in R. As noted in Section 1.2.3 (where you used `getwd` to print the current working directory and `setwd` to change it), folder separation must use a forward slash and not a backslash.

```
R> setwd("/folder1/folder2/folder3/")
```

File path specification crops up when reading and writing files, which you'll explore in Chapter 8.

4.2.4 Substrings and Matching

Pattern matching lets you inspect a given string to identify smaller strings within it.

The function `substr` takes a string and extracts the part of the string between two character positions (inclusive), indicated with numbers `start` and `stop` as arguments. Let's try it on the object `foo` from Section 4.2.1.

```
R> foo <- "This is a character string!"
R> substr(x=foo,start=21,stop=27)
[1] "string!"
```

Here, you've extracted the characters between positions 21 and 27, inclusive, to get "string!". The function `substr` can also be used with the assignment operator to directly substitute in a new set of characters. In this case, the replacement string should contain the same number of characters as the selected area.

```
R> substr(x=foo,start=1,stop=4) <- "Here"
R> foo
[1] "Here is a character string!"
```

If the replacement string is longer than the number of characters indicated by start and stop, then replacement still takes place, beginning at start and ending at stop. It cuts off any characters that overrun the number of characters you're replacing. If the string is shorter than the number of characters you're replacing, then replacement ends when the string is fully inserted, leaving the original characters up to ^{stop} untouched.

Substitution is more flexible using the functions `sub` and `gsub`. The `sub` function searches a given string for a smaller string pattern contained within. It then replaces the first instance with a new string, given as the argument. The function `gsub` does the same thing, but it replaces every instance of . Here's an example:

```
R> bar <- "How much wood could a woodchuck chuck"
R> sub(pattern="chuck",replacement="hurl",x=bar)
[1] "How much wood could a woodhurl chuck"
R> gsub(pattern="chuck",replacement="hurl",x=bar)
[1] "How much wood could a woodhurl hurl"
```

```
sub gsub replacement
pattern
```

With `and`, the values need not have the same number of characters as the being replaced. These functions also have search options like case-sensitivity. The help files `?sub` and `?gsub` have more details, as well as noting a handful of other pattern-matching functions and techniques. You might also want to check out the command `grep` and its variants; see the relevant help file `?grep`.

Exercise 4.4

- a. Re-create exactly the following output:

```
"The quick brown fox
jumped over
the lazy dogs"
```

- b. Suppose you've stored the values `num1 <- 4` and `num2 <- 0.75`. Write a line of R code that returns the following string:
- ```
[1] "The result of multiplying 4 by 0.75 is 3"
```

Make sure your code produces a string with the correct multiplication result for *any* two numbers stored as `num1` and `num2`.

- c. On my local machine, the directory for my work on this book is specified in R as `"/Users/tdavies/Documents/RBook/"`. Imagine it is your machine—write a line of code that replaces *tdavies* in this string with your first initial and surname.

- d. In Section 4.2.4, you stored the following string:
- ```
R> bar <- "How much wood could a woodchuck chuck"
```
- b. Store a new string by gluing onto the words "if a woodchuck could chuck wood"
- ii. In the result of (i), replace all instances of `wood` with `metal`.
- e. "STtwooe 6th-peasctkrsin fgo r. T\$h1e2n.9d9o" the following:
- i. Use `checkequal` to confirm that the substring beginning with character 5 and ending with character 10 is "6-pack".
- ii. Make it a better deal by changing the price to \$10.99.

4.3 Factors

In this section, you'll look at some simple functions related to creating, handling, and inspecting *factors*. Factors are R's most natural way of representing data points that fit in only one of a finite number of distinct categories, rather than belonging to a continuum. Categorical data like this can play an important role in data science, and you'll look at factors again in more detail from a statistical perspective in Chapter 13.

4.3.1 Identifying Categories

To see how factors work, let's start with a simple data set. Suppose you find eight people and record their first name, sex, and month of birth in Table 4-4.

Table 4-4: An Example Data Set of Eight Individuals

Person	Sex	Month of birth
Liz	Female	April
Jolene	Female	January
Susan	Female	December
Boris	Male	September
Rochelle	Female	November
Tim	Male	July
Simon	Male	July
Amy	Female	June

There's really only one sensible way to represent the name of each person in R—as a vector of character strings.

```
R>                               firstname                               <-  
c("Liz", "Jolene", "Susan", "Boris", "Rochelle", "Tim", "Simon", "Amy")
```

You have more flexibility when it comes to recording sex, however. Coding females as 0 and males as 1, a numeric option would be as follows:

```
R> sex.num <- c(0,0,0,1,0,1,1,0)
```

Of course, character strings are also possible, and many prefer this because you don't need to remember the numeric code for each group.

```
R>                               sex.char                               <-  
c("female", "female", "female", "male", "female", "male", "male", "female")
```

There is, however, a fundamental difference between an individual's name and their sex when stored as data. Where a person's name is a unique identifier that can take any one of an infinite number of possibilities, there are generally only two options for recording a person's sex. These kinds of data, where all possible values fall into a finite number of categories, are best represented in R using factors.

Factors are typically created from a numeric or a character vector (note that you cannot fill matrices or multidimensional arrays using factor values; factors can only take the form of vectors). To create a factor vector, use the function `factor`, as in this example working with `sex.num` and `sex.char`:

```
R> sex.num.fac <- factor(x=sex.num)  
R> sex.num.fac  
[1] 00010110  
Levels: 0 1  
R> sex.char.fac <- factor(x=sex.char)  
R> sex.char.fac  
[1] female female female male female male      male   female  
Levels: female male
```

Here, you obtain factor versions of the two vectors storing gender values.

At first glance, these objects don't look much different from the character and numeric vectors from which they were created. Indeed, factor objects work in much the same way as vectors, but with a little extra information attached (R's internal representation of factor objects is a little different as well). Functions like `length` and `which` work the same way on factor objects as with vectors, for example.

The most important extra piece of information (or *attribute*; see Section 6.2.1) that a factor object contains is its *levels*, which store the possible values in the factor. These levels are printed at the bottom of each factor

vector. You can extract the levels as a vector of character strings using the `levels` function.

```
R> levels(x=sex.num.fac)
[1] "0" "1"
R> levels(x=sex.char.fac)
[1] "female" "male"
```

You can also relabel a factor using `levels`. Here's an example:

```
R> levels(x=sex.num.fac) <- c("1","2") R>
sex.num.fac
[1]11121221
Levels: 1 2
```

- 1 This relabels the females and the males 2.
Factor-valued vectors are subsetted in the same way as any other vector.

```
R> sex.char.fac[2:5]
[1] female female male female
Levels: female male
R> sex.char.fac[c(1:3,5,8)]
[1] female female female female
Levels: female male
```

Note that after subsetting a factor object, the object continues to store *all* defined levels even if some of the levels are no longer represented in the subsetted object.

If you want to subset from a factor using a logical flag vector, keep in mind that the levels of a factor are stored as character strings, even if the original data vector was numeric, so you need to use a string when request- ing or testing for a particular level. To, for example, identify all the men using the newly relabeled `sex.num.fac`, use this:

```
R> sex.num.fac=="2"
[1] FALSE FALSE FALSE TRUE FALSE TRUE TRUE FALSE
```

`firstname` `sex` Since the elements in `and` have corresponding positions in their factor vectors, you can then use this logical vector to obtain the names of all the men (this time using the `/ facto` "rmvealcet"o "rf)e.male"

```
R> firstname[sex.char.fac=="male"]
[1] "Boris" "Tim" "Simon"
```

Of course, this simple subsetting could have been achieved in much the same way with the raw numeric vector `sex.num` or the raw character vector `sex.char`. In the next section, you'll explore some more distinctive advantages to having categorical data represented as a factor in R.

4.3.2 Defining and Ordering Levels

The sex factor from the previous section represents the simplest kind of factor variable—there are only two possible levels with no ordering, in that one level is not intuitively considered “higher than” or “following” the other. Here you’ll look at factors with levels that can be logically ordered; for example, month of birth (MOB), where there are 12 levels that have a natural order. Let’s store the observed MOB data from earlier as a character vector.

```
R> mob <- c("Apr","Jan","Dec","Sep","Nov","Jul","Jul","Jun")
```

There are two problems with the data in this vector. First, not all possible categories are represented since mob contains only seven unique months. Second, this vector doesn’t reflect the natural order of the months. If you compare January and December to see which is greater, you get:

```
R> mob[2]
[1] "Jan"
R> mob[3]
[1] "Dec"
R>      mob[2]
<mob[3]      [1]
FALSE
```

Alphabetically, this result is of course correct—*J* doesn’t occur before *D*. But in terms of the order of the calendar months, which is what we’re interested in, the FALSE result is incorrect.

If you create a factor object from these values, you can deal with both of these problems by supplying additional arguments to the function. You can define additional levels by supplying a character vector of all possible values to the argument and then instruct R to order the values precisely as they appear in by setting the argument to

```
R> ms <- c("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov",
"Dec")
R> mob.fac <- factor(x=mob,levels=ms,ordered=TRUE)
R> mob.fac
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
```

```
mob.fac
```

```
mob
```

Here the vector contains the same individual entries at the same index positions as the vector from earlier. But notice that this variable has 12 levels, even though you have not made any observations for the levels , , , or . (Note that if your R console window is too narrow to print all the levels to the screen, you may see a , indicating there’s more output that’s been hidden. Just widen your window and reprint the object to see the hidden levels.) Also, the strict order of these levels is

shown by the < symbol in the object output. Using this new factor object, you can perform the relational comparison from earlier and get the result you might expect.

```
R> mob.fac[2]
[1] Jan
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec R>
mob.fac[3]
[1] Dec
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec R>
mob.fac[2]<mob.fac[3]
[1] TRUE
```

These improvements are far from just cosmetic. There's a big difference, for example, between a data set with zero observations in some of the categories and the same data set defined with fewer categories to begin with. The choice of whether to instruct R to formally order a factor vector can also have important consequences in the implementation of various statistical methods, such as regression and other types of modeling.

4.3.3 Combining and Cutting

As you've seen, it's usually simple to combine multiple vectors of the same kind (whether numeric, logical, or character) using the `c` function. Here's an example:

```
R> foo <- c(5.1,3.3,3.1,4) R>
bar <- c(4.5,1.2)
R> c(foo,bar)
[1] 5.1 3.3 3.1 4.0 4.5 1.2
```

This combines the two numeric vectors into one. However, the `c` function doesn't work the same way with factor-valued vectors. Let's see what happens when you use it on the data in Table 4-4 and the MOB factor vector `mob.fac`, from Section 4.3.2. Suppose you now observe three more individuals with MOB values "Oct", "Feb", and "Feb", which are stored as a factor object, as follows.

```
R> new.values <- factor(x=c("Oct","Feb","Feb"),levels=levels(mob.fac),
ordered=TRUE)
R> new.values
[1] Oct Feb Feb
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
```

Now you have with the original eight observations and `mob.fac` new values with an additional three. Both are factor objects, defined with identical, ordered levels. You might expect that you can just use to combine the two as follows.

```
R> c(mob.fac,new.values)
[1]411291177 610 2 2
```

Clearly, this has not done what you want it to do. Combining the two factor objects resulted in a numeric vector. This is because the `c` function interprets factors as integers. Comparing this with the defined levels, you can see that the numbers refer to the index of each month within the ordered levels.

```
R> levels(mob.fac)
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

`levels(mob.fac)` This means you can use these integers with to retrieve a character vector of the complete observed data—the original eight observations plus the additional three.

```
R> levels(mob.fac)[c(mob.fac,new.values)]
[1] "Apr" "Jan" "Dec" "Sep" "Nov" "Jul" "Jul" "Jun" "Oct" "Feb" "Feb"
```

Now you have all the observations stored in a vector, but they are currently stored as strings, not factor values. The final step is to turn this vector into a factor object.

```
R> mob.new <- levels(mob.fac)[c(mob.fac,new.values)]
R> mob.new.fac <- factor(x=mob.new,levels=levels(mob.fac),ordered=TRUE)
R> mob.new.fac
[1] Apr Jan Dec Sep Nov Jul Jul Jun Oct Feb Feb
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
```

As this example shows, combining factors requires you to essentially deconstruct the two objects, obtaining the numeric index positions of each entry with respect to the factor levels, and then rebuild them together. This helps ensure that the levels are consistent and the observations are valid in the final product.

Factors are also often created from data that was originally measured on a continuum, for example the weight of a set of adults or the amount of a drug given to a patient. Sometimes you'll need to group (or *bin*) these types of observations into categories, like Small/Medium/Large or Low/High. In R, you can mold this kind of data into discrete factor categories using the `cut` function. Consider the following numeric vector of length 10:

```
R> Y <- c(0.53,5.4,1.5,3.33,0.45,0.01,2.4,2.1,99,1.01)
```

Suppose you want to bin the data as follows: *Small* refers to observations in the interval $[0,2)$, *Medium* refers to $[2,4)$, and *Large* refers

to [4,6]. A square bracket refers to *inclusion* of its nearest value, and a parenthesis indicates *exclusion*, so an observation y will fall in the Small interval if 0

$y < 2$, in Medium if $2 \leq y < 4$, or in Large if $4 \leq y \leq 6$. For this you'd use cut and supply your desired break intervals to the breaks argument:

```
R> br <- c(0,2,4,6)
R> cut(x=Y,breaks=br)
      [1] (0,2] (4,6] (0,2] (2,4] (0,2] (0,2] (0,2] (4,6] (0,2] (0,2]
Levels: (0,2] (2,4] (4,6]
```

This gives you a factor, with each observation now assigned an interval. However, notice that your boundary intervals are back-to-front—you want the boundary levels on the left like (0,2], rather than the right as they appear by default, (0,2]. You can fix this by setting the logical argument right FALSE

```
R> cut(x=Y,breaks=br,right=F)
      [1] (0,2] (4,6] (0,2] (2,4] (0,2] (0,2] (2,4] (4,6] (0,2] (0,2]
Levels: (0,2] (2,4] (4,6]
```

Now you've swapped which boundaries are inclusive and exclusive. This is important because it changes which categories the values fall into. Notice that the seventh observation has changed categories. But there's still a problem: the final interval currently *excludes* 6, and you want this maximum value to be *included* in the highest level. You can fix this with another logical argument: Eventhoughit'scalled"include.lowest",thisargument canalsobeusedtoinclude thehighestvalue if is,asindicatedin thehelpfile .

```
R> cut(x=Y,breaks=br,right=F,include.lowest=T)
      [1] (0,2] (4,6] (0,2] (2,4] (0,2] (0,2] (2,4] (4,6] (0,2] (0,2]
Levels: (0,2] (2,4] (4,6]
```

The intervals are now defined how you want. Finally, you want to add better labels to the categories, rather than using the interval levels that R applies by default, by passing a character string vector to the labels argument. The order of labels must match the order of the levels in the factor object.

```
R> lab <- c("Small","Medium","Large")
R> cut(x=Y,breaks=br,right=F,include.lowest=T,labels=lab)
      [1] Small Large Small Medium Small Small Medium Large  Small  Small
Levels: Small Medium Large
```

Exercise 4.5

The New Zealand government consists of the political parties National, Labour, Greens, and Māori, with several smaller parties labeled as Other. Suppose you asked 20 New Zealanders which of these they identified most with and obtained the following data:

- There were 12 males and 8 females; the individuals numbered 1, 5–7, 12, and 14–16 were females.
- The individuals numbered 1, 4, 12, 15, 16, and 19 identified with Labour; no one identified with Māori; the individuals numbered 6, 9, and 11 identified with Greens; 10 and 20 identified with Other; and the rest identified with National.

a. Use your knowledge of vectors (for example, subsetting and overwriting) to create two character vectors: sex with entries "M" (male) and "F" (female) and party with entries "National", "Labour", "Greens", "Maori", and "Other". Make sure the entries are placed in the correct positions as outlined earlier.

b. Create two different factor vectors based on sex and party. Does it make any sense to use `ordered=TRUE` in either case? How has R appeared to arrange the levels?

c. Use factors subsetting to do the following:

- Return the factor vector of chosen parties for only the male participants.
- Return the factor vector of genders for those who chose National.

d. Another six people joined the survey, with the results

`c("National", "Maori", "Maori", "Labour", "Greens", "Labour")` for the preferred party and `c("M", "M", "F", "F", "F", "M")` as their gender. Combine these results with the original factors from (b).

Suppose you also asked all individuals to state how confident they were that Labour will win more seats in Parliament than National in the next election and to attach a subjective percentage to that confidence. The following 26 results were obtained: 93, 55, 29, 100, 52, 84, 56, 0, 33, 52, 35, 53, 55, 46, 40, 40, 56, 45, 64, 31, 10, 29, 40, 95, 18, 61.

e. Create a factor with levels of confidence as follows: Low for percentages [0,30]; Moderate for percentages (30,70]; and High for percentages (70,100].

f. From (e), extract the levels corresponding to those individuals who originally said they identified with Labour. Do this also for National. What do you notice?

Important Code in This Chapter

Function/operator	Brief description	First occurrence
TRUE FALSE	Reserved logical values	Section 4.1.1, p. 60
T,F	Unreserved versions of above	Section 4.1.1, p. 60
==, !=, >, <, >=, <=	relational operators	Section 4.1.2, p. 61
any	Checks whether any entries are TRUE	Section 4.1.2, p. 63
all	Checks whether all entries are TRUE	Section 4.1.2, p. 63
&&, & , , !	logical operators	Section 4.1.2, p. 63
which	Determines indexes of TRUEs	Section 4.1.3, p. 65
" "	Creates a character string	Section 4.1.5, p. 69
nchar	Gets number of characters in a string	Section 4.2.1, p. 73
cat	Concatenates strings (no return)	Section 4.2.1, p. 73
paste	Pastes strings (returns a string)	Section 4.2.2, p. 74
\	String escape	Section 4.2.2, p. 74
substr	Subsets a string	Section 4.2.2, p. 74
sub, gsub	String matching and replacement	Section 4.2.3, p. 76
factor	Creates a factor vector	Section 4.2.4, p. 77
levels	Gets levels of a factor	Section 4.2.4, p. 78
cut	Creates factor from continuum	Section 4.3.1, p. 80
		Section 4.3.1, p. 81
		Section 4.3.3, p. 85

5

LISTS AND DATA FRAMES



Vectors, matrices, and arrays are efficient and convenient data storage structures in R, but they have one distinct limitation: they can store only one type of data. In this chapter, you'll explore two more data structures, lists and data frames, which can store multiple types of values at once.

5.1 Lists of Objects

The *list* is an incredibly useful data structure. It can be used to group together any mix of R structures and objects. A single list could contain a numeric matrix, a logical array, a single character string, and a factor object. You can even have a list as a component of another list. In this section, you'll see how to create, modify, and access components of these flexible structures.

5.1.1 Definition and Component Access

Creating a list is much like creating a vector. You supply the elements that you want to include to the `list` function, separated by commas.

```
R> foo <- list(matrix(data=1:4,nrow=2,ncol=2),c(T,F,T,T),"hello")
R> foo
[[1]]
[,1] [,2]
[1,] 1 3
[2,] 2 4

[[2]]
[1] TRUE FALSE TRUE TRUE

[[3]]
[1] "hello"
foo
```

×In the list, you've stored a 2x2 numeric matrix, a logical vector, and a character string. These are printed in the order they were supplied to list. If you want to check the number of components in a list,

```
R>
length(x=foo)
[1] 3
```

You can retrieve components from a list using indexes, which are entered in double square brackets.

```
R> foo[[1]]
[,1] [,2]
[1,] 1 3 [2,] 2
4 R> foo[[3]]
[1] "hello"
```

This action is known as a *member reference*. When you've retrieved a component this way, you can treat it just like a stand-alone object in the workspace; there's nothing special that needs to be done.

```
R> foo[[1]] + 5.5
[,1] [,2]
[1,] 6.5 8.5
[2,] 7.5 9.5
R> foo[[1]][1,2]
[1] 3
R> foo[[1]][2,]
[1] 2 4
R> cat(foo[[3]], "you!")
hello you!
```

To overwrite a member of `foo` , you use the assignment operator.

```
R> foo[[3]]
[1] "hello"
R> foo[[3]] <-
paste(foo[[3]], "you!") R> foo
[[1]]
[1] [,2]
[1,] 1 3
[2,] 2 4
[[2]]
[1] TRUE FALSE TRUE TRUE
[[3]]
[1] "hello you!"
```

Suppose now you want to access the second and third components of `foo` and store them as one object. Your first instinct might be to try something like this:

```
R> foo[[c(2,3)]]
[1] TRUE
```

But R hasn't done what you wanted. Instead, it returned the third element of the second component. This is because using double square brackets on a list is always interpreted with respect to a single member. Fortunately, member referencing with the double square brackets is not the only way to access components of a list. You can also use single square bracket notation. This is referred to as *list slicing*, and it lets you select multiple list items at once.

```
R> bar <- foo[c(2,3)]
R> bar
[[1]]
[1] TRUE FALSE TRUE
TRUE

[[2]]
[1] "hello you!"
```

Note that the result `bar` is itself a list with the two components stored in the order in which they were requested.

5.1.2 Naming

You can *name* list components to make the elements more recognizable and easy to work with. Just like the information stored about factor levels (as you saw in Section 4.3.1), a name is an R *attribute*.

Let's start by adding names to the list `foo` from earlier.

```
R> names(foo) <-  
c("mymatrix","mylogicals","mystring") R> foo  
$mymatrix  
[1,] [,2]  
[1,] 1 3  
[2,] 2 4  
  
$mylogicals  
[1] TRUE FALSE TRUE TRUE  
  
$mystring  
[1] "hello you!"
```

This has changed how the object is printed to the console. Where earlier it printed `[[1]]`, `[[2]]`, and `[[3]]` before each component, now it prints the names you specified: `$mymatrix`, `$mylogicals`, and `$mystring`. You can now perform member referencing using these names and the dollar operator, rather than the double square brackets.

```
R>  
foo$mymatrix  
[1,] [,2]  
[1,] 1 3 [2,] 2 4
```

This is the same as calling `foo[[1]]`. In fact, even when an object is named, you can still use the numeric index to obtain a member.

```
R> foo[[1]]  
[1,] [,2]  
[1,] 1 3 [2,] 2 4
```

Subsetting named members also works the same way.

```
R> all(foo$mymatrix[,2]==foo[[1]][,2])  
[1] TRUE
```

This confirms (using the function you saw in Section 4.1.2) that these two ways of extracting the second column of the matrix in `foo` provide an identical result.

To name the components of a list as it's being created, assign a label to each component in the list command. Using some components of `foo`, create a new, named list.

```
R> baz <- list(tom=c(foo[[2]],T,T,T,F),dick="g'day  
mate",harry=foo$mymatrix*2) R> baz  
$tom
```

```
[1] TRUE FALSE TRUE TRUETRUETRUETRUE FALSE
```

```
$dick
```

```
[1] "g'day mate"
```

```
$harry
```

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

```
[1,] 2 6
```

```
[2,] 4 8
```

The object baz

harry. now contains the three named components tom, dick, and

```
R> names(baz)
```

```
[1]"tom" "dick" "harry"
```

If you want to rename these members, you can simply assign a character vector of length 3 to names(baz), the same way you did for foo earlier.

NOTE

When using the names function, the component names are always provided and returned as character strings in double quotes. However, if you're specifying names when a list is created (inside the list function) or using names to extract members with the dollar operator, the names are entered without quotes (in other words, they are not given as strings).

5.1.3 Nesting

As noted earlier, a member of a list can itself be a list. When nesting lists like this, it's important to keep track of the depth of any member for subsetting or extraction later.

Note that you can add components to any existing list by using the dollar operator and a new name. Here's an example using foo and baz from earlier:

```
R> baz$bobby <- foo
```

```
R> baz
```

```
$tom
```

```
[1] TRUEFALSE TRUETRUETRUETRUETRUE FALSE
```

```
$dick
```

```
[1] "g'day mate"
```

```
$harry
```

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

```
[1,] 2 6
```

```
[2,] 4 8
```

```
$bobby
```

```

$bobby$mymatrix
[,1] [,2]
[1,] 1 3
[2,] 2 4

$bobby$mylogicals
[1] TRUE FALSE TRUE TRUE

$bobby$mystring
[1] "hello you!"

```

Here you've defined a fourth component to the list `baz` called `bobby`. The member `bobby` is assigned the entire list `foo`. As you can see by printing the new `baz`, there are now three components in `bobby`. Naming and indexes

are now both layered, and you can use either (or combine them) to retrieve members of the inner list.

```

R>
baz$bobby$mylogicals[1:3]
[1] TRUE FALSE TRUE
R> baz[[4]][[2]][1:3]
[1] TRUE FALSE TRUE
R> baz[[4]]$mylogicals[1:3]
[1] TRUE FALSE TRUE

```

These all instruct R to return the first three elements of the logical vector stored as the second component (`[[2]]`, also named `mylogicals`) of the list `bobby`, which in turn is the fourth component of the list `baz`. As long as you're aware of what is returned at each layer of a subset, you can continue

to subset as needed using names and numeric indexes. Consider the third line in this example. The first layer of the subset is `baz[[4]]`, which is a list with three components. The second layer of subsetting extracts the component `mylogicals` from that list by calling `baz[[4]]$mylogicals`. This component represents a vector of length 4, so the third layer of subsetting retrieves the first three elements of that vector with the line `baz[[4]]$mylogicals[1:3]`.

Lists are often used to return output from various R functions. But they can quickly become rather large objects in terms of system resources to store. It's generally recommended that when you have only one type of data, you should stick to using basic vector, matrix, or array structures to record

and store the observations.

Exercise 5.1

- a. Create a list that contains, in this order, a sequence of 20 evenly spaced numbers between -4 and 4; a 3×3 matrix of the logical vector `c(F,T,T,F,T,T,F,F)` filled column-wise; a character vector

with the two strings "don" and "quixote"; and a factor vector containing the observations c("LOW","MED","LOW","MED","MED","HIGH")
 con- Then, do the following:

- i. Extract row elements 2 and 1 of columns 2 and 3, in that order, of the logical matrix.
- ii. Use sub to overwrite "quixote" with "Quixote" and "don" with "Don" inside the list. Then, using the newly overwritten list member, concatenate to the console screen the following statement exactly:

```
"Windmills! ATTACK!"
-\Don Quixote/-
```

- iii 4. Obtain the values from these sequence between are greater than 1.
 - iv. Using which, determine which indexes in the factor vector are assigned the "MED" level.
- b. Create a new list with the factor vector from (a) as a component named "facs"; the numeric vector c(3,2.1,3.3,4,1.5,4.9) as a component named "nums"; and a nested list comprised of the first three members of the list from (a) (use list slicing to obtain this), named "oldlist". Then, do the following:
- i. Extract the elements of "facs" that correspond to elements of "nums" that are greater than or equal to 3.
 - ii. Add a new member to the list named "flags". This member should be a logical vector of length 6, obtained as a twofold repetition of the third column of the logical matrix in the "oldlist" component.
 - iii. Use "flags" and the logical negation operator ! to extract the entries of "num" corresponding to FALSE.
 - iv. Overwrite the character string vector component of "oldlist" with the single character string "Don Quixote".

5.2 DataFrames

A *data frame* is R's most natural way of presenting a data set with a collection of recorded observations for one or more variables. Like lists, data frames have no restriction on the data types of the variables; you can store numeric data, factor data, and so on. The R data frame can be thought of as a list with some extra rules attached. The most important distinction is that in a data frame (unlike a list), the members must all be vectors of equal length. The data frame is one of the most important and frequently used tools in R for statistical data analysis. In this section, you'll look at how to create data frames and learn about their general characteristics.

5.2.1 Construction

To create a data frame from scratch, use the `data.frame` function. You supply your data, grouped by variable, as vectors of the same length—the same way you would construct a named list. Consider the following example data set:

```
R> mydata <-  
data.frame(person=c("Peter","Lois","Meg","Chris","Stewie"),  
age=c(42,40,17,14,1),  
sex=factor(c("M","F","F","M","M")))  
R> mydata  
person age sex  
1 Peter 42 M 2  
Lois 40 F 3  
Meg 17 F 4  
Chris 14 M  
5Stewie 1 M
```

Here, you’ve constructed a data frame with the first name, age in years, and sex of five individuals. The returned object should make it clear why vectors passed to `data.frame` must be of equal length: vectors of differing lengths wouldn’t make sense in this context. If you pass vectors of unequal length to `data.frame`, then R will attempt to recycle any shorter vectors to match the longest, throwing your data off and potentially allocating observations to the wrong variable. Notice that data frames are printed to the console in rows and columns—they look more like a matrix than a named list. This natural spreadsheet style makes it easy to read and manipulate data sets. Each row in a data frame is called a *record*, and each column is a *variable*.

You can extract portions of the data by specifying row and column index positions (much as with a matrix). Here’s an example:

```
R>  
mydata[2,2]  
[1] 40
```

This gives you the element at row 2, column 2—the age of Lois. Now extract the third, fourth, and fifth elements of the third column:

```
R>  
mydata[3:5,3]  
[1] FMM  
Levels: F M
```

This returns a factor vector with the sex of Meg, Chris, and Stewie. The following extracts the entire third and first columns (in that order):

```
R>  
mydata[,c(3,1)]  
sex person  
1 M Peter  
2 F Lois  
3 F Meg
```

```
4 M Chris
5 M
  Stewie
```

This results in another data frame giving the sex and then the name of each person.

You can also use the names of the vectors that were passed to `data.frame` to access variables even if you don't know their column index positions, which can be useful for large data sets. You use the same dollar operator you used for member-referencing named lists.

```
R> mydata$age
[1] 42 40 17 141
```

You can subset this returned vector, too:

```
R>
mydata$age[2]
[1] 40
```

This returns the same thing as the earlier call of `mydata[2,2]`.

You can report the size of a data frame—the number of records and variables—just as you've seen for the dimensions of a matrix (first shown in Section 3.1.3).

```
R>
nrow(mydata)
[1] 5
R>
ncol(mydata)
[1] 3
R>
```

The function `nrow` retrieves the number of rows (records), `ncol` retrieves the number of columns (variables), and `dim` retrieves both. R's default behavior for character vectors passed to `data.frame` is to convert each variable into a factor object. Observe the following:

```
R> mydata$person
[1] Peter Lois Meg Chris Stewie
Levels: Chris Lois Meg Peter Stewie
```

Notice that this variable has levels, which shows it's being treated as a factor. But this isn't what you intended when you defined `mydata` earlier—you explicitly defined `sex` to be a factor but left `person` as a vector of character strings. To prevent this automatic conversion of character strings to factors when using `data.frame`, set the optional argument `stringsAsFactors` to `FALSE` (otherwise, it defaults to `TRUE`). Reconstructing `mydata` with this in place looks like this:

```
R> mydata <-
data.frame(person=c("Peter","Lois","Meg","Chris","Stewie"),
age=c(42,40,17,14,1),
```

```

sex=factor(c("M","F","F","M","M")),
stringsAsFactors=FALSE)
R> mydata
person age sex
1 Peter 42 M
2 Lois 40 F
3 Meg 17 F
4 Chris 14 M
5Stewie 1 M
R> mydata$person
[1]"Peter" "Lois" "Meg" "Chris" "Stewie"

```

You now have person in the desired, nonfactor form.

5.2.2 Adding Data Columns and Combining Data Frames

Say you want to add data to an existing data frame. This could be a set of observations for a new variable (adding to the number of columns), or it could be more records (adding to the number of rows). Once again, you can use some of the functions you've already seen applied to matrices. Recall the `rbind` and `cbind` functions from Section 3.1.2, which let you append rows and columns, respectively. These same functions can be used to extend data frames intuitively. For example, suppose you had another record to include in `mydata`: the age and sex of another individual, Brian. The first step is to create a new data frame that contains Brian's information.

```

R> newrecord <- data.frame(person="Brian",age=7,
sex=factor("M",levels=levels(mydata$sex)))
R> newrecord
person age sex
1 Brian 7 M

```

To avoid any confusion, it's important to make sure the variable names and the data types match the data frame you're planning to add this to. Note that for a factor, you can extract the levels of the existing factor variable using `levels`.

Now, you can simply call the following:

```

R> mydata <-
rbind(mydata,newrecord)
mydata
person age sex
1 Peter 42 M
2 Lois 40 F
3 Meg 17 F
4 Chris 14 M
5Stewie 1 M
6 Brian 7 M

```

Using `rbind`, you combined `mydata` with the new record and overwrote `mydata` with the result.

Adding a variable to a data frame is also quite straightforward. Let's say you're now given data on the classification of how funny these six individuals are, defined as a "degree of funniness." The degree of funniness can take three possible values: Low, Med (medium), and High. Suppose Peter, Lois, and Stewie have a high degree of funniness, Chris and Brian have a medium degree of funniness, and Meg has a low degree of funniness. In R, you'd have a factor vector like this:

```
R> funny <- c("High", "High", "Low", "Med", "High", "Med")
R> funny <- factor(x=funny, levels=c("Low", "Med", "High"))
R> funny
[1] High High Low Med High Med
Levels: Low Med High
```

The first line creates the basic character vector `funny`, and the second line overwrites by turning it into a factor. The order of these elements must correspond to the records in your data frame. Now, you can simply use `cbind` to append this factor vector as a column to the existing `mydata`.

```
R> mydata <- cbind(mydata, funny)
R> mydata
  person age sex funny
1 Peter  42  M   High
2 Lois  40  F   High
3 Meg   17  F    Low
4 Chris 14  M   Med
5 Stewie 1  M   High
6 Brian  7  M   Med
```

`rbind` `cbind`

The `and` functions aren't the only ways to extend a data frame. One useful alternative for adding a variable is to use the dollar operator, much like adding a new member to a named list, as in Section 5.1.3. Suppose now you want to add another variable to `mydata` by including a column with the age of the individuals in months, not years, calling this new variable `age.mon`.

```
R> mydata$age.mon <- mydata$age*12
R> mydata
  person age sex funny age.mon
1 Peter  42  M   High  504
2 Lois  40  F   High  480
3 Meg   17  F    Low  204
4 Chris 14  M   Med  168
5 Stewie 1  M   High  12
6 Brian  7  M   Med   84
```

This creates a new `age.mon` column with the dollar operator and at the same time assigns it the vector of ages in years (already stored as `age`) multiplied by 12.

5.2.3 Logical Record Subsets

In Section 4.1.5, you saw how to use logical flag vectors to subset data structures. This is a particularly useful technique with data frames, where you'll often want to examine a subset of entries that meet certain criteria. For example, when working with data from a clinical drug trial, a researcher might want to examine the results for just male participants and compare them to the results for females. Or the researcher might want to look at the characteristics of individuals who responded most positively to the drug.

Let's continue to work with `mydata`. Say you want to examine all records corresponding to males. From Section 4.3.1, you know that the following line will identify the relevant positions in the `sex` factor vector:

```
R> mydata$sex=="M"
[1] TRUE FALSE FALSE TRUE TRUE TRUE
```

This flags the male records. You can use this with the matrix-like syntax you saw in Section 5.2.1 to get the male-only subset.

```
R>
mydata[mydata$sex=="M",]
      person age sex funny age.mon
1 Peter   42  M  High  504
4 Chris14  M  Med  168 5Stewie
1 M High 12 6 Brian 7 M Med
```

⁸⁴This returns data for all variables for only the male participants. You can use the same behavior to pick and choose which variables to return in the subset. For example, since you know you are selecting the males only, you could omit `sex` from the result using a negative numeric index in the column dimension.

```
R> mydata[mydata$sex=="M",-3]
      person age funny age.mon
1 Peter42High 504
4 Chris 14 Med 168
5Stewie 1 High 12
6 Brian 7 Med 84
```

If you don't have the column number or if you want to have more control over the returned columns, you can use a character vector of variable names instead.

```
R>
mydata[mydata$sex=="M",c("person","age","funny","age.mon")]
person age funny age.mon
1Peter42High 504
4 Chris 14 Med 168
5Stewie 1 High 12
6 Brian 7 Med 84
```

The logical conditions you use to subset a data frame can be as simple or as complicated as you need them to be. The logical flag vector you place in the square brackets just has to match the number of records in the data frame. Let's extract from mydata the full records for individuals who are more than 10 years old OR have a high degree of funniness.

```
R>
mydata[mydata$age>10|mydata$funny=="High",]
person age sex funny age.mon
1 Peter 42 M High 504
2 Lois40 FHigh 480
3 Meg 17 F Low 204
4Chris14 M Med 168
5Stewie 1 M High 12
```

Sometimes, asking for a subset will yield no records. In this case, R returns a data frame with zero rows, which looks like this:

```
R> mydata[mydata$age>45,]
[1]personage sex funnyage.mon
<0 rows> (or 0-length row.names)
```

In this example, no records are returned from mydata because there are no individuals older than 45. To check whether a subset will contain any records, you can use `nrow` on the result—if this is equal to zero, then no records have satisfied the specified condition(s).

Exercise 5.2

- a. Create and store this data frame as `df` in your R workspace:

person	sex	funny
Stan	M	Hig
Francine	F	h
Steve	M	Med
Roger	M	Low
Hayley	F	Hig
Klaus M		h
		Med
		Med

The variables `person`, `sex`, and `funny` should be identical in nature to the variables in the `mydata` object studied throughout Section 5.2. That is, `person` should be a character vector, `sex` should be a factor with levels `F` and `M`, and `funny` should be a factor with levels `Low`, `Med`, and `High`.

- b. Stan and Francine are 41 years old, Steve is 15, Hayley is 21, and Klaus is 60. Roger is extremely old—1,600 years. Append these data as a new numeric column variable in `dframe` called `age`.
- c. Use your knowledge of reordering the column variables based on column index positions to overwrite `dframe`, bringing it in line with `mydata`. That is, the first column should be `person`, the second column `age`, the third column `sex`, and the fourth column `funny`.
- d. Turn your attention to `mydata` as it was left after you included the `age.mon` variable in Section 5.2.2. Create a new version of `mydata` called `mydata2` by deleting the `age.mon` column.
- e. Now, combine `mydata2` with `dframe`, naming the resulting object `mydataframe`.
- f. Write a single line of code that will extract from `mydataframe` just the names and ages of any records where the individual is female and has a level of funniness equal to `Med` OR `High`.
- g. Use your knowledge of handling character strings in R to extract all records from `mydataframe` that correspond to people whose names start with `S`. Hint: Recall `substr` from Section 4.2.4 (note that `substr` can be applied to a vector of multiple character strings).

Important Code in This Chapter

Function/operator	Brief description	First occurrence
<code>list</code>	Create a list	Section 5.1.1, p. 89
<code>[]</code>	Unnamed member reference	Section 5.1.1, p. 90
<code>[]</code>	List slicing (multiple members)	Section 5.1.1, p. 91
<code>\$</code>	Get named member/variable	Section 5.1.2, p. 92
<code>data.frame</code>	Create a data frame	Section 5.2.1, p. 96
<code>mydata[,]</code>	Extract data frame row/columns	Section 5.2.1, p. 96

6

SPECIAL VALUES, CLASSES, AND COERCION



You've now learned about numeric values, logicals, character strings, and factors, as well as their unique properties and applications. Now you'll look at some special values in R that aren't as well-defined. You'll see how they might come about and how to handle and test for them. Then you'll look at different data types in R and some general object class concepts.

6.1 SomeSpecialValues

Many situations in R call for special values. For example, when a data set has missing observations or when a practically infinite number is calculated, the software has some unique terms that it reserves for these situations. These special values can be used to mark abnormal or missing values in vectors, arrays, or other data structures.

6.1.1 Infinity

In Section 2.1, I mentioned that R imposes limits on how extreme a number can be before the software cannot reliably represent it. When a number is too large for R to represent, the value is deemed to be *infinite*. Of course, the mathematical concept of infinity (∞) does not correspond to a specific number—R simply has to define an extreme cutoff point. The precise cutoff value varies from system to system and is governed in part by the amount of memory R has access to. This value is represented by the special object `Inf`, which is case sensitive. Because it represents a numeric value, `Inf` can be associated only with numeric vectors. Let's create some objects to test it out.

```
R> foo <- Inf
R> foo
[1] Inf
R> bar <- c(3401,Inf,3.1,-555,Inf,43)
R> bar
[1] 3401.0 Inf 3.1 -555.0 Inf 43.0
R> baz <- 90000^100
R> baz
[1] Inf
```

Here, you've defined a `foo` object that is a single instance of an infinite value. You've also defined a numeric vector, `bar`, with two infinite elements, and then raised 90,000 to a power of 100 in `baz` to produce a result R deems infinite.

R can also represent negative infinity, with `-Inf`.

```
R> qux <- c(-42,565,-Inf,-Inf,Inf,-45632.3)
R> qux
[1] -42.0 565.0 -Inf -Inf Inf -45632.3
```

This creates a vector with two negative-infinite values and one positive-infinite value.

Though infinity does not represent any specific value, to a certain extent you can still perform mathematical operations on infinite values in R. For example, multiplying `Inf` by any negative value will result in `-Inf`.

```
R>
Inf*-9 [1]
-Inf
```

If you add to or multiply infinity, you also get infinity as a result.

```
R> Inf+1
[1] Inf
R> 4*-Inf
[1] -Inf
R> -45.2-Inf
[1] -Inf
```

```
R> Inf-
45.2 [1] Inf
R> Inf+Inf
[1] Inf
R> Inf/23
[1] Inf
```

Zero and infinity go hand in hand when it comes to division. Any (finite) numeric value divided by infinity, positive or negative, will result in zero.

```
R> -59/Inf
[1] 0
R> -59/-
Inf [1] 0
```

Though it isn't mathematically defined, note that in R, any nonzero value divided by zero will result in infinity (positive or negative depending on the sign of the numerator).

```
R>
-59/0
[1] -Inf
R> 59/0
[1] Inf
R>
Inf/0 [1]
Inf
```

Often, you'll simply want to detect infinite values in a data structure. The functions `is.infinite` and `is.finite` take in a collection of values, typically a vector, and return for each element a logical value answering the question posed. Here's an example using `qux` from earlier:

```
R> qux
[1] -42.0 565.0 -Inf -Inf R> Inf -45632.3
is.infinite(x=qux)
[1] FALSE FALSE TRUE TRUE TRUE
FALSE R> is.finite(x=qux)
[1] TRUE TRUE FALSE FALSE FALSE
TRUE
```

Note that these functions do not distinguish between positive or negative infinity, and the result of `is.finite` will always be the opposite (the negation) of the result of `is.infinite`. Finally, relational operators function as you might expect.

```
R> -
Inf<Inf [1]
TRUE
R> Inf>Inf
[1] FALSE
```

```
R> qux==Inf
[1] FALSE FALSE FALSE FALSE TRUE
FALSE R> qux==-Inf
[1] FALSE FALSE TRUE TRUE FALSE FALSE
```

Here, the first line confirms that `Inf` is treated as less than `Inf`, and the second line shows that `Inf` is not greater than `Inf`. The third and fourth lines, again using `qux`, test for equality, which is a useful way to distinguish between positive and negative infinity if you need to.

6.1.2 NaN

In some situations, it's impossible to express the result of a calculation using a number, `Inf`, or `-Inf`. These difficult-to-quantify special values are labeled `NaN` in R, which stands for *Not a Number*.

As with infinite values, `NaN` values are associated only with numeric observations. It's possible to define or include a `NaN` value directly, but this is rarely the way they're encountered.

```
R> foo <- NaN
R> foo
[1] NaN
R> bar <- c(NaN,54.3,-2,NaN,90094.123,-
Inf,55) R> bar
[1] NaN 54.30 -2.00 NaN 90094.12 -Inf 55.00
```

`NaN` Typically, is the unintended result of attempting a calculation that's impossible to perform with the specified values.

In Section 6.1.1, you saw that adding or subtracting from `Inf` or `-Inf` will simply result again in `Inf` or `-Inf`. However, if you attempt to cancel representations of infinity in any way, the result will be `NaN`.

```
R> -
Inf+Inf [1]
NaN
R> Inf/Inf
[1] NaN
```

Here, the first line won't result in zero because positive and negative infinity can't be interpreted in that numeric sense, so you get `NaN` as a result. The same thing happens if you attempt to divide `Inf` by itself. In addition, although you saw earlier that a nonzero value divided by zero will result in positive or negative infinity, `NaN` results when *zero* is divided by zero.

```
R>
0/0 [1]
NaN
```

Note that any mathematical operation involving `NaN` will simply result in `NaN`.

```

R> NaN+1
[1] NaN
R> 2+6*(4-4)/0
[1] NaN
R> 3.5^(-Inf/Inf)
[1] NaN

```

In the first line, adding 1 to “not a number” is still NaN. In the second line, you obtain NaN from the $(4-4)/0$, which is clearly $0/0$, so the result is also NaN. In the third line, NaN results from $-\text{Inf}/\text{Inf}$, so the result of the remaining calculation is again NaN. This begins to give you an idea of how NaN or infinite values might unintentionally crop up. If you have a function where various values are passed to a fixed calculation and you don’t take care to prevent, for example, from occurring, then the code will return NaN. Likewise, a special function (`is.nan`) is used to detect the presence of NaN values. Unlike infinite values, however, relational operators cannot be used with NaN. Here’s an example using `bar`, which was defined earlier:

```

R> bar
[1] NaN 54.30 -2.00 NaN 90094.12 -Inf 55.00
R> is.nan(x=bar)
[1] TRUE FALSE FALSE TRUE FALSE FALSE FALSE
R> !is.nan(x=bar)
[1] FALSE TRUE TRUE FALSE TRUE TRUE TRUE
R> is.nan(x=bar)|is.infinite(x=bar)
[1] TRUE FALSE FALSE TRUE FALSE TRUE FALSE
R> bar[!(which(is.nan(x=bar)|is.infinite(x=bar)))]
[1] 54.30 -2.00 90094.12 55.00
is.nan bar NaN
!

```

Using the function on flags the two positions as TRUE. In the second example, you use the negation operator to flag the positions where the elements are NOT NaN. Using the element-wise OR, (see Section 4.1.3), you then identify elements that are either OR infinite. Finally, the last line uses `which` to convert these logical values into numeric index positions so that you can remove them with negative indexes in square brackets (see Section 4.1.5 for a refresher on using `which`). You can find more details on the functionality and behavior of NaN and Inf in the R help file by entering `?Inf` at the prompt.

Exercise 6.1

- Store the following vector:

```

foo <-
c(13563, -14156, -14319, 16981, 12921, 11979, 9568, 8833, -12968, 8133)

```

Then, do the following:

- i. Output all elements of `foo` that, when raised to a power of 75, are NOT infinite.
 - ii. Return the elements of `foo`, excluding those that result in negative infinity when raised to a power of 75.
- b. Store the following 3
× 4 matrix as the object `bar`:

```
77875.4027551.4523764.30-36478.88
```

```
35466.2573... --333853659969-7058569
```

```
-39803.8155976.3476694.8247032.00
```

Now, do the following:

- i. Identify the coordinate-specific indexes of the entries of `bar` that are NaN when you raise `bar` to a power of 65 and divide by infinity.
- ii. Return the values in `bar` that are NOT NaN when `bar` is raised to a power of 67 and infinity is added to the result. Confirm this is identical to identifying those values in `bar` that, when raised to a power of 67, are not equal to negative infinity.
- iii. Identify those values in `bar` that are either negative infinity OR finite when you raise `bar` to a power of 67.

6.1.3 NA

In statistical analyses, data sets often contain missing values. For example, someone filling out a questionnaire may not respond to a particular item, or a researcher may record some observations from an experiment incorrectly. Identifying and handling missing values is important so that you can still use the rest of the data. R provides a standard special term to represent missing values, `NA`, which reads as *Not Available*.

`NA` entries are not the same as `NaN` entries. Whereas `NaN` is used only with respect to numeric operations, missing values can occur for any type of observation. As such, `NA`s can exist in both numeric and non-numeric settings. Here's an example:

```
R> foo <- c("character","a",NA,"with","string",NA)
R> foo
[1]"character""a" NA "with" "string"NA
R> bar
      bar
factor(c("blue",NA,NA,"blue","green","blue",NA,"red","red",NA, "green"))
R> bar
[1]blue <NA> <NA> blue greenblue <NA> red red <NA> green
Levels: blue green red
R> baz <- matrix(c(1:3,NA,5,6,NA,8,NA),nrow=3,ncol=3)
```

```
R> baz
      [,1] [,2] [,3]
[1,] 1 NA NA [2,] 2
5 8 [3,] 3 6 NA
```

```
foo
```

The object is a character vector with entries 3 and 6 missing; bar is a factor vector of length 11 with elements 2, 3, 7, and 10 missing; and baz is a numeric matrix with row 1, columns 2 and 3, and row 3, column 3, elements missing. In the factor vector, note that the NAs are printed as <NA>. This is to differentiate between bona fide levels of the factor and the missing observations, to prevent NA from being mistakenly interpreted as one of the levels. Like the other special values so far, you can identify NA elements using the function `is.na`. This is often useful for removing or replacing NA values. Consider the following numeric vector:

```
R> qux <- c(NA, 5.89, Inf, NA, 9.43, -2.35, NaN, 2.10, -8.53, -7.58, NA, -4.58, 2.01, NaN)
R> qux
      [1] NA 5.89 Inf NA 9.43 -2.35 NaN 2.10 -8.53 -7.58 NA -4.58
[13] 2.01 NaN
```

This vector has a total of 14 entries, including NA, NaN, and Inf.

```
R> is.na(x=qux)
[1] TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
[13] FALSE TRUE
```

`is.na` As you can see, flags the corresponding NA entries in `qux` as TRUE. But this is not all—note that it also flags elements 7 and 14, which are NaN, not NA. Strictly speaking, NA, NaN, and Inf are different entities, but numerically they are practically the same since there is almost nothing you can do with either. `is.na` treats them all as NA, allowing the user to remove or recode them at the same time. If you want to identify and entries separately, you can use `is.nan` in conjunction with logical operators. Here's an example:

```
R>
which(x=is.nan(x=qux))
[1] 7 14
```

This identifies the index positions whose elements are specifically NaN. If you want to identify entries only, try the following:

```
R> which(x=(is.na(x=qux)&!is.nan(x=qux)))
[1] 1 4 11
```

This identifies the element indexes for only the NA entries (by checking for entries where `is.na` is TRUE AND where `is.nan` is NOT TRUE).

After locating the offending elements, you could use negative indexes in square brackets to remove them, though R offers a more direct option. The `na.omit()` function will take a structure and delete all `NA`s from it; `na.omit()` will also apply to `NA`s if the elements are numeric.

```
R> quux <- na.omit(object=qux)
R> quux
[1] 5.89 Inf 9.43 -2.35 2.10 -8.53 -7.58 -4.58 2.01
attr(,"na.action")
[1] 1 4 7 11 14
attr(,"class")
[1] "omit"
```

`na.omit()` Note that the structure passed to `is` is given as the argument `object` and that some additional output is displayed in printing the returned object. These extra details are provided to inform the user that there were elements in the original vector that were removed (in this case, the element positions provided in the attribute `na.action`). Attributes will be discussed more in Section 6.2.1.

Similar to `NaN`, arithmetic calculations with `NA` result in `NA`. Using relational operators with either `NaN` or `NA` will also result in `NA`.

```
R> 3+2.1*NA-4
[1] NA
R> 3*c(1,2,NA,NA,NaN,6)
[1] 3 6 NA NANaNA      18
R> NA>76
[1] NA
R> 76>NaN
[1] NA
```

You can find more details on the usage and finer technicalities of `NA` values by entering `?NA`.

6.1.4 NULL

Finally, you'll look at the *null* value, written as `NULL`. This value is often used to explicitly define an “empty” entity, which is quite different from a “missing” entity specified with `NA`. An instance of `NA` clearly denotes an existing position that can be accessed and/or overwritten if necessary—not so for `NULL`. You can see an indication of this if you compare the assignment of `NA` with the assignment of a `NULL`.

```
R> foo <-
NULL R> foo
NULL
R> bar <- NA
```

```
R>
bar
[1] NA
```

Note that `bar`, the `NA` object, is printed with an index position `[1]`. This suggests you have a vector with a single element. In contrast, you explicitly instructed `foo` to be empty with `NULL`. Printing this object doesn't provide a position index because there is no position to access.

This interpretation of `NULL` also applies to vectors that have other well-defined items. Consider the following two lines of code:

```
R> c(2,4,NA,8)
[1] 2 4 NA 8
R>
c(2,4,NULL,8)
[1] 2 4 8
```

The first line creates a vector of length 4, with the third position coded as `NA`. The second line creates a similar vector but using `NULL` instead of `NA`. The result is a vector with a length of only 3. That's because `NULL` cannot take up a position in the vector. As such, it makes no sense to assign `NULL` to multiple positions in a vector (or any other structure). Again, here's an example:

```
R> c(NA,NA,NA)
[1] NA NA NA
R> c(NULL,NULL,NULL)
NULL
```

The first line can be interpreted as “three possible slots with unrecorded observations.” The second line simply provides “emptiness three times,” which is interpreted as one single, unsubsettable, empty object.

At this point, you might wonder why there is even a need for `NULL`. If something is empty and doesn't exist, why define it in the first place? The answer lies in the need to be able to explicitly state or check whether a certain object has been defined. This occurs often when calling functions in R.

For example, when a function contains optional arguments, internally the function has to check which of those arguments have been supplied and which are missing or empty. The `NULL` value is a useful and flexible tool that the author of a function can use to facilitate such checks. You'll see examples of this later on in Chapter 11.

The `is.null` function is used to check whether something is explicitly `NULL`. Suppose you have a function with an optional argument named `opt.arg` and that, if supplied, `opt.arg` should be a character vector of length 3. Let's

say a user calls this function with the following.

```
R> opt.arg <- c("string1","string2","string3")
```

Now if you check whether the argument was supplied using NA you might call this:

```
R> is.na(x=opt.arg)
[1] FALSE FALSE FALSE
```

The position-specific nature of NA means that this check is element-wise and returns an answer for each value in opt.arg. This is problematic because you want only a single answer—is opt.arg empty or is it supplied?

~~This is~~
when comes to the party.

```
R> is.null(x=opt.arg)
[1] FALSE
```

Quite clearly ^{opt.arg} is not empty, and the function can proceed as necessary. If the argument is empty, using NULL over NA for the check is again better for these purposes.

```
R> opt.arg <- c(NA,NA,NA)
R> is.na(x=opt.arg)
[1] TRUE TRUE TRUE
```

```
R> opt.arg <- c(NULL,NULL,NULL) R>
is.null(x=opt.arg)
[1] TRUE
```

As noted earlier, filling a vector with NULL isn't usual practice; it's done here just for illustration. But usage of NULL is far from specific to this particular example. It's commonly used throughout both ready-to-use and user-contributed functionality in R.

The empty NULL has an interesting effect if it's included in arithmetic or relational comparisons.

```
R> NULL+53
numeric(0)
R>
53<=NULL
logical(0)
```

Rather than NULL as you might expect, the result is an “empty” vector of a type determined by the nature of the operation attempted. NULL typically dominates any arithmetic, even if it includes other special values.

```
R> NaN-
NULL+NA/Inf
numeric(0)
```

~~NULL~~ also occurs naturally when examining lists and data frames. For example, define a new list foo.

```
R> foo <- list(member1=c(33,1,5.2,7),member2="NA or NULL?")
R> foo
$member1
[1]33.0 1.0 5.2 7.0
```

```
$member2
[1] "NA or NULL?"
```

member3

This list obviously doesn't include a member called member3. Look at what happens when you try to access a member in foo by that name:

```
R>
foo$member3
NULL
```

The result of NULL signals that a member called member3 in foo doesn't exist, or in R terms, is empty. Therefore, it can be filled with whatever you want.

```
R> foo$member3 <-
NA R> foo
$member1
[1]33.0 1.0 5.2 7.0
```

```
$member2
[1] "NA or NULL?"
```

```
$member3
[1] NA
```

The same principle applies when querying a data frame for a non-existent column or variable using the dollar operator (as in Section 5.2.2).

For more technical details on how NULL and is.null are handled by R, see the help file accessed by ?NULL.

Exercise 6.2

- a. Consider the following line of code:

```
foo <- c(4.3,2.2,NULL,2.4,NaN,3.3,3.1,NULL,3.4,NA)
```

Decide yourself which of the following statements are true and which are false and then use R to confirm:

- The length of foo is 8.
- Calling which(x=is.na(x=foo)) will not result in 4 and 8.

- iii. Checking `is.null(x=foo)` will provide you with the location of the two NULL values that are present.
- iv. Executing `is.na(x=foo[8])+4/NULL` will not result in NA.
- b. Create and store a list containing a single member: the vector `c(7,7,NA,3,NA,1,1,5,NA)`. Then, do the following:
 - i. Name the member "alpha".
 - ii. Confirm that the list doesn't have a member with the name "beta" using the appropriate logical valued function.
 - iii. Add a new member called beta, which is the vector obtained by identifying the index positions of alpha that are NA.

6.2 Understanding Types, Classes, and Coercion

By now, you've studied many of the fundamental features in the R language for representing, storing, and handling data. In this section, you'll examine how to formally distinguish between different kinds of values and structures and look at some simple examples of conversion from one type to another.

6.2.1 Attributes

Each R object you create has additional information about the nature of the object itself. This additional information is referred to as the object's *attributes*. You've seen a few attributes already. In Section 3.1.3, you identified the dimensions attribute of a matrix using `dim`. In Section 4.3.1, you used levels to get the levels attribute of a factor. It was also noted that names can get the member names of a list in Section 5.1.2, and in Section 6.1.3, that an attribute annotates the result of applying `na.omit`.

In general, you can think of attributes as either *explicit* or *implicit*. Explicit attributes are immediately visible to the user, while R determines implicit attributes internally. You can print explicit attributes for a given object with the `attributes` function, which takes any object and returns a named list. Consider, for example, the following 3 × 3 matrix:

```
R> foo <- matrix(data=1:9,nrow=3,ncol=3)
R> foo
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9

R> attributes(foo)
$dim
[1] 3 3
```

Here, calling `attributes` returns a list with one member: `dim`. Of course, you can retrieve the contents of `dim` with `attributes(foo)$dim`, but if you know the name of an attribute, you can also use `attr`:

```
R> attr(x=foo,which="dim")
[1] 3 3
```

This function takes the object in as `x` and the name of the attribute as `which`. Recall that names are specified as character strings in R. To make things even more convenient, the most common attributes have their own functions (usually named after the attribute) to access the corresponding value. For the dimensions of a matrix, you’ve already seen the function `dim`.

```
R> dim(foo)
[1] 3 3
```

These attribute-specific functions are useful because they also allow access to implicit attributes, which, while still controllable by the user, are set automatically by the software as a matter of necessity. The names and levels functions mentioned earlier are also both attribute-specific functions. Explicit attributes are often optional; if they aren’t specified, they are `NULL`. For example, when building a matrix with the `matrix` function, you can use the optional argument `dimnames` to annotate the rows and columns with names. You pass `dimnames` a list made up of two members, both character vectors of the appropriate lengths—the first giving row names and the second giving column names. Let’s define the matrix as follows:

```
R> bar <- matrix(data=1:9,nrow=3,ncol=3,dimnames=list(c("A","B","C"),
c("D","E","F")))
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

Because the dimension names are attributes, they appear when you call `.attributes(bar)`

```
R> attributes(bar)
$dim
[1] 3 3

$dimnames
$dimnames[[1]]
[1] "A" "B" "C"

$dimnames[[2]]
[1] "D" "E" "F"
```

Note that `dimnames` is itself a list, nested inside the larger attributes list. Again, to extract the values of this attribute, you can use list member referencing, you can use `attr` as shown earlier, or you can use the attribute-specific function.

```
R>
dimnames(bar)
[[1]]
[1] "A" "B" "C"

[[2]]
[1] "D" "E" "F"
```

Some attributes can be modified after an object has been created (as you saw already in Section 5.1.2, where you renamed members of a list). Here, to make `foo` match `bar` exactly, you can give `foo` some `dimnames` by assigning them to the attribute-specific function:

```
R> dimnames(foo) <- list(c("A","B","C"),c("D","E","F")) R>
foo DEF
A 1 4 7
B 2 5 8
C 3 6 9
```

I've used matrices in the discussion here, but optional attributes for other objects in R are treated the same way. Attributes are not restricted to built-in R objects, either. Objects you build yourself can be defined with their own attributes and attribute-specific functions. Just remember that the role of an attribute is typically to provide descriptive data about an object, or you could end up overcomplicating your object structures unnecessarily.

6.2.2 Object Class

An object's *class* is one of the most useful attributes for describing an entity in R. Every object you create is identified, either implicitly or explicitly, with at least one class. R is an *object-oriented* programming language, meaning entities are stored as objects and have methods that act upon them. In such a language, class identification is formally referred to as *inheritance*.

NOTE

S3. This section will focus on the most common classing structure used in R, called S3. There is another structure, S4, which is essentially a more formal set of rules for the identification and treatment of different objects. For most practical intents and certainly for beginners, understanding and using S3 will be sufficient. You can find further details in R's online documentation.

The class of an object is explicit in situations where you have user-defined object structures or an object such as a factor vector or data frame where other attributes play an important part in the handling of the object itself—for example, level labels of a factor vector, or variable names in a data

frame, are modifiable attributes that play a primary role in accessing the observations of each object. Elementary R objects such as vectors, matrices, and arrays, on the other hand, are implicitly classed, which means the class is not identified with the `attributes` function. Whether implicit or explicit, the class of a given object can always be retrieved using the attribute-specific function `class`.

Stand-Alone Vectors

Let's create some simple vectors to use as examples.

```
R> num.vec1 <- 1:4
R> num.vec1
[1] 1 2 3 4
R> num.vec2 <- seq(from=1,to=4,length=6)
R> num.vec2
[1] 1.0 1.6 2.2 2.8 3.4 4.0
R> char.vec <- c("a","few","strings","here")
R> char.vec
[1] "a" "few" "strings" "here"
R> logic.vec <- c(T,F,F,F,T,F,T,T)
R> logic.vec
[1] TRUE FALSE FALSE FALSE TRUE FALSE TRUE TRUE
R> fac.vec <- factor(c("Blue","Blue","Green","Red","Green","Yellow")) R>
fac.vec
[1] Blue Blue Green Red Green Yellow
Levels: Blue Green Red Yellow
```

`class`

You can pass any object to the `class` function, and it returns a character vector as output. Here are examples using the vectors just created:

```
R> class(num.vec1)
[1] "integer"
R> class(num.vec2)
[1] "numeric"
R> class(char.vec)
[1] "character"
R> class(logic.vec)
[1] "logical"
R> class(fac.vec) [1]
"factor"
```

The output from using `class` on the character vector, the logical vector, and the factor vector simply match the kind of data that has been stored. The output from the number vectors is a little more intricate, however. So far, I've referred to any object with an arithmetically valid set of numbers as "numeric." If all the numbers stored in a vector are whole, then R identifies the vector as "integer". Numbers with decimal places (called *floating-point* numbers), on the other hand, are identified as "numeric". This distinction

is necessary because some tasks strictly require integers, not floating-point numbers. Colloquially, I'll continue to refer to both types as "numeric" and

in fact, the `is.numeric` function will return `TRUE` for both integer and floating-point structures, as you'll see in Section 6.2.3.

Other Data Structures

As mentioned earlier, R's classes are essentially designed to facilitate object-oriented programming. As such, `class` usually reports on the nature of the data *structure*, rather than the type of data that's stored—it returns the data type only when used on stand-alone vectors. Let's try it on some matrices.

```
R> num.mat1 <- matrix(data=num.vec1,nrow=2,ncol=2)
R> num.mat1
[,1] [,2]
[1,] 1 3
[2,] 2 4
R> num.mat2 <- matrix(data=num.vec2,nrow=2,ncol=3)
R> num.mat2
[,1] [,2] [,3]
[1,] 1.0 2.2 3.4
[2,] 1.6 2.8 4.0
R> char.mat <- matrix(data=char.vec,nrow=2,ncol=2)
R> char.mat
[,1] [,2]
[1,] "a" "strings"
[2,] "few" "here"
R> logic.mat <- matrix(data=logic.vec,nrow=4,ncol=2) R>
logic.mat
[,1] [,2]
[1,] TRUE TRUE
[2,] FALSE FALSE
[3,] FALSE TRUE
[4,] FALSE TRUE
```

Note from Section 4.3.1 that factors are used only in vector form, so is not included here. Now check these matrices with `class`.

```
R> class(num.mat1)
[1] "matrix"
R> class(num.mat2)
[1] "matrix"
R> class(char.mat)
[1] "matrix"
R> class(logic.mat)
[1] "matrix"
```

You see that regardless of the data type, `class` reports the structure of the object itself—all matrices. The same is true for other object structures, like arrays, lists, and data frames.

Multiple Classes

Certain objects will have multiple classes. A variant on a standard form of

an object, such as an ordered factor vector, will inherit the usual factor class and also contain the additional class. Both are returned if you use the function.

```
R> ordfac.vec <- factor(x=c("Small","Large","Large","Regular","Small"),
  levels=c("Small","Regular","Large"),
  ordered=TRUE)
R> ordfac.vec
[1] Small Large Large Regular Small
Levels: Small < Regular < Large
R> class(ordfac.vec)
[1] "ordered" "factor"
```

ordfac.vec "factor" ordfac.vec

"factor", was identified as only, but the class of

"ordered" components. It's still identified as "factor", but it also includes

"ordered", which identifies

the ordered components. The variant of the class also presents in the output, you can think

words, it is a special case that inherits from, and therefore behaves like, a "factor". For further technical details on R subclasses, I recommend Chapter 9 of by Matloff (2011).

The Art of R Programming

NOTE

I have focused on the class function here because it's directly relevant to the object-oriented programming style exercised in this text, especially in Part II. There are other functions that show some of the complexities of R's classing rules. For example, the function `typeof` reports the type of data contained within an object, not just for vectors but also for matrices and arrays. Note, however, that the terminology in the output of `typeof` doesn't always match the output of `class`. See the help file `?typeof` for details on the values it returns.

To summarize, an object's class is first and foremost a descriptor of the data structure, though for simple vectors, the class function reports the type of data stored. If the vector entries are exclusively whole numbers, then R classes the vector as "integer", whereas "numeric" is used to label a vector with floating-point numbers.

6.2.3 Is-Dot Object-Checking Functions

Identifying the class of an object is essential for functions that operate on stored objects, especially those that behave differently depending on the class of the object. To check whether the object is a specific class or data type, you can use the *is-dot* functions on the object and it will return a TRUE or FALSE logical value.

Is-dot functions exist for almost any sensible check you can think of. For example, consider once more the `num.vec1` vector from Section 6.2.2 and the following six checks:

```
R> num.vec1 <- 1:4
R> num.vec1
[1]1234
R> is.integer(num.vec1)
[1] TRUE
R> is.numeric(num.vec1)
[1] TRUE
R> is.matrix(num.vec1)
[1] FALSE
R> is.data.frame(num.vec1)
[1] FALSE
R> is.vector(num.vec1)
[1] TRUE
R> is.logical(num.vec1)
[1] FALSE
```

The first, second, and sixth is-dot functions check the kind of data stored in the object, while the others check the structure of the object itself. The results are to be expected: `num.vec1` *is* an “integer” (and *is* “numeric”), and it *is* a “vector.” It’s not a matrix or a data frame, nor is it logical.

Briefly, it’s worth noting that these checks use more general categories than the formal classes identified with `class`. Recall that `num.vec1` was identified solely as “integer” in Section 6.2.2, but using `is.numeric` here still returns TRUE. In this example, the `num.vec1` with integer data is generalized to be “numeric”. Similarly, for a data frame, an object of class “data.frame” will return TRUE for `is.data.frame` *and* `is.list` because a data frame is intuitively generalized to a list.

There’s a difference between the object is-dot functions detailed here and functions such as `is.na` discussed in Section 6.1. The functions to check for the special values like NA should be thought of as a check for equality;

they exist because it is not legal syntax to write something like `foo==NA`. Those functions from Section 6.1 thus operate in R’s element-wise fashion, whereas the object is-dot functions inspect the object *itself*, returning only a single logical value.

6.2.4 As-Dot Coercion Functions

You’ve seen different ways to modify an object after it’s been created—by accessing and overwriting elements, for example. But what about the structure of the object itself and the type of data contained within?

Converting from one object or data type to another is referred to as *coercion*. Like other features of R you’ve met so far, coercion is performed either implicitly or explicitly. Implicit coercion occurs automatically when

elements need to be converted to another type in order for an operation to complete. In fact, you’ve come across this behavior already, in Section 4.1.4, for example, when you used numeric values for logical values. Remember that logical values can be thought of as integers—one for TRUE and zero for FALSE. Implicit coercion of logical values to their numeric counterparts occurs in lines of code like this:

```
R> 1:4+c(T,F,F,T)
[1]2235
```

In this operation, R recognizes that you’re attempting an arithmetic calculation with +, so it expects numeric quantities. Since the logical vector is not in this form, the software internally coerces it to ones and zeros before completing the task.

Another frequent example of implicit coercion is when paste and cat are used to glue together character strings, as explored in Section 4.2.2. Non-character entries are automatically coerced to strings before the concatenation takes place. Here’s an example:

```
R> foo <- 34
R> bar <- T
R> paste("Definitely foo: ",foo,"; definitely bar: ",bar,".",sep="") [1] "Definitely
foo: 34; definitely bar: TRUE."
```

34T Here, the integer and the logical are implicitly coerced to characters
~~paste~~ Since R knows the output of must be a string.

In other situations, coercion won’t happen automatically and must be carried out by the user. This explicit coercion can be achieved with the *as-dot* functions. Like *is-dot* functions, *as-dot* functions exist for most typical R data types and object classes. The previous two examples can be coerced explicitly, as follows.

```
R> as.numeric(c(T,F,F,T))
[1]1001
R> 1:4+as.numeric(c(T,F,F,T))
[1]2235
R> foo <- 34
R> foo.ch <- as.character(foo)
R> foo.ch
[1] "34"
R> bar <- T
R> bar.ch <- as.character(bar)
R> bar.ch
[1] "TRUE"
R> paste("Definitely foo: ",foo.ch,"; definitely bar: ",bar.ch,".",sep="") [1] "Definitely
foo: 34; definitely bar: TRUE."
```

Coercions are possible in most cases that “make sense.” For example, it’s easy to see why R is able to read something like this:

```
R> as.numeric("32.4")
[1] 32.4
```

However, the following conversion makes no sense:

```
R> as.numeric("g'day mate")
[1] NA
Warning message:
NAs introduced by coercion
```

Since there is no logical way to translate “g’day mate” into numbers, the entry is returned as NA (in this case, R has also issued a warning message). This means that in certain cases, multiple coercions are needed to attain the final result. Suppose, for example, you have the character vector `c("1","0","1","0","0")` and you want to coerce it to a logical-valued vector. Direct character to logical coercion is not possible, because even if all the character strings contained numbers, there is no guarantee in general that they would all be ones and zeros.

```
R> as.logical(c("1","0","1","0","0")) [1]
NA NA NA NA NA
```

However, you know that character string numbers can be converted to a numeric data type, and you know that ones and zeros are easily coerced to logicals. So, you can perform the coercion in those two steps, as follows:

```
R> as.logical(as.numeric(c("1","0","1","0","0"))) [1]
TRUEFALSETRUEFALSEFALSE
```

Not all data-type coercion is entirely straightforward. Factors, for example, are trickier because R treats the levels as integers. In other words, regardless of how the levels of a given factor are actually labeled, the software will refer to them internally as level 1, level 2, and so on. This is clear if you try to coerce a factor to a numeric data type.

```
R> baz <- factor(x=c("male","male","female","male")) R>
baz
[1]male male femalemale
Levels: female male
R> as.numeric(baz)
[1]2212
```

Here, you see that R has assigned the numeric representation of the factor in the stored order of the factor labels (alphabetic by default). Level 1 refers to female, and level 2 refers to male. This example is simple enough,

though it's important to be aware of the behavior since coercion from factors with numeric levels can cause confusion.

```
R> qux <- factor(x=c(2,2,3,5)) R>
qux
[1] 2 2 3 5
Levels: 2 3 5
R> as.numeric(qux)
[1] 1 1 2 3
```

The numeric representation of the factor `qux` is `c(1,1,2,3)`. This highlights again that the levels of `qux` are simply treated as level 1 (even though it has a label of 2), level 2 (which has a label of 3), and level 3 (which has a label of 5).

Coercion between object classes and structures can also be useful. For example, you might need to store the contents of a matrix as a single vector.

```
R> foo <- matrix(data=1:4,nrow=2,ncol=2)
R> foo
[,1] [,2]
[1,] 1 3
[2,] 2 4
R> as.vector(foo)
[1] 1 2 3 4
```

`as.vector`

Notethat has coerced the matrix by “stacking” the columns into a single vector. The same column-wise deconstruction occurs for higher-dimensional arrays, in order of layer or block.

```
R> bar <- array(data=c(8,1,9,5,5,1,3,4,3,9,8,8),dim=c(2,3,2)) R>
bar ,,1
```

```
 [,1] [,2] [,3]
[1,] 8 9 5
[2,] 1 5 1
```

,,2

```
 [,1] [,2] [,3]
[1,] 3 3 8
[2,] 4 9 8
```

```
R> as.matrix(bar)
[,1]
[1,] 8
[2,] 1
[3,] 9
```

```
[4,] 5
[5,] 5
[6,] 1
[7,] 3
[8,] 4
[9,] 3
[10,] 9
[11,] 8
[12,] 8
```

```
R> as.vector(bar)
[1] 8 1 9 5 5 1 3 4 3 9 8 8
```

as.matrix

× You may want to store the array as a 12
 as.vector stores it as a single vector. Similar commonsense rules for data
 types apply to coercion when working with object structures. For example,
 coercing the following list to a data frame produces an error:

```
R> baz <- list(var1=foo,var2=c(T,F,T),var3=factor(x=c(2,3,4,4,2)))
R> baz
$var1
[,1] [,2]
[1,] 1 3
[2,] 2 4
```

```
$var2
[1] TRUE FALSE TRUE
```

```
$var3
[1] 2 3 4 4 2
Levels: 2 3 4
```

```
R> as.data.frame(baz)
Error in data.frame(var1 = 1:4, var2 = c(TRUE, FALSE, TRUE), var3 =  

  c(1L, arguments imply differing number of rows: 2, 3, 5) :
```

The error occurs because the variables do not have matching lengths.
 But there is no problem with coercing the list qux, shown here, which
 has equal-length members:

```
R> qux <- list(var1=c(3,4,5,1),var2=c(T,F,T,T),var3=factor(x=c(4,4,2,1))) R>
qux $var1
[1] 3 4 5 1

$var2
[1] TRUE FALSE TRUE TRUE
```

```
$var3
[1]4421
Levels: 1 2 4
```

```
R> as.data.frame(qux)
var1 var2var3
1 3TRUE 4
2 4FALSE 4
3 5TRUE 2
4 1TRUE 1
```

This stores the variables as a data set in a column-wise fashion, in the order that your list supplies them as members. This discussion on object classes, data types, and coercion is not exhaustive, but it serves as a useful introduction to how R deals with issues surrounding the formal identification, description, and handling of the objects you create—issues that are present for most high-level languages. Once you're more familiar with R, the help files (such as the one accessed by entering `?as` at the prompt) provide further details about object handling in the software.

Exercise 6.3

- a. Identify the class of the following objects. For each object, also state whether the class is explicitly or implicitly defined.
 - i. `foo<-array(data=1:36,dim=c(3,3,4))`
 - ii. `bar<-as.vector(foo)`
 - iii. `baz <- as.character(bar)`
 - iv. `qux <- as.factor(baz)`
 - v. `quux<-bar+c(-0.1,0.1)`
- b. For each object defined in (a), find the sum of the result of calling `is.numeric` and `is.integer` on it separately. For example, `is.numeric(foo)+is.integer(foo)` would compute the sum for (i). Turn the collection of five results into a factor with levels 0, 1, and 2, identified by the results themselves. Compare this factor vector with the result of coercing it to a numeric vector.
- c. Turn the following:

```
      [,1] [,2] [,3] [,4]
[1,]  2  5  8 11 [2,]  3  6  9
12 [3,]  4  7 10 13
```

into the following:

```
[1] "2"  "5"  "8"  "11" "3"  "6"  "9"  "12" "4"  "7"  "10" "13"
```

d. Store the following matrix:

```
3401
2312
3311
4 2 0 1
```

4102 Then, do the following:

- i. Coerce the matrix to a data frame.
- ii. As a data frame, coerce the second column to be logical-valued.
- iii. As a data frame, coerce the third column to be factor-valued.

Important Code in This Chapter

Function/operator	Brief descriptionFirst occurrence
Inf, -Inf	V
is.infinite	±ianlufien iftoyrSection 6.1.1, p. 104
is.finite	Element-wise check for InfSection 6.1.1, p. 105
NaN	Element-wise check for finitenessSection 6.1.1, p. 105
is.nan	Value for invalid numericsSection 6.1.2, p. 106
NA	Element-wise check for NaNSection 6.1.2, p. 107
is.na	Value for missing observationSection 6.1.3, p. 108
na.omit	Element-wise check for NA OR NaNSection 6.1.3, p. 109
NULL	Delete all NAs and NaNsSection 6.1.3, p. 110
is.null	Value for “empty”Section 6.1.4, p. 110
attributes	Check for NULLSection 6.1.4, p. 111
attr	List explicit attributesSection 6.2.1, p. 114
dimnames	Obtain specific attributeSection 6.2.1, p. 115
class	Get array dimension namesSection 6.2.1, p. 116
is.	Get object class (S3)Section 6.2.2, p. 117
as. -	Object-checking functionsSection 6.2.3, p. 120
-	Object-coercion functionsSection 6.2.4, p. 121

7

BASIC PLOTTING



One particularly popular feature of R is its incredibly flexible plotting tools for finding model visualization. This is what draws many to R in the first place. Mastering R's graphical functionality does require practice, but the fundamental concepts are straightforward.

In this chapter, I'll provide an overview of the plot function and some useful options for controlling the appearance of the final graph. Then I'll cover the basics of using ggplot2, a powerful library for visualizing data in R. This chapter will cover just the basics of plotting, and then you'll learn more about creating different types of statistical plots in Chapter 14, and about advanced plotting techniques in Part V.

7.1 Using plot with Coordinate Vectors

The easiest way to think about generating plots in R is to treat your screen as a blank, two-dimensional canvas. You can plot points and lines using x- and y-coordinates. On paper, these coordinates are usually represented with points written as a pair: (x value, y value). The R function plot, on the other hand, takes in two vectors—one vector of x locations and one vector of y locations—and opens a *graphics device* where it displays the result. If a

graphics device is already open, R's default behavior is to refresh the device, overwriting the current contents with the new plot.

For example, let's say you wanted to plot the points (1.1,2), (2,2.2), (3.5, -1.3), (3.9,0), and (4.2,0.2). In plot, you must provide the vector of x locations first, and the y locations second. Let's define these as foo and

bar,

```
foo <- c(1.1,2,3.5,3.9,4.2) R>
```

```
bar <- c(2,2.2,-1.3,0,0.2) R>
```

```
plot(foo,bar)
```

Figure 7-1 shows the resulting graphics device with the plot (I'll use this simple data set as a working example throughout this section).

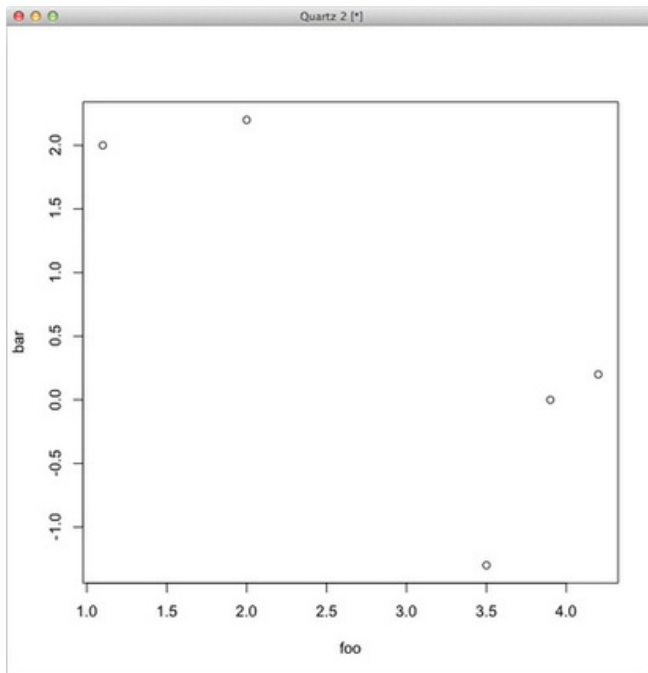


Figure 7-1: The five plotted points using R's default behavior

The x and y locations don't necessarily need to be specified as separate vectors. You can also supply coordinates in the form of a matrix, with the x values in the first column and the y values in the second column, or as a list. For example, setting up a matrix of the five points, the following code exactly reproduces Figure 7-1 (note the window pane will look slightly different depending on your operating system):

```
R> baz <- cbind(foo,bar)
```

```
R> baz
```

```
foo bar
```

```
[1,]1.1    2.0
[2,]2.0 2.2 [3,]
3.5      -1.3
[4,]3.9    0.0
[5,]4.2 0.2 R>
plot(baz)
```

The function `plot` is one of R's versatile *generic* functions. It works differently for different objects and allows users to define their own methods for handling objects (including user-defined object classes). Technically, the version of the `plot` command that you've just used is internally identified as `plot.default`. The help file `?plot.default` provides additional details on this *scatterplot* style of data visualization.

7.2 Graphical Parameters

There are a wide range of *graphical parameters* that can be supplied as arguments to the `plot` function (or other plotting functions, such as those in Section 7.3). These parameters invoke simple visual enhancements, like coloring the points and adding axis labels, and can also control technical aspects of the graphics device (Chapter 23 covers the latter in more detail). Some of the most commonly used graphical parameters are listed here; I'll briefly discuss each of these in turn in the following sections:

type Tells R how to plot the supplied coordinates (for example, as stand-alone points or joined by lines or both dots and lines).

main, xlab, ylab Options to include plot title, the horizontal axis label, and the vertical axis label, respectively.

col Color (or colors) to use for plotting points and lines.

pch Stands for *point character*. This selects which character to use for plotting individual points.

cex Stands for *character expansion*. This controls the size of plotted point characters.

lty Stands for *line type*. This specifies the type of line to use to connect the points (for example, solid, dotted, or dashed).

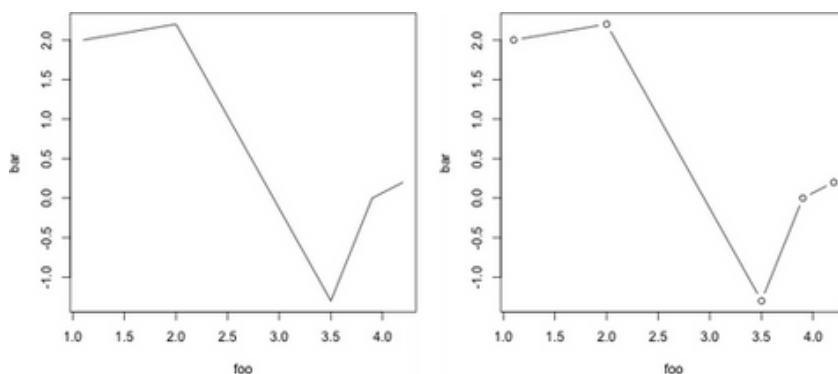
lwd Stands for *line width*. This controls the thickness of plotted lines.

xlim, ylim This provides limits for the horizontal range and vertical range (respectively) of the plotting region.

7.2.1 Automatic Plot Types

By default, the `plot` function will plot individual points, as shown in Figure 7-1. This is the default plot type, but other plot types will have a different appearance. To control the plot type, you can specify a single character-valued option for the argument `type`.

```
R> plot(foo,bar,type="l")
```



The default value for type is "p", which can be interpreted as "points only." Since you didn't specify anything different, this is what was used for the graph in Figure 7-1. In this last example, on the other hand, you've set type="l" (meaning "lines only"). Other options include "b" for both points *and* lines (shown in the right panel of Figure 7-2) and "o" for overplotting the points with lines (this eliminates the gaps between points and lines visible for type="b"). The option type="n" results in no points or lines plotted, creating an empty plot, which can be useful for complicated plots that must be constructed in steps.

By default, a basic plot won't have a main title, and its axes will be labeled with the names of the vectors being plotted. But a main title and more descriptive axis labels often make the plotted data easier to interpret. You can add these by supplying text as character strings to `main` for a title, `xlab` for the x-axis label, and `ylab` for the y-axis label. Note that these strings may include escape sequences (discussed in Section 4.2.3). The following code produces the plots in Figure 7-3:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="x axis label",
ylab="location y")
R> plot(foo,bar,type="b",main="My lovely plot\n title on two
lines",xlab="", ylab="")
```

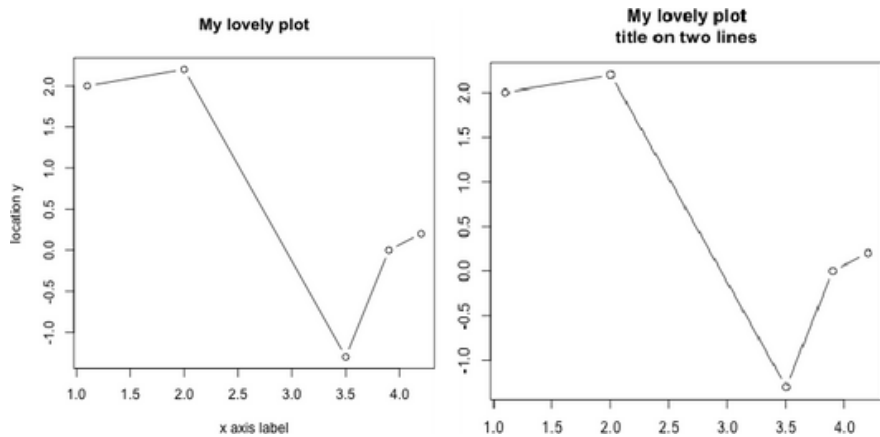


Figure 7-3: Two examples of plots with axis labels and titles

In the second plot, note how the new line escape sequence splits the title into two lines. In that plot, `xlab` and `ylab` are also set to the empty string `""` to prevent R from labeling the axes with the names of the `x` and `y` vectors.

7.2.3 Color

Adding color to a graph is far from just an aesthetic consideration. Color can make data much clearer—for example by distinguishing factor levels or emphasizing important numeric limits. You can set colors with the `col` parameter in a number of ways. The simplest options are to use an integer selector or a character string. There are a number of color string values recognized by R, which you can see by entering `colors()` at the prompt.

The default color is integer 1 or the character string `"black"`. The top row of Figure 7-4 shows two examples of colored graphs, created by the following code:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",col=2)
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",col="seagreen4")
```

There are eight possible integer values (shown in the leftmost plot of Figure 7-5) and around 650 character strings to specify color. But you aren't limited to these options since you can also specify colors using RGB (red, green, and blue) levels and by creating your own palettes. I'll talk more about the last two options in Chapter 25.

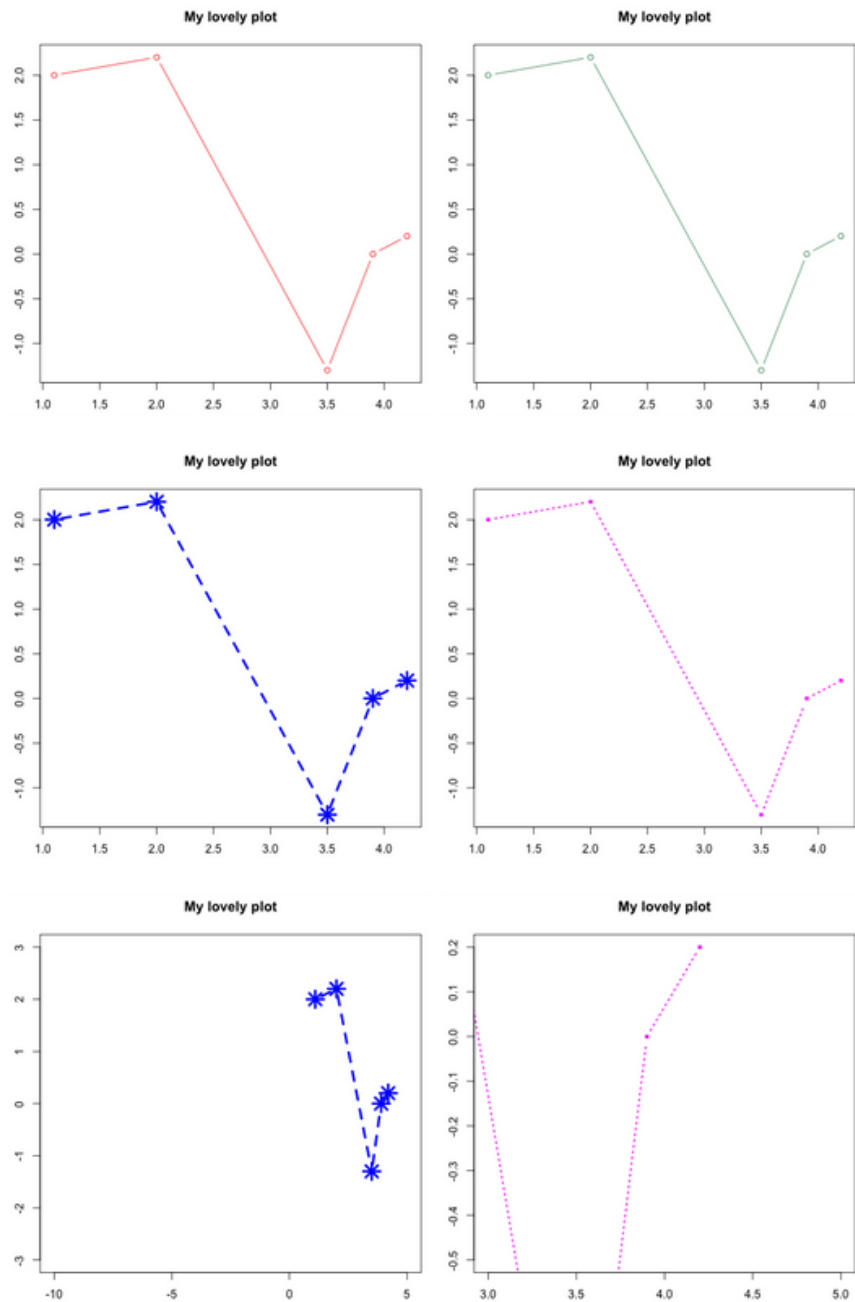


Figure 7-4: Experimenting with basic R plotting. Top row: Two examples of colored plots with `col=2` (left) and `col="seagreen4"` (right). Middle row: Two further examples making use of `pch`, `lty`, `cex`, and `lwd`. Bottom row: Setting plotting region limits `xlim=c(-10,5)`, `ylim=c(-3,3)` (left), and `xlim=c(3,5)`, `ylim=c(-0.5,0.2)` (right).

7.2.4 Line and Point Appearances

To alter the appearance of the plotted points you would use `pch`, and to alter the lines you would use `lty`. The `pch` parameter controls the character used to plot individual data points. You can specify a single character to

use for each point, or you can specify a value between 1 and 25 (inclusive). The symbols corresponding to each integer are shown in the middle plot

of Figure 7-5. The `lty` parameter, which affects the type of line drawn, can take the values 1 through 6. These options are shown in the rightmost plot of Figure 7-5.

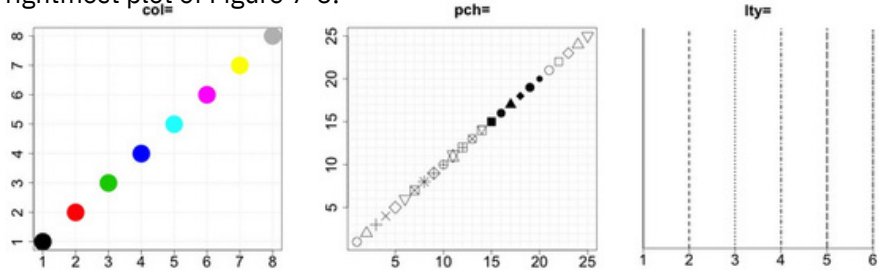


Figure 7-5: Some reference plots giving the results of possible integer options of `col` (left), `pch` (middle), and `lty` (right)

You can also control the size of plotted points using `cex` and the thickness of lines using `lwd`. The default size and thickness for both of these is 1. To request half-size points, for example, you'd specify `cex=0.5`. To double-thickness lines, use `lwd=2`.

The following two lines produce the two plots in the middle row of Figure 7-4, showing off `cex`, `lwd`, and `col`:

```
R> plot(foo, bar, type="b", main="My lovely plot", xlab="", ylab="",
      col=4, pch=8, lty=2, cex=2.3, lwd=3.3)
R> plot(foo, bar, type="b", main="My lovely plot", xlab="", ylab="",
      col=6, pch=15, lty=3, cex=0.7, lwd=2)
```

7.2.5 Plotting Region Limits

As you can see in the plots of `foo` and `bar`, by default R sets the range of each axis by using the range of the supplied `x` and `y` values (plus a small constant to pad a little area around the outermost points). But you might need more space than this to, for example, annotate individual points, add a legend, or plot additional points that fall outside the original ranges (as you'll see in Section 7.3). You can set custom plotting area limits using `xlim` and `ylim`. Both parameters require a numeric vector of length 2, provided as `c(lower, upper)`.

Consider the plots in the bottom row of Figure 7-4, created with the following two commands:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
col=4,pch=8,lty=2,cex=2.3,lwd=3.3,xlim=c(-10,5),ylim=c(-3,3))
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
col=6,pch=15,lty=3,cex=0.7,lwd=2,xlim=c(3,5),ylim=c(-0.5,0.2))
```

These plots are exactly the same as the two in the middle row, except for one important difference. In the bottom-left plot of Figure 7-4, the x- and y-axes are set to be much wider than the observed data, and the plot on the right restricts the plotting window so that only a portion of the data is displayed.

7.3 Adding Points, Lines, and Text to an Existing Plot

Generally speaking, each call to `plot` will refresh the active graphics device for a new plotting region. But this is not always desired—to build more complicated plots, it’s easiest to start with an empty plotting region and progressively add any required points, lines, text, and legends to this canvas. Here are some useful, ready-to-use functions in R that will add to a plot without refreshing or clearing the window:

points Adds points

lines, abline, segments Adds lines

text Writes text

arrows Adds arrows

legend Adds a legend

The syntax for calling and setting parameters for these functions is the same as `plot`. The best way to see how these work is through an extended example, which I’ll base on some hypothetical data made up of 20 (x, y) locations.

```
R> x <- 1:20
R> y <- c(-1.49,3.37,2.59,-2.78,-3.94,-0.92,6.43,8.51,3.41,-8.23,
-12.01,-6.58,2.87,14.12,9.63,-4.58,-14.78,-11.67,1.17,15.62)
```

Using these data, you’ll build up the plot shown in Figure 7-6 (note that you may need to manually enlarge your graphics device and replot to ensure the legend doesn’t overlap other features of the image). It’s worth remembering a generally accepted rule in plotting: “keep it clear and simple.” Figure 7-6 is an exception for the sake of demonstrating the R commands used.

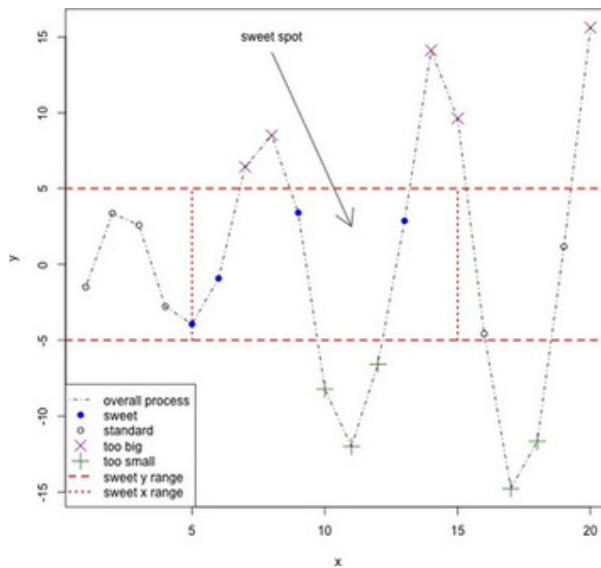


Figure 7-6: An elaborate final plot of some hypothetical data

In Figure 7-6, the data points will be plotted differently according to their x and y locations, depending on their relation to the “sweet spot” pointed out in the figure. Points with a y value greater than 5 are marked with a purple x; points with a y value less than -5 are marked with a green +. Points between these two y values but still outside of the sweet spot are marked with a blue .

Points with an x value between 5 and 15 and with y between

-5 and 5) are marked as a blue . Red horizontal and vertical lines delineate the sweet spot, which is labeled with an arrow, and there’s also a legend.

Ten lines of code were used to build this plot in its entirety (plus one additional line to add the legend). The plot, as it looks at each step, is given in Figure 7-7. The lines of code are detailed next.

1. The first step is to create the empty plotting region where you can add points and draw lines. This first line tells R to plot the data in x and y, though the option type is set to “n”. As mentioned in Section 7.2, this opens or refreshes the graphics device and sets the axes to the appropriate scale (width, height, point size, line width, and axes), but it doesn’t plot any

2. A function is a simple way to add straight lines spanning a plot. The line (or lines) can be specified with *slope* and *intercept* values (see the later discussions on regression in Chapter 20). You can also simply add horizontal or vertical lines. This line of code adds two separate horizontal lines, one at y = 5 and the other at y = -5, using `h=c(-5,5)`. The three parameters (covered in Section 7.2) make these

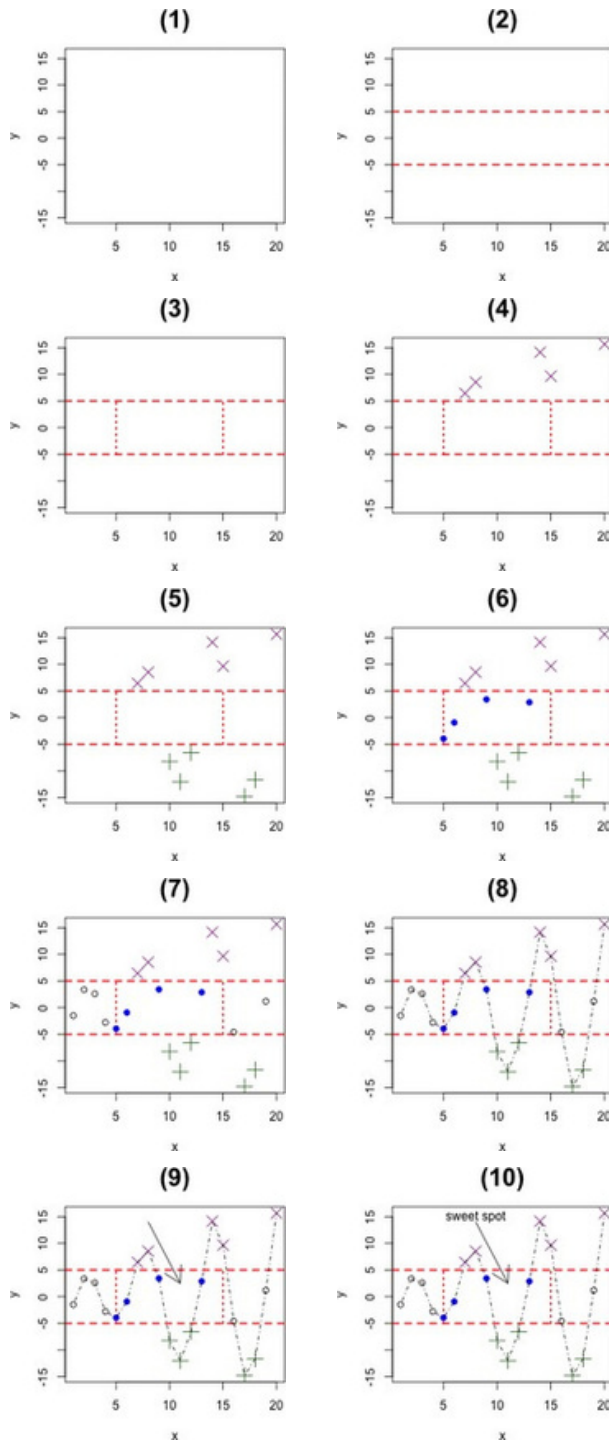


Figure 7-7: Building the final plot given in Figure 7-6. The plots (1) through (10) correspond to the itemized lines of code in the text.

two lines red, dashed, and double-thickness. For vertical lines, you could use `abline(v=c(-5,5),lty=2,lwd=2)`, which would have drawn the lines at $x = -5$ and $x = 5$.

```
R> abline(v=c(-5,5),col="red",lty=2,lwd=2)
```

3. The third line of code adds shorter vertical lines between the horizontal ones drawn in step 2 to form a box. For this you use `segments`, not `abline`, since you don't want these lines to span the entire plotting region. The `segments` command takes a "from" coordinate (given as x_0 and y_0) and a "to" coordinate (as x_1 and y_1) and draws the corresponding line. The vector-oriented behavior of R matches up the two sets of "from" and "to" coordinates. Both lines are red and dotted and have double-thickness. (You could also supply vectors of length 2 to these parameters, in which case the first segment would use the first parameter value and the second segment would use the second value.)

```
R> segments(x0=c(5,15),y0=c(-5,-5),x1=c(5,15),y1=c(5,5),col="red",lty=3,
lwd=2)
```

4. As step 4, you use `points` to begin adding specific coordinates from the data set to the plot. Just like `abline`, it takes two vectors of equal lengths with x and y values. In this case, you want points plotted differently according to their location, so you use logical vector subsetting (see Section 4.1.5) to identify and extract elements of x and y where the y value is greater than or equal to 5. These (and only these) points are added as purple $\times$ symbols and are enlarged by a factor of 2 with `cex`.

```
R> points(x[y>=5],y[y>=5],pch=4,col="darkmagenta",cex=2)
```

5. The fifth line of code is much like the fourth; this time it extracts the coordinates where y values are less than or equal to -5. A $+$ point character is used, and you set the color to dark green.

```
R> points(x[y<=-5],y[y<=-5],pch=3,col="darkgreen",cex=2)
```

6. The sixth step adds the blue "sweet spot" points, which are identified with `(x>=5&x<=15)&(y>-5&y<5)`. This slightly more complicated set of conditions extracts the points whose x location lies between 5 and 15 (inclusive) AND whose y location lies between -5 and 5 (exclusive). Note that this line uses the "short" form of the logical operator throughout since you want element-wise comparisons here (see Section 4.1.3).

```
R> points(x[(x>=5&x<=15)&(y>-5&y<5)],y[(x>=5&x<=15)&(y>-5&y<5)],pch=19,
col="blue")
```

7. This next command identifies the remaining points in the data set (with an x value that is either less than 5 OR greater than 15 AND a y value between -5 and 5). No graphical parameters are specified, so these points are plotted with the default black.

```
R> points(x[(x<5|x>15)&(y>-5&y<5)],y[(x<5|x>15)&(y>-5&y<5)])
```

8. To draw lines connecting the coordinates in `x` and `y`, you use `lines`. Here you've also set `lty=4`, which draws a dash-dot-dash style line.

```
R> lines(x,y,lty=4)
```

9. The ninth line of code adds the arrow pointing to the sweet spot. The function `arrows` is used just like `segments`, where you provide a “from” coordinate (`x0`, `y0`) and a “to” coordinate (`x1`, `y1`). By default, the head of the arrow is located at the “to” coordinate, though this (and other options such as the angle and length of the head) can be altered using optional arguments described in the help file `?arrows`.

```
R> arrows(x0=8,y0=14,x1=11,y1=2.5)
```

10. The tenth line prints a label on the plot at the top of the arrow. As the default behavior of `text`, the strings supplied as labels are *centered* on the coordinates provided with the arguments `x` and `y`.

```
R> text(x=8,y=15,labels="sweet spot")
```

As a finishing touch, you can add the legend with the `legend` function, which gives you the final product shown in Figure 7-6.

```
legend("bottomleft",
legend=c("overall process","sweet","standard",
"too big","too small","sweet y range","sweet x range"),
pch=c(NA,19,1,4,3,NA,NA),lty=c(4,NA,NA,NA,NA,2,3),
col=c("black","blue","black","darkmagenta","darkgreen","red","red"),
lwd=c(1,NA,NA,NA,NA,2,2),pt.cex=c(NA,1,1,2,2,NA,NA))
```

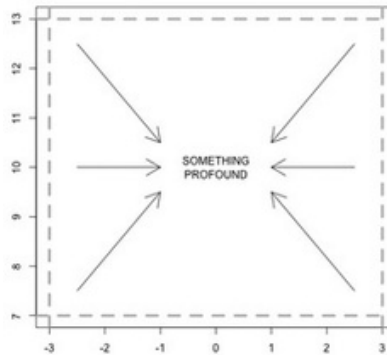
The first argument sets where the legend should be placed. There are various ways to do this (including setting exact `x`- and `y`-coordinates), but it often suffices to pick a corner using one of the four following character strings: `"topleft"`, `"topright"`, `"bottomleft"`, or `"bottomright"`. Next you supply the labels as a vector of character strings to the `legend` argument. Then you need to supply the remaining argument values in vectors of the same length so that the right elements match up with each label.

For example, for the first label (“overall process”), you want a line of type 4 with default thickness and color. So, in the first positions of the remaining argument vectors, you set `pch=NA`, `lty=4`, `col="black"`, `lwd=1`, and `pt.cex=NA` (all of these are default values, except for `lty`). Here, `pt.cex` simply refers to the `cex` parameter when calling points (using just `cex` in `legend` would expand the text used, not the points).

Note that you have to fill in some elements in these vectors with `NA` when you don’t want to set the corresponding graphical parameter. This is just to preserve the equal lengths of the vectors supplied so R can track which parameter values correspond to each particular reference. As you work through this book, you’ll see plenty more examples using `legend`.

Exercise 7.1

- a. As closely as you can, re-create the following plot:



- b. With the following data, create a plot of weight on the x-axis and height on the y-axis. Use different point characters or colors to distinguish between males and females and provide a matching legend. Label the axes and give the plot a title.

Weight	Height	Sex
(kg) 55	(cm) 161	female
85	185	male
75	174	male
42	154	female
93	188	male
63	178	male
58	170	female
75	167	male
89	181	male
67	178	female

7.4 The ggplot2 Package

This chapter so far has shown off R's built-in graphical tools (often called *base R graphics* or *traditional R graphics*). Now, let's look at another important suite of graphical tools: ggplot2, a prominent contributed package by Hadley Wickham (2009). Available on CRAN like any other contributed package, ggplot2 offers particularly powerful alternatives to the standard plotting procedures in R. The *gg* stands for *grammar of graphics*—a particular approach to graphical production described by Wilkinson (2005). In following this approach, ggplot2 standardizes the production of different plot and graph types, streamlines some of the more fiddly aspects of adding to existing plots (such as including a legend), and lets you build plots by defining and manipulating *layers*. For the moment, let's see the elementary behavior of ggplot2

using the same simple examples in Sections 7.1–7.3. You’ll get familiar with the basic plotting function `qplot` and how it differs from the generic plot function used earlier. I’ll return to the topic of `ggplot2` when I cover statistical plots in Chapter 14, and you’ll investigate even more advanced abilities in Chapter 24.

7.4.1 A Quick Plot with `qplot`

First, you must install the `ggplot2` package by downloading it manually or simply entering `install.packages("ggplot2")` at the prompt (see Section A.2.3). Then, load the package with the following:

```
R> library("ggplot2")
```

Now, let’s go back to the five data points originally stored in Section 7.1 and

```
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
```

You can produce `ggplot2`’s version of Figure 7-1 using its “quick plot” function.

```
R> qplot(foo,bar)
```

The result is shown in the left image of Figure 7-8. There are some obvious differences between this image and the one produced using `plot`, but the basic syntax of `qplot` is the same as earlier. The first two arguments passed to `qplot` are vectors of equal length, with the x-coordinates in `foo` supplied first, followed by the y-coordinates in `bar`.

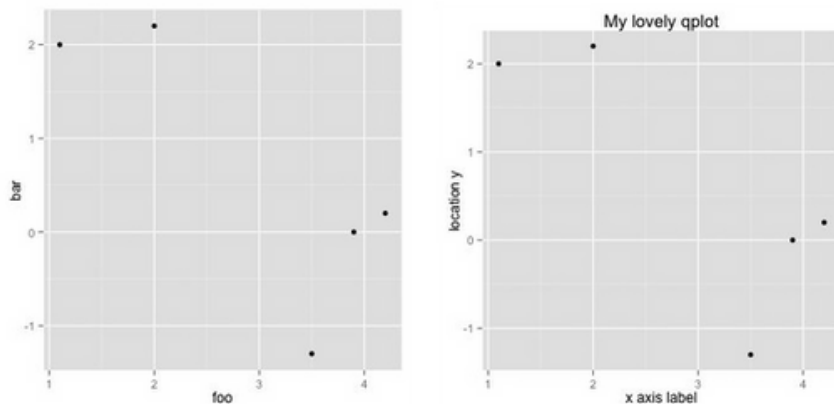


Figure 7-8: Five plotted points using `ggplot2`’s default behavior for the `qplot` function (left) and with title and axis labels added (right)

Adding a title and axis labels also uses the same arguments you already saw with `qplot()` in Section 7.2.

```
R> qplot(foo,bar,main="My lovely qplot",xlab="x axis label",ylab="location y")
```

This produces the right panel of Figure 7-8.

Underneath this basic similarity in syntax, though, there is a fundamental difference between how `ggplot2` and base R graphics create plots. Constructing plots using the built-in graphics tools is essentially a live, step- by-step process. This was particularly noticeable in Section 7.3, where you treated the graphics device as an active canvas where you added points,

lines, and other features one by one. By contrast, `ggplot2` plots are stored as objects, which means they have an underlying, static representation until you *change* the object—what you essentially visualize with `qplot` is the printed object at any given time. To highlight this, enter the following code:

```
R> baz <- plot(foo,bar) R>
baz
NULL
R> qux <- qplot(foo,bar)
R> qux
```

The first assignment uses the built-in plot function. When you run that line of code, the plot in Figure 7-1 pops up. Since nothing is actually stored in the workspace, printing the supposed object `baz` yields the empty `NULL` value. On the other hand, it makes sense to store the `qplot` content (stored as the object `qux` here). This time, when you perform the assignment, no plot is displayed. The graphic, which matches Figure 7-8, is displayed only when you enter `qux` at the prompt, which invokes the print method for that object. This may seem like a minor point, but the fact that you can save a plot this way before displaying it opens up new ways to modify or enhance plots before displaying them (as you will see in a moment), and it can be a distinct advantage over base R graphics.

7.4.2 Setting Appearance Constants with Geoms

To add and customize points and lines in a `ggplot2` graphic, you alter the object itself, rather than using a long list of arguments or secondary functions executed separately (such as `points` or `lines`). You can modify the object using `ggplot2`'s convenient suite of *geometric modifiers*, known as *geoms*. Let's say you want to connect the five points in `foo` and `bar` with a line, just as you did in Section 7.1. You can first create a blank plot object and then use geometric modifiers on it like this:

```
R> qplot(foo,bar,geom="blank") + geom_point() + geom_line()
```

The resulting plot is shown on the left of Figure 7-9. In the first call to `qplot`, you create an empty plot object by setting the initial geometric

modifier as `geom="blank"` (if you displayed this plot, you would just see the gray background and the axes). Then you layer on the two other geoms as `geom_point()` and `geom_line()`. As indicated by the parentheses, these geoms are functions that result in their own specialized objects. You can add geoms to the `qplot` object using the `+` operator. Here, you haven't supplied any arguments to either geom, which means they'll operate on the same data originally supplied to `qplot` (`foo` and `bar`) and they'll stick to the default settings for any other features, such as color or point/line type. You can control those features by specifying optional arguments, as shown here:

```
R> qplot(foo,bar,geom="blank") + geom_point(size=3,shape=6,color="blue")  
+ geom_line(color="red",linetype=2)
```

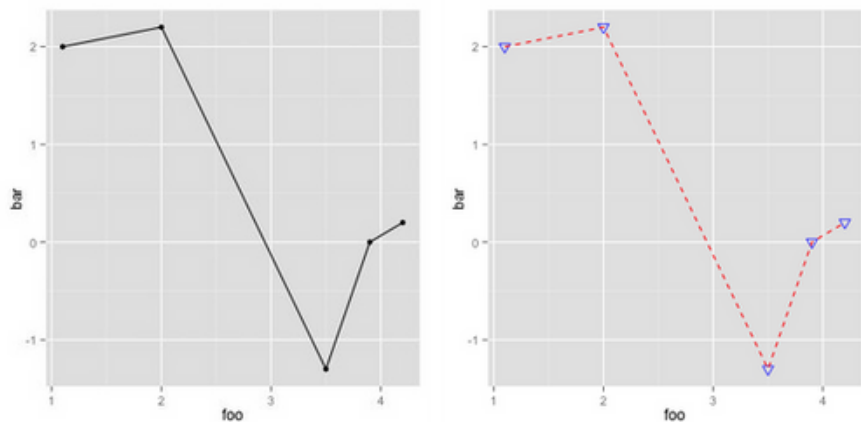


Figure 7-9: Two simple plots that use geometric modifiers to alter the appearance of a `qplot` object. Left: Adding points and lines using default settings. Right: Using the geoms to affect point character, size, and color, and line type and color.

Note that some of `ggplot2`'s argument names used here for things such as point characters and size (`shape` and `size`) are different from the base R graphics arguments (`pch` and `cex`). But `ggplot2` is actually compatible with many of the common graphical parameters used in R's standard plot function, so you can use those arguments here too if you prefer. For instance, setting `cex=3` and `pch=6` in `geom_point` in this example would result in the same image.

The object-oriented nature of `ggplot2` graphics means tweaking a plot or experimenting with different visual features no longer requires you to rerun every plotting command each time you change something. This is facilitated by geoms. Say you like the line type used on the right side of Figure 7-9 but want a different point character. To experiment, you could first store the `qplot` object you created earlier and then use `geom_point` with that object to try different point styles.

```
R> myqplot <- qplot(foo,bar,geom="blank") + geom_line(color="red",linetype=2)
R> myqplot + geom_point(size=3,shape=3,color="blue")
R> myqplot + geom_point(size=3,shape=7,color="blue")
```

The first line stores the original plot in `myqplot`, and the next lines call `myqplot` with different point shapes. The second and third lines produce the graphics on the left and right of Figure 7-10, respectively.

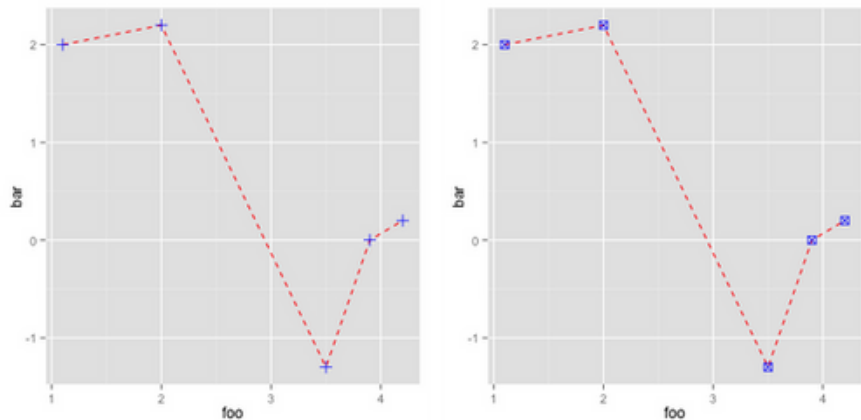


Figure 7-10: Using the object-oriented nature of `ggplot2` graphics to experiment with different point characters

There are a number of geometric modifiers that can be called using a function name beginning with `geom_` in `ggplot2`. To obtain a list, simply ensure the package is loaded and enter `??"geom_"` as a help search at the prompt.

7.4.3 Aesthetic Mapping with Geoms

Geoms and `ggplot2` also provide efficient, automated ways to apply different styles to different subsets of a plot. If you split a data set into categories using a factor object, `ggplot2` can automatically apply particular styles to different categories. In `ggplot2`'s documentation, the factor that holds these categories is called a *variable*, which `ggplot2` can *map* to *aesthetic* values. This gets rid of much of the effort that goes into isolating subsets of data and plotting them separately using base R graphics (as you did in Section 7.3). All this is best illustrated with an example. Let's return to the 20 observations you manually plotted, step-by-step, to produce the elaborate plot in Figure 7-6.

```
R> x <- 1:20
R> y <- c(-1.49,3.37,2.59,-2.78,-3.94,-0.92,6.43,8.51,3.41,-8.23,
        -12.01,-6.58,2.87,14.12,9.63,-4.58,-14.78,-11.67,1.17,15.62)
```

In Section 7.3, you defined several categories that classified each observation as either “standard,” “sweet,” “too big,” or “too small” based on their x and y values. Using those same classification rules, let’s explicitly define a factor to correspond to and .

```
R> ptype <- rep(NA,length(x=x))
R> ptype[y>=5] <- "too_big"
R> ptype[y<=-5] <- "too_small"
R> ptype[(x>=5&x<=15)&(y>-5&y<5)] <- "sweet"
R> ptype[(x<5|x>15)&(y>-5&y<5)] <- "standard"
R> ptype <- factor(x=ptype)
R> ptype
[1]standard standard standard standard sweet sweet
      [8]too_big sweet too_smalltoo_smalltoosweetsweet      too_bi
[15]too_big standard too_smalltoosmallstandard too_big      g
Levels: standard sweet too_big too_small      too_bi
g
```

Now you have a factor with 20 values sorted into four levels. You’ll use this factor to tell qplot how to map your aesthetics. Here’s a simple way to do that:

```
R> qplot(x,y,color=ptype,shape=ptype)
```

This single line of code produces the left plot in Figure 7-11, which separates the four categories by color and point character and even provides a legend. This was all done by the aesthetic mapping in the call to qplot, where you set color and shape to be mapped to the ptype variable.

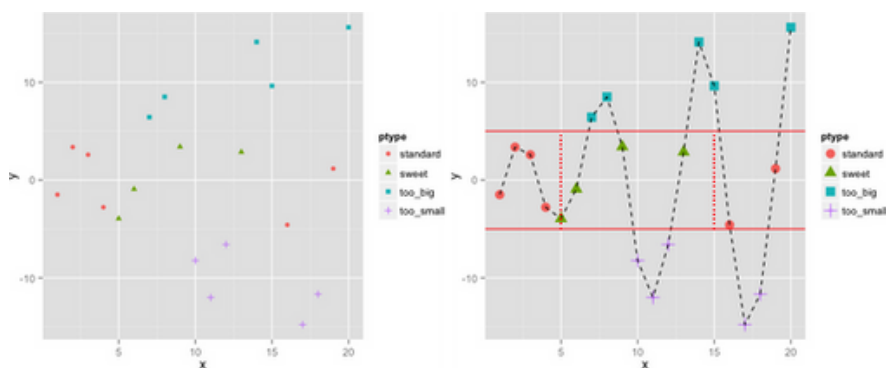


Figure 7-11: Demonstration of aesthetic mapping using qplot and geoms in ggplot2. Left: The initial call to qplot, which maps point character and color using ptype. Right: Augmenting the left plot using various geoms to override the default mappings.

Now, let's replot these data using the same `qplot` object along with a suite of `geom` modifications in order to get something more like Figure 7-6. Executing the following produces the plot on the right of Figure 7-11:

```
R> qplot(x,y,color=ptype,shape=ptype) + geom_point(size=4) +
  geom_line(mapping=aes(group=1),color="black",lty=2) +
  geom_hline(mapping=aes(yintercept=c(-5,5)),color="red") +
    geom_segment(mapping=aes(x=5,y=-5,xend=5,yend=5),color="red",lty=3) +
    geom_segment(mapping=aes(x=15,y=-5,xend=15,yend=5),color="red",lty=3)
```

`geom_point(size=4)` In the first line, you add to increase the size of all the points on the graph. In the lines that follow, you add a line connecting all the points, plus horizontal and vertical lines to mark out the sweet spot. For those last four lines, you have to use `aes` to set alternate aesthetic mappings for the point categories. Let's look a little closer at what's going on there. Since you used `ptype` for aesthetic mapping in the initial call to `qplot`, by default all other geoms will be mapped to each category in the same way, *unless* you override that default mapping with `aes`. For example, when you call `geom_line` to connect all the points, if you were to stick with the default mapping to `ptype` instead of including `mapping=aes(group=1)`, this geom would draw lines connecting points within each category. You would see four separate dashed lines—one connecting all “standard” points, another connecting all “sweet” points, and so on. But that's not what you want here; you want a line that connects all of the points, from left to right. So, you tell `geom_line` to treat all the observations as one group by entering `aes(group=1)`. After that, you use the `geom_hline` function to draw horizontal lines at $y = -5$ and $y = 5$ using its `yintercept` argument, again passed to `aes` to redefine that geom's mapping. In this case, you need to redefine the mapping to operate on the vector `c(-5,5)`, rather than using the observed data in `x` and `y`. Similarly, you end by using `geom_segment` to draw the two vertical dotted line segments. `geom_segment` operates much like segments—you redefine the mapping based on a “from” coordinate (arguments `x` and `y`) and a “to” coordinate (`xend` and `yend` here). Since the first geom, `geom_point(size=4)`, sets a constant enlarged size for every plotted point, it doesn't matter how the geom is mapped because it simply makes a uniform change to each point. Plotting in R, from base graphics to contributed packages like `ggplot2`, stays true to the nature of the language. The element-wise matching allows you to create intricate plots with a handful of straightforward and intuitive functions. Once you display a plot, you can save it to the hard drive by selecting the graphics device and choosing File → Save. However, you can also write plots to a file directly, as you'll see momentarily in Section 8.3. The graphical capabilities explored in this section are merely the tip of the iceberg, and you'll continue to use data visualizations from this point onward.

Exercise 7.2

In Exercise 7.1 (b), you used base R graphics to plot some weight and height data, distinguishing males and females using different points or colors. Repeat this task using ggplot2.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
plot	Create/display base R plot	Section 7.1, p. 128
typ	Set plot type	Section 7.2.1, p. 130
main, xlab, ylab	Set axis labels	Section 7.2.2, p. 130
col, cex	Set point/line color	Section 7.2.3, p. 131
pch	Set point type/size	Section 7.2.4, p. 133
lty, lwd	Set line type/width	Section 7.2.4, p. 133
xlim, ylim	Set plot region limits	Section 7.2.5, p. 134
abline	Add vertical/horizontal line	Section 7.3, p. 137
segments	Add specific line segments	Section 7.3, p. 137
points	Add points	Section 7.3, p. 137
lines	Add lines following coords	Section 7.3, p. 138
arrows	Add arrows	Section 7.3, p. 138
text	Add text	Section 7.3, p. 138
legend	Add/control legend	Section 7.3, p. 138
qplot	Create ggplot2 “quick plot”	Section 7.4.1, p. 140
geom_point	Add points geom	Section 7.4.2, p. 141
geom_line	Add lines geom	Section 7.4.2, p. 141
size	Set geom constants	Section 7.4.2, p. 142
linetype	Set geom line type	Section 7.4.2, p. 142
color, hline	Geom aesthetic mapping	Section 7.4.3, p. 145
geom_segment	Add horizontal lines geom	Section 7.4.3, p. 145
aes	Add line segments geom	Section 7.4.3, p. 145

8

READING AND WRITING FILES



Now I'll cover one more fundamental aspect of working with R: loading and saving data in an active workspace by reading and writing files. Typically, to work with a large data set, you'll need to read in the data from an external file, whether it's stored as plain text, in a spreadsheet file, or on a website. R provides command line

functions you can use to import these data sets, usually as a data frame object. You can also export data frames from R by writing a new file on your computer, plus you can save any plots you create as image files. In this chapter, I'll go over some useful command-based read and write operations for importing and exporting data.

8.1 R-ReadyDataSets

First, let's take a brief look at some of the data sets that are built into the software or are part of user-contributed packages. These data sets are useful samples to practice with and to experiment with functionality. Enter `data()` at the prompt to bring up a window listing these ready-to-use data sets along with a one-line description. These data sets are organized in alphabetical order by name and grouped by package (the exact list that

appears will depend on what contributed packages have been installed from CRAN; see Section A.2).

8.1.1 Built-in Data Sets

There are a number of data sets contained within the built-in, automatically downloaded seed package. To see a summary of the data sets contained in the package, you can use the function `library(help="datasets")`:

R-ready data sets have a corresponding help file where you can find important details about the data and how it's organized. For example, one of the built-in data sets is named `ChickWeight`. If you enter `?ChickWeight` at the prompt, you'll see the window in Figure 8-1.

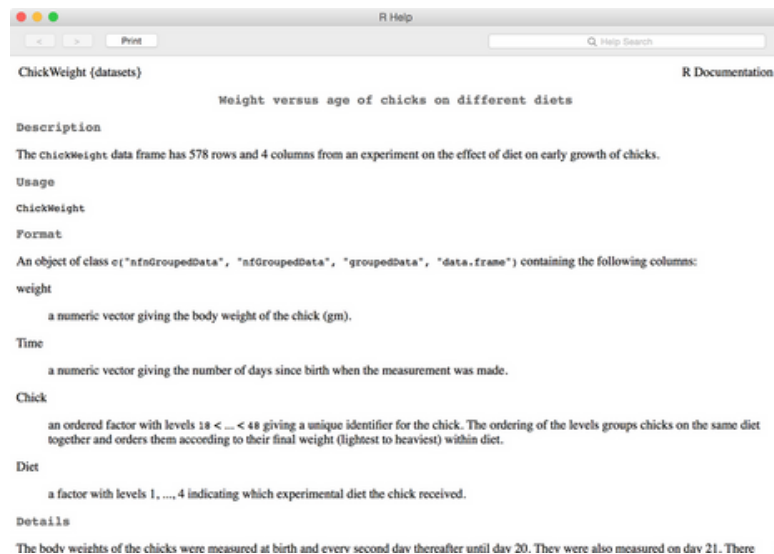


Figure 8-1: The help file for the `ChickWeight` data set

As you can see, this file explains the variables and their values; it notes that the data are stored in a data frame with 578 rows and 4 columns. Since the objects in datasets are built in, all you have to do to access `ChickWeight` is enter its name at the prompt. Let's look at the first 15 records.

```
R> ChickWeight[1:15,]
      weight Time Chick Diet
1      142    0    18    1
2      251    2    25    1
3      359    4    35    1
4      464    6    46    1
5      576    8    57    1
```

```

6      93      1      1      1
7      10      0      1      1
8       6      1      1      1
9      12      2      1      1
10     5       1      1      1
11     14      4      1      1
12     9       1      1      1
13     17      6      2      1
14     1       1      2      1
15     19      8      2      1
      9       2
      20      0

```

You can treat this data set like any other data frame you’ve created in R—note the use of `[1:15,]` to access the desired rows from such an object, as detailed in Section 5.2.1.

```

58      2
      4

```

8.1.2 Contributed Data Sets

There are many more R-ready data sets that come as part of contributed packages. To access them, first install and load the relevant package. Consider the data set `ice.river`, which is in the contributed package `tseries` by Trapletti and Hornik (2013). First, you have to install the package, which you can do by running the line `install.packages("tseries")` at the prompt. Then, to access the components of the package, load it using :

```
R> library("tseries")
```

```
'tseries' version: 0.10-32
```

```
'tseries' is a package for time series analysis and computational finance.
```

See `library(help="tseries")` for details.

```
library(help="tseries")
```

Now you can enter to see the list of data sets in this package, and you can enter to find more details about the data set you want to work with here. The help file describes a

“time series object” comprised of river flow, precipitation, and temperature measurements—data initially reported in Tong (1990). To access this object itself, you must explicitly load it using the data function. Then you can work with

in your workspace as usual. Here are the first five records:

```
R> data(ice.river)
```

```
R> ice.river[1:5,]
```

```

      flow.vat flow.jok prec temp
[1,] 16.10 30.2 8.1 0.9
[2,] 19.20 29.0 4.4 1.6
[3,] 14.50 28.4 7.0 0.1
[4,] 11.00 27.8 0.0 0.6
[5,] 13.60 27.8 0.0 2.0

```

The availability and convenience of these R-ready data sets make it easy to test code, and I'll use them in subsequent chapters for demonstrations. To analyze your own data, however, you'll often have to import them from some external file. Let's see how to do that.

8.2 Reading in External Data Files

R has a variety of functions for reading characters from stored files and making sense of them. You'll look at how to read *table-format* files, which are among the easiest for R to read and import.

8.2.1 The Table Format

Table-format files are best thought of as plain-text files with three key features that fully define how R should read the data.

Header If a header is present, it's always the first line of the file. This optional feature is used to provide names for each column of data. When importing a file into R, you need to tell the software whether a header is present so that it knows whether to treat the first line as variable names or, alternatively, observed data values.

Delimiter The all-important *delimiter* is a character used to separate the entries in each line. The delimiter character cannot be used for anything else in the file. This tells R when a specific entry begins and ends (in other words, its exact position in the table).

Missing value This is another unique character string used exclusively to denote a missing value. When reading the file, R will turn these entries into the form it recognizes: NA.

Typically, these files have a *.txt* extension (highlighting the plain-text style) or *.csv* (for *comma-separated values*). Let's try an example, using a variation on the data frame *mydata* as defined at the end of Section 5.2.2. Figure 8-2 shows an appropriate table-format file called *mydatafile.txt*, which has the data from that data frame with a few values now marked as missing. This data file can be found on the book's website at <https://www.nostarch.com/bookofr/>, or you can create it yourself from Figure 8-2 using a text editor.



Figure 8-2: A plain-text table-format file

Note that the first line is the header, the values are delimited with a single space, and missing values are denoted with an asterisk (*). Also, note that each new record is required to start on a new line. Suppose you're handed this plain-text file for data analysis in R. The ready-to-use command `read.table` imports table-format files, producing a data frame object, as follows:

```
R> read.table(file="/Users/tdavies/mydatafile.txt",
             header=TRUE, sep=" ", na.strings="",
             stringsAsFactors=FALSE)
R> mydatafile
person age sex funny age.mon
1 Peter NA M High 504
2 Lois 40 F <NA> 480
3 Meg 17 F Low 204
4 Chris 14 M Med 168
5 Stewie 1 M High NA
6 Brian NA M Med NA
```

`read.table` takes a character string with the filename and folder location (using forward slashes), is a logical value telling R whether the file has a header (in this case), takes a character string providing the delimiter (a single space, in this case), and a character string providing the characters used to denote missing values (in this case).

If you're reading in multiple files and don't want to type the entire folder location each time, it's possible to first set your working directory via `setwd` (Section 1.2.3) and then simply use the filename and its extension as the character string supplied to the file argument. However, both approaches require you to know exactly where your file is located when you're working at the R prompt. Fortunately, R possesses some useful additional tools should you forget your file's precise location. You can view textual output of the contents of any folder by using `list.files`. The following example betrays the messiness of my local user directory.

```
R> list.files("/Users/tdavies")
[1] "bands-SCHIST1L200.txt" "Brass"
[4] "Documents" "DOSGames"
[7] "Dropbox" "Exercise2-20Data.txt"
[10] "iCloud" "Library"
[13] "Movies" "Music"
[16] "OneDrive" "peritonitis.sav"
[19] "Personal9414" "Pictures"
[22] "Research" "Rintro.tex"
[25] "Rstartup.R" "spreadsheetfile.csv"
[28] "TakeHome_template.tex" "WISE-
P2L" [31] "WISE-SCHIST1L200.txt"
      "Desktop"
      "Downloads"
      "Google Drive"
      "log.txt"
      "mydatafile.txt"
      "peritonitis.txt"
      "Public"
      "Rprofile.txt"
      "spreadsheetfile.xlsx"
      "WISE-P2S.txt"
```

One important feature to note here, though, is that it can be difficult to distinguish between files and folders. Files will typically have an extension, and folders won't; however, WISE-P2L is a file that happens to have no extension and looks no different from any of the listed folders. You can also find files interactively from R. The `file.choose()` command opens your filesystem viewer directly from the R prompt—just as any other program does when you want to open something. Then, you can navigate to the folder of interest, and after you select your file (see Figure 8-3), only a character string is returned.

```
R> file.choose()
[1] "/Users/tdavies/mydatafile.txt"
```

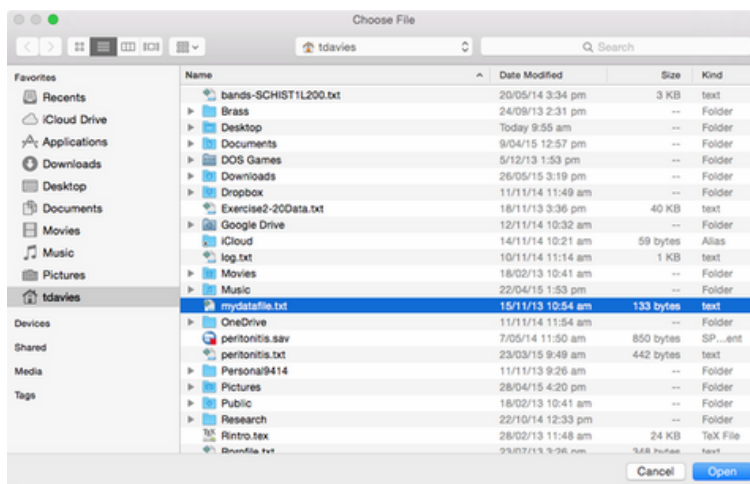


Figure 8-3: My local file navigator opened as the result of a call to `file.choose()`. When the file of interest is opened, the R command returns the full file path to that file as a character string.

This command is particularly useful, as it returns the character string of the directory in precisely the format that's required for a command such as `read.table()`. So, calling the following line and selecting `mydatafile.txt`, as in Figure 8-3, will produce an identical result to the explicit use of the file path in `shownearlier()`:

```
R> mydatafile <- read.table(file=file.choose(), header=TRUE, sep=" ",
                             na.strings="*", stringsAsFactors=FALSE)
```

If your file has been successfully loaded, you should be returned to the R prompt without receiving any error messages. You can check this with a call to `mydatafile`, which should return the data frame. When importing data into data frames, keep in mind the difference between character string observations and factor observations. No factor attribute information is

stored in the plain-text file, but `read.table` will convert non-numeric values into factors by default. Here, you want to keep some of your data saved as strings, so set `stringsAsFactors=FALSE`, which prevents R from treating all non-numeric elements as factors. This way, `person`, `sex`, and `funny` are all stored as character strings.

You can then overwrite `sex` and `funny` with factor versions of themselves if you want them as that data type.

```
R> mydatafile$sex <- as.factor(mydatafile$sex)
R> mydatafile$funny <- factor(x=mydatafile$funny,levels=c("Low","Med","High"))
```

8.2.2 Spreadsheet Workbooks

Next, let's examine some ubiquitous spreadsheet software file formats. The standard file format for Microsoft Office Excel is `.xls` or `.xlsx`. In general, these files are not directly compatible with R. There are some contributed package functions that attempt to bridge this gap—see, for example, `gdata` by Warnes et al. (2014) or `XLConnect` by Mirai Solutions GmbH (2014)—but it's generally preferable to first export the spreadsheet file to a table format, such as CSV. Consider the hypothetical data from Exercise 7.1 (b), which has been stored in an Excel file called *spreadsheetfile.xlsx*, shown in Figure 8-4.

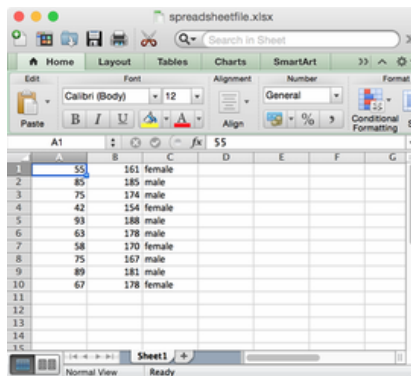


Figure 8-4: A spreadsheet file of the data from Exercise 7.1 (b)

To read this spreadsheet with R, you should first convert it to a table format. In Excel, File

→ Save As... provides a wealth of options. Save the spreadsheet as a comma-separated file, called *spreadsheet.csv*. R has a shortcut version of `read.table` for these files.

```
R> spread <- read.csv(file="/Users/tdavies/spreadsheetfile.csv",
header=FALSE,stringsAsFactors=TRUE)
R> spread
V1 V2 V3
1 55 161 female
```

```

2 85185 male
3 75174 male
4 42154female
5 93188 male
6 63178 male
7 58170female
8 75167 male
9 89181 male
10 67 178 female

```

file
Here, the argument `file` specifies the desired file, which has no header, so `header=FALSE`. You set `stringsAsFactors=TRUE` because you do want to treat the variable (the only non-numeric variable) as a factor. There are no missing values, so you don't need to specify `na.strings` (though if there were, this argument is simply used in the same way as earlier), and by definition, `.csv` files are comma-delimited, which `read.csv` correctly implements by default, so you don't need the `sep` argument. The resulting data frame, `spread`, can then be printed in your R console.

As you can see, reading tabular data into R is fairly straightforward—you just need to be aware of how the data file is headed and delimited and how missing entries are identified. The simple table format is a natural and common way for data sets to be stored, but if you need to read in a file with a more complicated structure, R and its contributed packages make available some more sophisticated functions. See, for example, the documentation for the `scan` and `readLines` functions, which provide advanced control over how to parse a file. You can also find documentation on `read.table` and `read.csv` by accessing `?read.table` from the prompt.

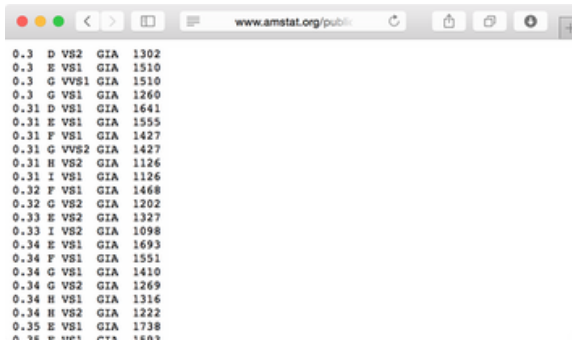
8.2.3 Web-Based Files

With an Internet connection, R can read in files from a website with the same `read.table` command. All the same rules concerning headers, delimiters, and missing values remain in place; you just have to specify the URL address of the file instead of a local folder location.

As an example, you'll use the online repository of data sets made available by the *Journal of Statistics Education* (JSE) through the American Statistical Association at http://www.amstat.org/publications/jse/jse_data_archive.htm.

One of the first files linked to at the top of this page is the table-format data set `4cdata.txt` (<http://www.amstat.org/publications/jse/v9n2/4cdata.txt>), which contains data on the characteristics of 308 diamonds from an analysis by Chu (2001) based on an advertisement in a Singaporean newspaper. Figure 8-5 shows the data.

You can look at the documentation file (`4c.txt`) and the accompanying article linked from the JSE site for details on what is recorded in this table. Note that of the five columns, the first and fifth are numeric, and the others would be well represented by factors. The delimiter is blank whitespace, there's no header, and there are no missing values (so you don't have to specify a value used to represent them).



0.3	D	VS2	GIA	1302
0.3	E	VS1	GIA	1510
0.3	G	VVS1	GIA	1510
0.3	G	VS1	GIA	1260
0.31	D	VS1	GIA	1641
0.31	E	VS1	GIA	1555
0.31	F	VS1	GIA	1427
0.31	G	VVS2	GIA	1427
0.31	H	VS2	GIA	1126
0.31	I	VS1	GIA	1126
0.32	F	VS1	GIA	1468
0.32	G	VS2	GIA	1202
0.33	E	VS2	GIA	1327
0.33	I	VS2	GIA	1098
0.34	E	VS1	GIA	1693
0.34	F	VS1	GIA	1551
0.34	G	VS1	GIA	1410
0.34	G	VS2	GIA	1269
0.34	H	VS1	GIA	1316
0.34	H	VS2	GIA	1222
0.35	E	VS1	GIA	1738

Figure 8-5: A table-format data file found online

With this in mind, you can create a data frame directly from the R prompt simply with the following lines:

```
R> dia.url <- "http://www.amstat.org/publications/jse/v9n2/4cdata.txt"
R> diamonds <- read.table(dia.url)
```

Note that you haven't supplied any extra values in this call to `read.table` because the defaults all work just fine. Because there's no header in the table, you can leave the default header value `FALSE`. The default value for `sep` is `" "`, meaning whitespace (not to be confused with `" "`, meaning an explicit space character), which is exactly what this table uses. The default value

for `stringsAsFactors` is `TRUE`, which is what you want for your character string columns. Following the import, you can supply names (based on the information in the documentation) to each column as follows:

```
R> names(diamonds) <- c("Carat", "Color", "Clarity", "Cert", "Price")
R> diamonds[1:5,]
Carat Color Clarity Cert Price
1 0.30 D VS2 GIA 1302
2 0.30 E VS1 GIA 1510
3 0.30 G VVS1 GIA 1510
4 0.30 G VS1 GIA 1260
5 0.31 D VS1 GIA 1641
```

Viewing the first five records shows that the data frame is displayed as you intended.

8.2.4 Other File Formats

There are other file formats besides `.txt` or `.csv` files that can be read into

R, such as the data file format `.dat`. These files can also be imported using `read.table`, though they may contain extra information at the top that must be skipped using the optional `skip` argument. The `skip` argument asks for

the number of lines at the top of the file that should be ignored before R begins the import.

As mentioned in Section 8.2.2, there are also contributed packages that can cope with other statistical software files; however, if there are multiple worksheets within a file it can complicate things. The R package *foreign* (R Core Team, 2015), available from CRAN, provides support for reading data files used by statistical programs such as Stata, SAS, Minitab, and SPSS. Other contributed packages on CRAN can help R handle files from various database management systems (DBMSs). For example, the *RODBC* package (Ripley and Lapsley, 2013) lets you query Microsoft Access databases and return the results as a data frame object. Other interfaces include the packages *RMySQL* (James and DebRoy, 2012) and *RJDBC* (Urbanek, 2013).

8.3 Writing Out Data Files and Plots

Writing out new files from data frame objects with R is just as easy as reading in files. R's vector-oriented behavior is a fast and convenient way to recode data sets, so it's perfect for reading in data, restructuring it, and writing it back out to a file.

8.3.1 Data Sets

The function for writing table-format files to your computer is `write.table`. You supply a data frame object as `x`, and this function writes its contents to a new file with a specified name, delimiter, and missing value string. For example, the following line takes the `mydatafile` object from Section 8.2 and writes it to a file:

```
R> write.table(x=mydatafile,file="/Users/tdavies/somenewfile.txt",
              sep="@",na="??",quote=FALSE,row.names=FALSE)
```

You provide with the folder location, ending in the filename you want for your new data file. This command creates a new table-format file `c@alled somenewfile.txt` in the specified folder location, delimited by and with missing values denoted with `??` (because you're actually creating a new file, the file choose command doesn't tend to be used here). Since `mydatafile` has variable names, these are automatically written to the file as a header. The optional logical argument `quote` determines whether to encapsulate each non-numeric entry in double quotes (if you explicitly need them in your file for, say, formatting requirements of other software); request no quotes by setting the argument to `FALSE`. Another optional logical argument, `row.names`, asks whether to include the row names of `mydatafile` (in this example, this would just be the numbers 1 to 6), which you also omit with `FALSE`. The resulting file, shown in Figure 8-6, can be opened in a text editor. Like `read.csv`, `write.csv` is a shortcut version of the `write.table` function designed specifically for `.csv` files.

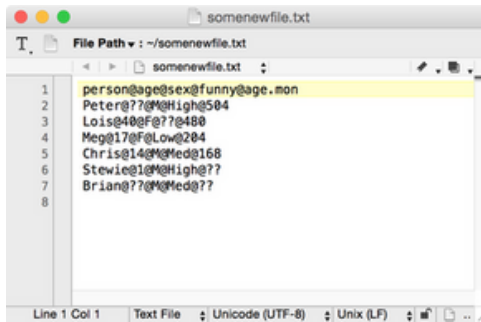


Figure 8-6: The contents of somenewfile.txt

8.3.2 Plots and Graphics Files

Plots can also be written directly to a file. In Chapter 7, you created and displayed plots in an active graphics device. This graphics device needn't be a screen window; it can be a specified file. Instead of displaying the plot immediately on the screen, you can have R follow these steps: open a “file” graphics device, run any plotting commands to create the final plot, and close the device. R supports direct writing to *.jpeg*, *.bmp*, *.png*, and *.tiff* files using functions of the same names. For example, the following code uses these three steps to create a *.jpeg* file:

```
R> jpeg(filename="/Users/tdavies/myjpegplot.jpeg",width=600,height=600) R>
plot(1:5,6:10,ylab="a nice ylab",xlab="here's an xlab",
main="a saved .jpeg plot")
R> points(1:5,10:6,cex=2,pch=4,col=2)
R> dev.off()
null device
1
```

The file graphics device is opened by a call to `jpeg`, where you provide the intended name of the file and its folder location as `filename`. By default, the dimensions of the device are set to 480 × 480 pixels, but here you change them to 600

× 600. You could also set these dimensions by supplying other units (inches, centimeters, or millimeters) to width and height and by specifying the unit with an optional `units` argument. Once the file is opened, you execute any R plotting commands you need in order to create the image—this example plots some points and then includes some additional points with a second command. The final graphical result is silently written to the specified file just as it would have been displayed on the screen. When you've finished plotting, you must explicitly close the file device with a call to `dev.off()`, which prints information on the remaining active device (here, “null device” can be loosely interpreted as “nothing is left open”). If `dev.off()` isn't called, then R will continue to output any subsequent plotting commands to the file, and possibly overwrite what you have there. The left plot in Figure 8-7 shows the resulting file created in this example.

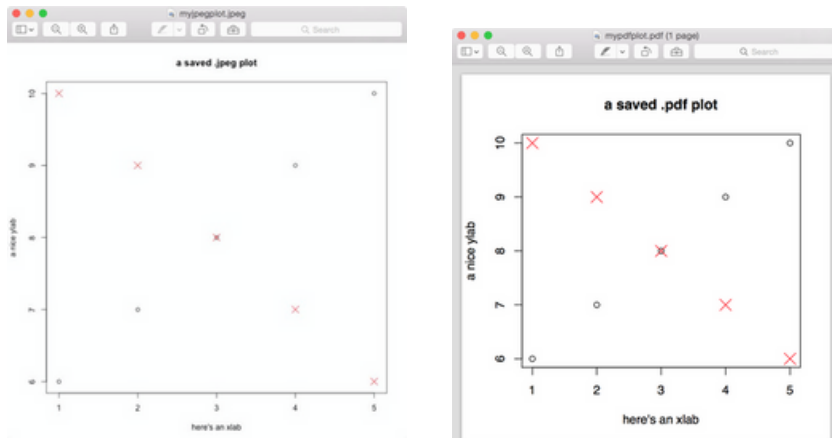


Figure 8-7: R plots that have been written directly to disk: a.jpegversion (left) and a .pdf version (right) of the same plotting commands

You can also store R plots as other file types, such as PDFs (using the `pdf` function) and EPS files (using the `postscript` function). Though some argument names and default values are different for these functions, they follow the same basic premise. You specify a folder location, a filename, and width and height dimensions; enter your plotting commands; and then close the device with `dev.off()`. The right panel of Figure 8-7 shows the .pdf file created with the following code:

```
R> pdf(file="/Users/tdavies/mypdfplot.pdf",width=5,height=5) R>
plot(1:5,6:10,ylab="a nice ylab",xlab="here's an xlab",
main="a saved .pdf plot")
R> points(1:5,10:6,cex=2,pch=4,col=2)
R> dev.off()
null device
1
```

Here, you use the same plotting commands as before, and there are just a few minor differences in the code. The argument for the file is `file` (as opposed to `filename`), and the units for width and height default to inches in pdf. The difference of appearance between the two images in Figure 8-7 results primarily from these differences in width and height.

This same process also works for `ggplot2` images. True to style, however, `ggplot2` provides a convenient alternative. The `ggsave` function can be used to write the most recently plotted `ggplot2` graphic to file and performs the device open/close action in one line.

For example, the following code creates and displays a `ggplot2` object from a simple data set.

```
R> foo <- c(1.1,2,3.5,3.9,4.2) R>
bar <- c(2,2.2,-1.3,0,0.2)
```

```
R> qplot(foo,bar,geom="blank")
      + geom_point(size=3,shape=8,color="darkgreen")
      + geom_line(color="orange",linetype=4)
```

Now, to save this plot to a file, all you need is the following line:

```
R> ggsave(filename="/Users/tdavies/mypngqplot.png")
Saving 7 x 7 in image
```

This writes the image to a *.png* file in the specified filename directory. (Note that dimensions are reported if you don't explicitly set them using width and height; these will vary depending on the size of your graphics device.) The result is shown in Figure 8-8.

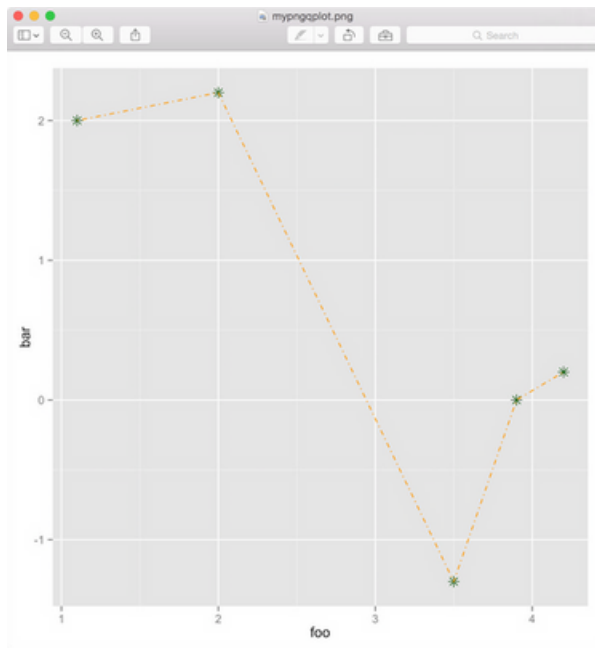


Figure 8-8: The *.png* file created using `ggplot2`'s `ggsave` command

Beyond just being concise, `ggsave` is convenient in a few other ways. For one, you can use the same command to create a variety of image file types—the type is simply determined by the extension you supply in the filename argument. Also, `ggsave` has a range of optional arguments if you want to control the size of the image and the quality or scaling of the graphic. For more details on saving images from base R graphics, see the `?jpeg`, `?pdf`, and `?postscript` help files. You can consult `?ggsave` for more on saving images with `ggplot2`.

8.4 AdHocObjectRead/WriteOperations

For the typical R user, the most common input/output operations will probably revolve around data sets and plot images. But if you need to read or write other kinds of R objects, such as lists or arrays, you'll need the `dput` and `dget` commands, which can handle objects in a more ad hoc style. Suppose, for example, you create this list in the current session:

```
R> somelist <- list(foo=c(5,2,45),
                    bar=matrix(data=c(T,T,F,F,F,T,F,T),nrow=3,ncol=3),
                    baz=factor(c(1,2,2,3,1,1,3),levels=1:3,ordered=T))
R> somelist
$foo
[1] 5 2 45

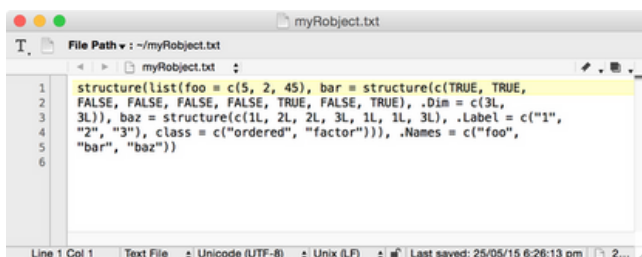
$bar
[,1] [,2] [,3]
[1,] TRUE FALSE TRUE
[2,] TRUE FALSE FALSE
[3,] FALSE FALSE TRUE

$baz
[1] 1 2 2 3 1 1 3
Levels: 1 < 2 < 3
```

This object can itself be written to a file, which is useful if you want to pass it to a colleague or open it in a new R session elsewhere. Using `dput`, the following line stores the object as a plain-text file that is interpretable by R:

```
R> dput(x=somelist,file="/Users/tdavies/myRobject.txt")
```

In technical terms, this command creates an American Standard Code for Information Interchange (ASCII) representation of the object. As you call `dput`, the object you want to write is specified as `x`, and the folder location and name of the new plain-text file are passed to `file`. Figure 8-9 shows the contents of the resulting file.

A screenshot of a text editor window titled 'myRobject.txt'. The window shows the ASCII representation of the R object 'somelist' created by the 'dput' command. The text is as follows:

```
1 structure(list(foo = c(5, 2, 45), bar = structure(c(TRUE, TRUE,
2 FALSE, FALSE, FALSE, TRUE, FALSE, TRUE), .Dim = c(3L,
3 3L)), baz = structure(c(1L, 2L, 2L, 3L, 1L, 1L, 3L), .Label = c("1",
4 "2", "3"), class = c("ordered", "factor"))), .Names = c("foo",
5 "bar", "baz"))
6
```

The status bar at the bottom indicates 'Line 1 Col 1', 'Text File', 'Unicode (UTF-8)', 'Unix (LF)', and 'Last saved: 25/05/15 6:26:13 pm'.

Figure 8-9: myRobject.txt created by using `dput` on `somelist`

Notice that `dput` stores all of the members of the object plus any other relevant information, such as attributes. The third element of `somelist`, for example, is an ordered factor, so it isn't enough to simply represent it in the text file as a stand-alone vector.

Now, let's say you want to import this list into an R workspace. If a file has been created with `dput`, then it can be read into any other workspace using `dget`.

```
R> newobject <- dget(file="/Users/tdavies/myRobject.txt")
```

```
R> newobject
```

```
$foo
```

```
[1] 5 245
```

```
$bar
```

```
[,1] [,2] [,3]
```

```
[1,] TRUE FALSE TRUE
```

```
[2,] TRUE FALSE FALSE
```

```
[3,] FALSE FALSE TRUE
```

```
$baz
```

```
[1] 1 2 2 3 1 1 3
```

```
Levels: 1 < 2 < 3
```

```
dget
```

You read the object from the *myRobject.txt* file using `dget` and assign it to `newobject`. This object is the same as the original object, with all structures and attributes present.

There are some drawbacks to using these commands. For starters, `dput` is not as reliable a command as `write.table` because it's sometimes quite difficult for R to create the necessary plain-text representation for an object (fundamental object classes typically cause no problems, but complex user-defined classes can). Also, because they need to store structural information, files created using `dput` are relatively inefficient both in terms of required space and in terms of how long it takes to execute read and write operations. This becomes more noticeable for objects that contain a lot of data. Nevertheless, `dput` and `dget` are useful ways to store or transfer specific objects without having to save an entire workspace.

Exercise 8.1

- a. In R's built-in datasets library is the data frame `quakes`. Make sure you can access this object and view the corresponding help file to get an idea of what this data represents. Then, do the following:
 - i. Select only those records that correspond to a magnitude (mag) of greater than or equal to 5 and write them to a table-format file called *q5.txt* in an existing folder on your

- machine. Use a delimiting character of ! and do not include any row names.
- ii. Read the file back into your R workspace, naming the object `q5.dframe`.
- b. In the contributed package `car`, there's a data frame called `Duncan`, which provides historical data on perceived job prestige in 1950. Install the `car` package and access the `Duncan` data set and its help file. Then, do the following:
- i. Write R code that will plot education on the x-axis and income on the y-axis, with both x- and y-axis limits fixed to be `[0,100]`. Provide appropriate axis labels. For jobs with a prestige value of less than or equal to 80, use a black point character. For jobs with prestige greater than 80, use a blue point character.
 - ii. Add a legend explaining the difference between the two points.
- c. Create a list called `exer` that contains the three data sets `quakes`, `q5.dframe`, and `Duncan`. Then, do the following:
- i. Write the list object directly to disk, calling it `Exercise8-1.txt`. Briefly inspect the contents of the file in a text editor.
 - ii. Read `Exercise8-1.txt` back into your workspace; call the resulting object `list.of.dataframes`. Check that `list.of.dataframes` does indeed contain the three data frame objects.
- d. In Section 7.4.3, you created a `ggplot2` graphic of 20 observations displayed as the bottom image of Figure 7-11 on page 144. Use `ggsave` to save a copy of this plot as a `.tiff` file.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
<code>data</code>	Load contributed data set	Section 8.1.2, p. 149
<code>read.table</code>	Import table-format data file	Section 8.2.1, p. 151
<code>list.files</code>	Print specific folder contents	Section 8.2.1, p. 151
<code>file.choose</code>	Interactive file selection	Section 8.2.1, p. 152
<code>read.csv</code>	Import comma-delimited file	Section 8.2.2, p. 153
<code>write.table</code>	Write table-format file to disk	Section 8.3.1, p. 156
<code>jpeg</code> , <code>bmp</code> , <code>png</code> , <code>dev.off</code>	Write image/plot file to disk	Section 8.3.2, p. 157
<code>tiff</code> , <code>pdf</code> , <code>postscript</code>	Write image/plot file to disk	Section 8.3.2, p. 158
<code>ggsave</code>	Write <code>ggplot2</code> plot file to disk	Section 8.3.2, p. 159
<code>dput</code>	Write R object to file (ASCII)	Section 8.4, p. 160
<code>dget</code>	Import ASCII object file	Section 8.4, p. 161

Part IIPR

9

CALLING FUNCTIONS



Before you start writing your own functions in R, it's useful to understand how functions are called and interpreted in an R session. First, you'll look at how variable names are compartmentalized in R. You'll see R's rules for naming arguments and objects, and how R searches for arguments and other variables when a function is called. Then you'll look at some alternative ways to specify arguments when calling a function.

9.1 Scoping

To begin with, it's important to understand R's *scoping rules*, which determine how the language compartmentalizes objects and retrieves them in a given session. This framework also defines the situations in which duplicate object names can exist at once. For example, you've used the argument `data` when calling `matrix` (Section 3.1), but `data` is also the name of a ready-to-use function that loads data sets from contributed packages (Section 8.1.2). In this section, you'll gain an introductory understanding of

how R behaves internally in these circumstances, which will help you later on when it comes to programming and executing your own functions and those of other packages.

9.1.1 Environments

R enforces scoping rules with virtual *environments*. You can think of environments as separate compartments where data structures and functions are stored. They allow R to distinguish between identical names that are associated with different scopes and therefore stored in different environments. Environments are dynamic entities—new environments can be created, and existing environments can be manipulated or removed.

NOTE *Technically speaking, environments don't actually contain items. Rather, they have pointers to the location of those items in the computer's memory. But using the "compartment" metaphor and thinking of objects "belonging to" these compartments is useful when you're first getting a general sense of how environments work.*

There are three important kinds of environments: global environments, package environments and namespaces, and local or lexical environments.

Global Environment

The *global environment* is the compartment set aside for user-defined objects. Every object you've created or overwritten so far has resided in the global environment of your current R session. In Section 1.3.1, I mentioned that a call to `ls()` lists all the objects, variables, and user-defined functions in the active workspace—more precisely, `ls()` prints the names of everything in the current global environment.

Starting with a new R workspace, the following code creates two objects and confirms their existence in the global environment:

```
R> foo <- 4+5
R> bar <- "stringtastic"
R> ls()
[1] "bar" "foo"
```

But what about all the ready-to-use objects and functions? Why aren't those printed alongside `foo` and `bar` as members of this environment? In fact, those objects and functions belong to package-specific environments, described next.

Package Environments and Namespaces

For simplicity, I'll use the term *package environment* rather loosely to refer to the items made available by each package in R. In fact, the structure

of R packages in terms of scoping is a bit more complicated. Each package environment actually represents several environments that control different aspects of a search for a given object. A package *namespace*, for example, essentially defines the visibility of its functions. (A package can

have visible functions that a user is able to use and invisible functions that provide internal support to the visible functions.) Another part of the package environment handles *imports* designations, dealing with any functions or objects from other libraries that the package needs to import for its own functionality.

To clarify this, you can think of all the ready-to-use functions and objects you're working with in this book as belonging to specific package environments. The same is true for the functions and objects of any contributed packages you've explicitly loaded with a call to `library`. You can use `ls` to list the items in a package environment as follows:

```
R> ls("package:graphics")
[1]"abline" "arrows" "assocplot"
[5]"Axis" "axis.Date" "axis.POSIXct"
[9]"barplot" "barplot.default" "box"
[13]"boxplot.default" "boxplot.matrix" "bxp"
[17]"clip" "close.screen" "co.intervals"
[21]"contour.default" "coplot" "curve"
[25]"erase.screen" "filled.contour" "fourfoldplot"
[29]"grconvertX" "grconvertY" "grid"
[33]"hist.default" "identify" "image"
[37]"layout" "layout.show" "lcm"
[41]"lines" "lines.default" "locator"
[45]"matplot" "matpoints" "mosaicplot"
[49]"pairs" "pairs.default" "panel.smooth"
[53]"persp" "pie" "plot"
[57]"plot.design" "plot.function" "plot.new"
[61]"plot.xy" "points" "points.default"
[65]"polypath" "rasterImage" "rect"
[69]"screen" "segments" "smoothScatter"
[73]"split.screen" "stars" "stem"
[77]"stripchart" "strwidth" "sunflowerplot"
[81]"text" "text.default" "title"
[85]"xspline" "xyinch" "yinch"
```

`ls`

The command `ls` lists all of the visible objects contained in the `graphics` package environment. Note that this list includes some of the functions you used in Chapter 7, such as `arrows`, `plot.segments`, and `.`

Local Environments

Each time a function is called in R, a new environment is created called the *local environment*, sometimes referred to as the *lexical environment*. This local environment contains all the objects and variables created in and visible to the function, including any arguments you've supplied to the function upon execution. It's this feature that allows the presence of argument names that are identical to other object names accessible in a given workspace.

For example, say you call `matrix` and pass in the argument `data`, as follows:

```
R>                                youthspeak                                <-  
matrix(data=c("OMG", "LOL", "WTF", "YOLO"), nrow=2, ncol=2) R> youthspeak  
[,1] [,2]  
[1,] "OMG" "WTF"  
[2,] "LOL" "YOLO"
```

Calling this function creates a local environment containing the `data` variable. When you execute the function, it begins by looking for in this

local environment. That means R isn't confused by other objects or functions named `data` in other environments (such as the `data` function automatically loaded from the `utils` package environment). If a required item isn't found in the local environment, only then does R begin to widen its search for that item (I'll discuss this feature a little more in Section 9.1.2). Once the function has completed, this local environment is automatically removed. The same comments apply to the `nrow` and `ncol` arguments.

9.1.2 Search Path

To access data structures and functions from environments other than the immediate global environment, R follows a *search path*. The search path lays out all the environments that a given R session has available to it.

The search path is basically a list of the environments that R will search when an object is requested. If the object isn't found in one environment, R proceeds to the next one. You can view R's search path at any time using `search()`

```
R> search()  
[1] ".GlobalEnv"          "tools:RGUI"          "package:stats"  
[4] "package:graphics"    "package:grDevices"    "package:utils"  
[7] "package:datasets"    "package:methods"    "Autoloads"  
[10] "package:base"
```

From the command prompt, this path will always begin at the global user environment (`.GlobalEnv`) and end after the base package environment (`package:base`). You can think of these as belonging to a hierarchy, with an arrow pointing from left to right between each pair of environments. For my current session, if I request a certain object at the R prompt, the program will inspect `.GlobalEnv` → `tools:RGUI` → `package:stats` → . . . → `package:base` in turn, stopping the search when the desired object is found and retrieved. Note that, depending on your operating system and whether you're using the built-in GUI, `tools:RGUI` might not be included in your search path. If R doesn't find what it's looking for by following the environments in the search path, the *empty environment* is reached. The empty environment is not explicitly listed in the output from `search()`, but it's always the final

destination after `package:base`. This environment is special because it marks the end of the search path.

For example, if you call the following, a number of things happen internally:

```
R> baz <- seq(from=0,to=3,length.out=5)
R> baz
[1] 0.00 0.75 1.50 2.25 3.00
```

R first searches the global environment for a function called `seq`, and when this isn't found, it goes on to search in the enclosing environment, which is the next level up in the search path (according to the left-to-right arrows mentioned earlier). It doesn't find it there, so R keeps going through the path to the next environment, searching the packages that have been loaded (automatically or otherwise) until it finds what it's looking for. In this example, R locates `seq` in the built-in base package environment. Then it executes the `seq` function (creating a temporary local environment) and assigns the results to a new object, `baz`, which resides in the global environment. In the subsequent call to print `baz`, R begins by searching the global environment and immediately finds the requested object.

You can look up the enclosing environment of any function using `environment`, as follows:

```
R> environment(seq)
<environment: namespace:base>
R> environment(arrows)
<environment: namespace:graphics>
```

Here, I've identified the package namespace of `base` as the owner of the `seq` function and the `graphics` package as the owner of the `arrows` function. Each environment has a *parent*, to direct the order of the search path. Examining the earlier output from the call `search()`, you can see that the parent of `package:stats`, for example, is `package:graphics`. The specific parent-child structure is dynamic in the sense that the search path changes when additional libraries are loaded or data frames are attached. When you load a contributed package with a call to `library`, this essentially just inserts the desired package in the search path. For example, in Exercise 8.1 on page 161, you installed the contributed package `car`. After loading this package, your search path will include its contents.

```
R> library("car")
R> search()
[1] ".GlobalEnv"           "package:car"           "tools:RGUI"
     [4] "package:stats"        "package:graphics"      "package:grDevices"
     [7] "package:utils"        "package:datasets"      "package:methods"
    [10] "Autoloads"            "package:base"
```

Note the position of the car package environment in the path—inserted directly after the global environment. This is where each subsequently loaded package will be placed (followed by any additional packages it depends upon for its own functionality). As noted earlier, R will stop searching once it has exhausted the entire search path and reached the empty environment. If you request a function or object that you haven't defined, that doesn't exist, or that is perhaps in a contributed package that you've forgotten to load (this is quite a common little mistake), then an error is thrown. These “cannot find” errors are recognizable for both functions and other objects.

```
R> neither.here()
Error: could not find function "neither.here" R>
nor.there
Error: object 'nor.there' not found
```

Environments help compartmentalize the huge amount of functionality in R. This becomes particularly important when there are functions with the same name in different packages in the search path. At that point, *masking*, discussed in Section 12.3, comes into play.

As you get more comfortable with R and want more precise control over how it operates, it's worth investigating in full how R handles environments. For more technical details on this, Gupta (2012) provides a particularly well-written online article.

9.1.3 Reserved and Protected Names

A few key terms are strictly forbidden from being used as object names in R. These *reserved* names are necessary in order to protect fundamental operations and data types frequently used in the language.

The following identifiers are reserved:

- a
- ndif else
- , ,andforwhile in function
- repeatbreak next TRUE
- , ,andFALSE
- andInf -Inf
- andNANaN NULL
- , ,and

I haven't yet covered some of the terms on this list. These items represent the core tools for programming in the R language, and you'll begin to explore them in the following chapter. The last three bullet points

include the familiar logical values (Section 4.1) and special terms used to represent things like infinity and missing entries (Section 6.1).

If you try to assign a new value to any of these reserved terms, an error occurs.

```
R> NaN <- 5
Error in NaN <- 5 : invalid (do_set) left-hand side to assignment
```

Because R is case sensitive, it's possible to assign values to any case-variant of the reserved names, but this can be confusing and is generally not advisable.

```
R> False <- "confusing"
R> nan <- "this is"
R> cat(nan,False)
this is confusing
```

Also be wary of assigning values to T and F, the abbreviations of TRUE and FALSE. The full identifiers TRUE and FALSE are reserved, but the abbreviated versions are not.

```
R> T <- 42
R> F <-
TRUE R>
F&&TRUE [1]
TRUE
```

Assigning values to T and F this way will affect any subsequent code that intends to use T and F to refer to TRUE and FALSE. The second assignment (F <- TRUE) is perfectly legal in R's eyes, but it's extremely confusing given the normal usage of F as an abbreviation: the line F&&TRUE now represents a TRUE&&TRUE comparison! It's best to simply avoid these types of assignments. If you've been following along with the examples in your R console, it's prudent at this point to clear the global environment (thereby deleting the objects False, nan, T, and F from your workspace). To do this, use the rm function as shown next. Using ls(), supply a character vector of all objects in the global environment as the argument list.

```
R> ls()
[1]"bar"      "baz"      "F"       [7]"T"          "False"    "foo"      "nan"
"youthspeak" R> rm(list=ls())
R> ls()
character(0)
```

Now the global environment is empty, and calling ls() returns an empty character vector (character(0)).

Exercise 9.1

- a. Identify the first 20 items contained in the built-in and automatically loaded methods package. How many items are there in total?
- b. Determine the environment that owns each of the following functions:
 - i. `read.table`
 - ii. `data`
 - iii. `matrix`
 - iv. `jpeg`
- c. Use `is` and a test for character string equality to confirm the function `smoothScatter` is part of the `graphics` package.

9.2 ArgumentMatching

Another set of rules that determine how R interprets function calls has to do with *argument matching*. Argument matching conditions allow you to provide arguments to functions either with abbreviated names or without names at all.

9.2.1 Exact

So far, you've mostly been using *exact* matching of arguments, where each argument tag is written out in full. This is the most exhaustive way to call a function. It's helpful to write out full argument names this way when first getting to know R or a new function.

Other benefits of exact matching include the following:

- Exact matching is less prone to mis-specification of arguments than other matching styles.
- The order in which arguments are supplied doesn't matter.
- Exact matching is useful when a function has many possible arguments but you want to specify only a few.

The main drawbacks of exact matching are clear:

- It can be cumbersome for relatively simple operations.
- Exact matching requires the user to remember or look up the full, case-sensitive tags.

As an example, in Section 6.2.1, you used exact matching to execute the following:

```
R> bar <- matrix(data=1:9,nrow=3,ncol=3,dimnames=list(c("A","B","C"),
                                                    c("D","E","F")))
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

`bar`.
This creates a 3 × 3 matrix object with a attribute for the rows and columns. Since the argument tags are fully specified, the order of the arguments doesn't matter. You could switch around the arguments, and the function still has all the information it requires.

```
R> matrix(nrow=3,dimnames=list(c("A","B","C"),c("D","E","F")),ncol=3,
data=1:9)
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

This behaves the same way as the previous function call. For the sake of consistency, you usually won't switch around arguments each time you call a function, but this example shows a benefit of exact matching: you don't have to worry about the order of any optional arguments or about skipping them.

9.2.2 Partial

Partial matching lets you identify arguments with an abbreviated tag. This can shorten your code, and it still lets you provide arguments in any order. Here is another way to call `matrix` that takes advantage of partial matching:

```
R> bar <- matrix(nr=3,di=list(c("A","B","C"),c("D","E","F")),nc=3,dat=1:9) R>
bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

Notice I've shortened the `nrow`, `dimnames`, and `ncol` argument tags to the first two letters and shortened the `data` argument to the first three. For partial matching, there's no set number of letters you have to provide, as long as each argument is still uniquely identifiable for the function being called. Partial matching has the following benefits:

- It requires less code than exact matching.
- Argument tags are still visible (which limits the possibility of mis-specification).
- The order of supplied arguments still doesn't matter.

But partial matching also has some limitations. For one, it gets trickier if there are multiple arguments whose tags start with the same letters. Here's an example:

```
R> bar <- matrix(nr=3,di=list(c("A","B","C"),c("D","E","F")),nc=3,d=1:9) Error in
matrix(nr = 3, di = list(c("A", "B", "C"), c("D", "E", "F")), :
argument 4 matches multiple formal arguments
```

An error has occurred. The fourth argument tag is designated simply as `d`, which is meant to stand for `data`. This is illegal because another argument, namely `dimnames`, also starts with `d`. Even though `dimnames` is specified separately as `di` earlier in the same line, the call isn't valid. Drawbacks of partial matching include the following:

- The user must be aware of other potential arguments that can be matched by the shortened tag (even if they aren't specified in the call or have a default value assigned).
- Each tag must have a unique identification, which can be difficult to remember.

9.2.3 Positional

The most compact mode of function calling in R is *positional matching*. This is when you supply arguments without tags, and R interprets them based solely on their order.

Positional matching is usually used for relatively simple functions with only a few arguments, or functions that are very familiar to the user. For this type of matching, you *must* be aware of the precise positions of each argument. You can find that information in the "Usage" section of the function's help file, or it can be printed to the console with the `args` function. Here's an example:

```
R> args(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL) NULL
```

This shows the defined order of arguments of the matrix function, as well as the default value for each argument. To construct the matrix bar with positional matching, execute the following:

```
R> bar <- matrix(1:9,3,3,F,list(c("A","B","C"),c("D","E","F"))) R>
bar DEF
A 1 4 7
B 2 5 8
C 3 6 9
```

The benefits of positional matching are as follows:

- Shorter, cleaner code, particularly for routine tasks
- No need to remember specific argument tags

Notice that when using exact and partial matching, you didn't need to supply anything for the `byrow` argument, which, by default, is set to `FALSE`. With positional matching, you must provide a value (given here as `F`) for `byrow` as the fourth argument because R relies on position alone to interpret the function call. If you leave out the argument, you get an error, as follows:

```
R> bar <- matrix(1:9,3,3,list(c("A","B","C"),c("D","E","F")))
Error in matrix(1:9, 3, 3, list(c("A", "B", "C"), c("D", "E", "F"))):
invalid 'byrow' argument
```

Here R has tried to assign the fourth argument (the list you intended for `dimnames`) as the value for the logical `byrow` argument. This brings us to the drawbacks of positional matching:

- You must look up and exactly match the defined order of arguments.
- Reading code written by someone else can be more difficult, especially when it includes unfamiliar functions.

9.2.4 Mixed

Since each matching style has pros and cons, it's quite common, and perfectly legal, to mix these three styles in a single function call. For instance, you can avoid the type of error shown in the previous example like so:

```
R> bar <- matrix(1:9,3,3,dim=list(c("A","B","C"),c("D","E","F"))) R>
bar DEF
A 1 4 7
B 2 5 8
C 3 6 9
```

Here I've used positional matching for the first three arguments, which are by now familiar to you. At the same time, I've used partial matching to explicitly tell R that the list is meant as a `dimnames` value, not for `byrow`.

9.2.5 Dot-Dot-Dot: Use of Ellipses

Many functions exhibit *variadic* behavior. That is, they can accept any number of arguments, and it's up to the user to decide how many arguments

to provide. The functions `c`, `data.frame`, and `list` are all like this. When you call a function like `data.frame`, you can specify any number of members as arguments.

This flexibility is achieved in R through the special *dot-dot-dot* designation (`...`), also called the *ellipsis*. This construct allows the user to supply any number of data vectors (these become the columns in the final data frame). You can see whether an ellipsis is used in a function on the function's help page or with `args`. Looking at `data.frame`, notice the first argument slot is an ellipsis:

```
R> args(data.frame)
function (..., row.names = NULL, check.rows = FALSE, check.names =
TRUE, stringsAsFactors = default.stringsAsFactors())
NULL
```

When you call a function and supply an argument that can't be matched with one of the function's defined argument tags, normally this would produce an error. But if the function is defined with an ellipsis, any arguments that aren't matched to other argument tags are matched to the ellipsis. Functions that employ ellipses generally fall into two groups. The first group includes functions such as `c`, `data.frame`, and `list`, where the ellipsis always represents the "main ingredients" in the function call. That is, the objective of the function is to use contents of the ellipsis in the resulting object or output. The second group consists of functions where the ellipsis is meant as a *supplementary* or *potential* repository of optional arguments. This is common when the function of interest calls other *subfunctions* that themselves require additional arguments depending upon the originally supplied items. Rather than explicitly copy all the arguments desired by the subfunction into the argument list of the "parent" function, the parent function can instead be defined including an ellipsis that is subsequently provided to the subfunction.

Here's an example of the ellipsis used for supplementary arguments with the generic function:

`plot`

```
R> args(plot)
function (x, y, ...)
NULL
```

From examining the arguments, it's clear that optional arguments such as point size (argument tag `cex`) or line type (argument tag `lty`), if supplied,

are matched to the ellipsis. These optional arguments are then passed in to the function to be used by various methods that tweak graphical parameters.

Ellipses are a convenient programming tool for writing variadic functions or functions where an unknown number of arguments may be supplied. This will become clearer when you start writing your own functions in Chapter 11. However, when writing functions like this, it's important to properly document the intended use of ... so the potential users of the function know exactly which arguments can be passed to it and what those arguments are subsequently used for in execution.

Exercise 9.2

- a. Use positional matching with `seq` to create a sequence of values between -4 and 4 that progresses in steps of 0.2.
- b. In each of the following lines of code, identify which style of argument matching is being used: exact, partial, positional, or mixed. If mixed, identify which arguments are specified in each style.
 - i. `array(8:1,dim=c(2,2,2))`
 - ii. `rep(1:2,3)`
 - iii. `seq(from=10,to=8,length=5)`
 - iv. `sort(decreasing=T,x=c(2,1,1,2,0.3,3,1.3))`
 - v. `which(matrix(c(T,F,T,T),2,2))`
 - vi. `which(matrix(c(T,F,T,T),2,2),a=T)`
- c. Suppose you explicitly ran the plotting function `plot.default` and supplied values to arguments tagged `type`, `pch`, `xlab`, `ylab`, `lwd`, `lty`, and `col`. Use the function documentation to determine which of these arguments fall under the umbrella of the ellipsis.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
<code>ls</code>	Inspect environment objects	Section 9.1.1, p. 167
<code>search</code>	Current search path	Section 9.1.2, p. 168
<code>environment</code>	Function environment properties	Section 9.1.2, p. 169
<code>rm</code>	Delete objects in workspace	Section 9.1.3, p. 171
<code>args</code>	Show function arguments	Section 9.2.3, p. 174

10



To write more sophisticated programs with R, you'll need to control the flow and order of execution in your code. One fundamental way to do this is to make the execution of certain sections of code dependent on a *condition*. Another basic control mechanism is the *loop*, which repeats a block of code a certain number of times. In this chapter, we'll explore these core programming techniques using if-else statements, for and while loops, and other control structures.

10.1 if Statements

The if statement is the key to controlling exactly which operations are carried out in a given chunk of code. An if statement runs a block of code only if a certain condition is true. These constructs allow a program to respond differently depending on whether a condition is TRUE or FALSE.

10.1.1 Stand-Alone Statement

Let's start with the stand-alone `if` statement, which looks something like this:

```
if (condition){  
    do any code here  
}
```

The *condition* is placed in parentheses after the `if` keyword. This condition must be an expression that yields a single logical value (TRUE or FALSE). If it's TRUE, the code in the braces, {}, will be executed. If the condition isn't satisfied, the code in the braces is skipped, and R does nothing (or continues on to execute any code after the closing brace).

Here's a simple example. In the console, store the following:

```
R>a<-3  
R> mynumber <- 4
```

Now, in the R editor, write the following code chunk:

```
if(a<=mynumber){  
    a <- a^2  
}
```

When this chunk is executed, what will the value of `a` be? It depends on the condition defining the `if` statement, as well as what's actually specified in the braced area. In this case, when the condition `a<=mynumber` is evaluated, the result is TRUE since 3 is indeed less than or equal to 4. That means the code inside the braces is executed, which sets `a` to `a^2`, or 9.

Now highlight the entire chunk of code in the editor and send it to the console for evaluation. Remember, you can do this in several ways:

- Copy and paste the selected text from the editor directly into the console.
- From the menu, select **Edit** → **Run line or selection** in Windows or select

Edit

→ **Execute** in OS X.



- Use the keystroke shortcuts such as `CTRL-R` in Windows or `⌘-RETURN` on a Mac.

Once you execute the code in the console, you'll see something

like this:

```
R> if(a<=mynumber){  
+ a<-a^2  
+ }
```

Then, look at the objecta, shown here:

```
R>a
[1]
9
```

Next, suppose you execute the same if statement again right away. Will a be squared once more, giving 81? Nope! Since a is now 9 and mynumber is still 4, the condition `a<=mynumber` will be FALSE, and the code in the braces will not be executed; a will remain at 9.

Note that after you send the if statement to the console, each line after the first is prefaced by a +. These + signs do not represent any kind of arithmetic addition; rather, they indicate that R is expecting more input before it begins execution. For example, when a left brace is opened, R will not begin any kind of execution until that section is closed with a right brace. To avoid redundancy, in future examples I won't show this repetition of code sent from the editor to the console.

NOTE *You can change the prompt symbol by assigning a different character string to the `continue` component of the `getOption` command, in the way you reset the prompt in Section 1.2.1.*

R's
The if statement offers a huge amount of flexibility—you can place any kind of code in the braced area, including more if statements (see the upcoming discussion of nesting in Section 10.1.4), enabling your program to make a sequence of decisions.

To illustrate a more complicated if statement, consider the following two new objects:

```
R> myvec <- c(2.73,5.40,2.15,5.29,1.36,2.16,1.41,6.97,7.99,9.52) R>
myvec
[1] 2.73 5.40 2.15 5.29 1.36 2.16 1.41 6.97 7.99 9.52
R> mymat <- matrix(c(2,0,1,2,3,0,3,0,1,1),5,2)
R> mymat
[,1] [,2]
[1,] 2 0
[2,] 0 3
[3,] 1 0
[4,] 2 1
[5,] 3 1
```

Use these two objects in the code chunk given here:

```
if(any((myvec-1)>9)||matrix(myvec,2,5)[2,1]<=6){
cat("Condition satisfied --\n")
new.myvec <- myvec
new.myvec[seq(1,9,2)] <- NA
mylist <- list(aa=new.myvec,bb=mymat+0.5)
cat("-- a list with",length(mylist),"members now exists.")
}
```

Send this to the console, and it produces the following output:

```
Condition satisfied --  
-- a list with 2 members now exists.
```

Indeed, an object `mylist` has been created that you can examine.

```
R> mylist  
$aa  
 [1] NA 5.40   NA 5.29   NA 2.16   NA 6.97   NA 9.52  
  
$bb  
      [,1] [,2]  
[1,] 2.5 0.5 [2,]  
0.5 3.5 [3,] 1.5  
0.5 [4,] 2.5 1.5  
[5,] 3.5 1.5
```

In this example, the condition consists of two parts separated by an OR statement using `||`, which produces a single logical result. Let's walk through it.

- The first part of the condition looks at `myvec-1`, takes away from each element, and checks whether any of the results are greater than 9. If you run this part on its own, it yields `FALSE`.

```
R> myvec-1  
[1] 1.73 4.40 1.15 4.29 0.36 1.16 0.41 5.97 6.99 8.52  
R> (myvec-1)>9  
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  
FALSE R> any((myvec-1)>9)  
[1] FALSE
```

- The second part of the condition uses positional matching in a call to `matrix` to construct a two-row, five-column, column-filled matrix using entries of the original `myvec`. Then, the number in the second row of the first column of that result is checked to see whether it's less than or equal to 6, which it is.

```
R> matrix(myvec,2,5)  
      [,1] [,2] [,3] [,4] [,5]  
[1,] 2.73 2.15 1.36 1.41 7.99  
[2,] 5.40 5.29 2.16 6.97 9.52 R>  
matrix(myvec,2,5)[2,1]  
[1] 5.4  
R> matrix(myvec,2,5)[2,1]<=6 [1]  
TRUE
```

This means the overall condition being checked by the `if` statement will be `TRUE` because `as.numeric(myvec[2,1]) > 9`.

```
R> any((myvec-1)>9)||matrix(myvec,2,5)[2,1]<=6
[1] TRUE
```

As a result, the code inside the braces is accessed and executed. First, it prints the "Condition satisfied" string and copies `myvec` to `new.myvec`. Using `seq`, it then accesses the odd-numbered indexes of `new.myvec` and overwrites them with `NA`. Next, it creates `mylist`. In this list, `new.myvec` is stored in a member named `aa`, and then it takes the original `mymat`, increases all its elements by 0.5, and stores the result in `bb`. Lastly, it prints the length of the resulting list.

Note that `if` statements don't have to match the exact style I'm using here. Some programmers, for example, prefer to open the left brace on a new line after the condition, or some may prefer a different amount of indentation.

10.1.2 else Statements

The `else` statement executes a chunk of code if and only if a defined condition is `FALSE`.

If you want something different to happen when the condition is `FALSE`, you can use the `else` declaration. Here's an example in pseudocode:

```
if (condition)
  do any code in here if condition is TRUE
else
  do any code in here if condition is FALSE
```

do any code in here if condition is **FALSE**

E

You set the condition, then in the first set of braces you place the code to run if the condition is `TRUE`. After this, you declare `else` followed by a new set of braces where you can place code to run if the condition is `FALSE`.

Let's return to the first example in Section 10.1.1, once more storing these values at the console prompt.

```
R> a <- 3
R> mynumber <- 4
```

In the editor, create a new version of the earlier `if` statement.

```
if(a <= mynumber){
  cat("Condition was", a <= mynumber)
a <- a^2
} else {
  cat("Condition was", a <= mynumber)
a <- a-3.5
```

```
}
a
```

Here, you again square `a` if the condition `a<=mynumber` is TRUE, but if FALSE, `a` is overwritten by the result of itself minus 3.5. You also print text to the console stating whether the condition was met. After resetting `mynumber` and to their original values, the first run of the if loop computes `a` as 9, just as earlier, outputting the following:

```
Condition was TRUE
R>a
[1]9
```

Now, immediately highlight and execute the entire statement again. This time around, it will evaluate to FALSE and execute the code after .

```
Condition was FALSE
R>a
[1] 5.5
```

10.1.3 Using ifelse for Element-wise Checks

An if statement can check the condition of only a single logical value. If you pass in, for example, a vector of logicals for the condition, the if statement will only check (and operate based on) the very first element. It will issue a warning saying as much, as the following dummy example shows:

```
R> if(c(FALSE,TRUE,FALSE,TRUE,TRUE)){
Warning message:
In if (c(FALSE, TRUE, FALSE, TRUE, TRUE)) { :
the condition has length > 1 and only the first element will be used
```

If there is, however, a shortcut function available, `ifelse`, which can perform this kind of vector-oriented check in relatively simple cases. To demonstrate how it works, consider the objects `x` and `y` defined as follows:

```
R>x<-5
R>y <- -5:5
R>y
[1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

Now, suppose you want to produce the result of `x/y` but with any instance of Inf (that is, any instance where `x` is divided by zero) replaced with NA. In other words, for each element in `y`, you want to check whether `y` is zero. If so, you want the code to output NA, and if not, it should output the result of `x/y`.

As you've just seen, a simple statement won't work here. Since it accepts only a single logical value, it can't run through the entire logical vector produced by `test`.

```
R> y==0
[1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
ifelse
```

Instead, you can use the element-wise `ifelse` function for this kind of scenario.

```
R> result <- ifelse(test=y==0,yes=NA,no=x/y)
R> result
[1] -1.000000 -1.250000 -1.666667 -2.500000 -5.000000 NA 5.000000 2.500000
[9] 1.666667 1.250000 1.000000
```

Using exact matching, this command creates the desired result vector in one line. Three arguments must be specified: `test` takes a logical-valued data structure, `yes` provides the element to return if the condition is satisfied, and `no` gives the element to return if the condition is FALSE. As noted in the function documentation (which you can access with `?ifelse`), the returned structure will be of “the same length and attributes as `test`.”

Exercise 10.1

- a. Create the following two vectors:

```
vec1 <- c(2,1,1,3,2,1,0)
vec2 <- c(3,8,2,2,0,0,0)
```

Without executing them, determine which of the following if statements would result in the string being printed to the console. Then confirm your answers in R.

- i. `if((vec1[1]+vec2[2])==10){cat("Printme!")}`
 - ii. `if(vec1[1]>=2&&vec2[1]>=2){cat("Printme!")}`
 - iii. `if(all((vec2-vec1)[c(2,6)]<7)){ cat("Print me!") }`
 - iv. `if(!is.na(vec2[3])){ cat("Print me!") }`
- b. Using `vec1` and `vec2` from (a), write and execute a line of code that multiplies the corresponding elements of the two vectors together *if* their sum is greater than 3. Otherwise, the code should simply sum the two elements.

- c. In the editor, write R code that takes a square character matrix and checks *if* any of the character strings on the diagonal (top left to bottom right) begin with the letter *g*, lowercase or uppercase. If satisfied, these specific entries should be overwritten with the string "HERE". Otherwise, the entire matrix should be replaced with an identity matrix of the same dimensions. Then, try your code on the following matrices, checking the result each time:

- i. `mymat <- matrix(as.character(1:16),4,4)`
- ii. `mymat <- matrix(c("DANDELION","Hyacinthus","Gerbera",
"MARIGOLD","geranium","ligularia",
"Pachysandra","SNAPDRAGON","GLADIOLUS"),3,3)`
- iii. `mymat <- matrix(c("GREAT","exercises","right","here"),2,2,
byrow=T)`

Hint: This requires some thought—you will find the functions `diag` from Section 3.2.1 and `substr` from Section 4.2.4 useful.

10.1.4 Nesting and Stacking Statements

An *if* statement can itself be placed within the outcome of another *if* statement. By *nesting* or *stacking* several statements, you can weave intricate paths of decision-making by checking a number of conditions at various stages during execution.

In the editor, modify the example once more as follows:

```
mynumber  
if(a<=mynumber){  
  cat("First condition was TRUE\n")  
  a <- a^2  
  if(mynumber>3){  
    cat("Second condition was TRUE")  
    b <- seq(1,a,length=mynumber)  
  } else {  
    cat("Second condition was FALSE")  
  }  
  b <- a*mynumber  
} else {  
  cat("First condition was FALSE\n")  
  a <- a-3.5  
  if(mynumber>=4){  
    cat("Second condition was TRUE")  
    b <- a^(3-mynumber)  
  } else {  
    cat("Second condition was FALSE")  
  }  
  b <- rep(a+mynumber,times=3)
```

```

    }
  }
a
b

```

Here you see the same initial decision being made as earlier. The value `a` is squared if it's less than or equal to `mynumber`; if not, it has 3.5 subtracted from it. But now there's another if statement within each braced area. If the first condition is satisfied and `a` is squared, you then check whether `mynumber` is greater than 3. If TRUE, `b` is assigned `seq(1,a,length=mynumber)`. If FALSE, `b` is assigned `a*mynumber`.

If the first condition fails and you subtract 3.5 from `a`, then you check a second condition to see whether `mynumber` is greater than or equal to 4. If it is, then `b` becomes `a^(3-mynumber)`. If it's not, `b` becomes `rep(a+mynumber,times=3)`. Note that I've indented the code within each subsequent braced area to

make it easier to see which lines are relevant to each possible decision.

Now, reset `a <- 3` and `mynumber <- 4` either directly in the console or from the editor. When you run the `mynumber` example code, you'll get the following output:

```

First condition was TRUE
Second condition was TRUE
R>a
[1] 9
R>b
[1] 1.000000 3.666667 6.333333 9.000000

```

The result indicates exactly which code was invoked—the first condition and second condition were both TRUE. Trying another run of the same code, after first setting

```

R>a<-6
R> mynumber <- 4

```

you see this output:

```

First condition was FALSE
Second condition was TRUE
R>a
[1] 2.5
R>b
[1] 0.4

```

This time the first condition fails, but the second condition checked inside the `else` statement is TRUE.

Alternatively, you could accomplish the same thing by sequentially *stack-ing* if statements and using a combination of logical expressions in each condition. In the following example, you check for the same four situations,



Simple | Smart | Speed

SCHOOL