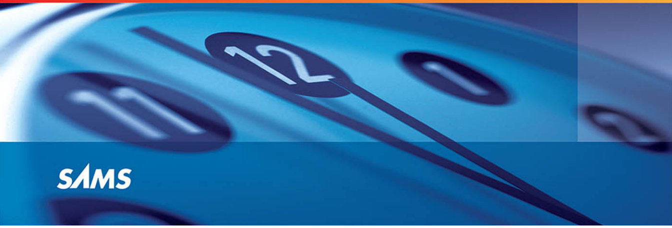
Andy Nicholls
Richard Pugh
Aimee Gott

Sams Teach Yourself R in 24 Hours



Andy Nicholls
Richard Pugh
Aimee Gott

Sams **Teach Yourself**

R in 24 Hours

in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself R in 24 Hours

Copyright © 2016 by Pearson Education, Inc.

ISBN-13: 978-0-672-33848-9
ISBN-10: 0-672-33848-3

Library of Congress Control Number: 2015913320

First Printing December 2015

Editor-in-Chief
Mark Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Elaine Wiley

Copy Editor
Bart Reed

Indexer
Tim Wright

Proofreader
Katie Matejka

Technical Editor
Stephanie Locke

Editorial Assistant
Olivia Basegio

Cover Designer
Mark Shirar

Compositor
Nonie Ratcliff

Contents at a Glance

Table of Contents

Preface

Mango Solutions has been teaching face-to-face R training courses to business professionals and academics alike for over 13 years. In this time, we've seen R grow from its early days as a low cost alternative to S-PLUS and SAS to become the leading analytical programming language in the world today, with several thousand contributors and somewhere upward of a million users. R is widely used throughout academia and is commercially supported by the likes of Microsoft, Google, HP, and Oracle.

In Mango's face-to-face training program we teach R to statisticians, data scientists, physicists, biologists, chemists, geographers, and psychologists among others. All are looking to R to help improve the way they analyze their data in a professional environment. Our aim with this book was to take tried and tested training material and turn it into a lasting resource for anyone looking to learn R for analysis.

Who Should Read This Book?

This book is designed for professional statisticians, data scientists, and analysts looking to widen the scope of analytical tools available to them by learning R. Although it is expected that you might have some programming knowledge in another analytical application or language for data analysis, such as SAS, Python, or Excel/VBA, this is not a prerequisite. This book is suitable for complete novices in programming. From the start, we do not assume any prior knowledge of R; however, those familiar with the basics may find that they can jump straight to later chapters.

What Should You Expect from This Book?

This book is designed to take you from the basics of the R language through common tasks in data science, including data manipulation, visualization, and modeling, to elements of the language that will allow you to produce high-quality, production-ready code. As with our face-to-face training, this book is structured around simple and easy-to-follow examples, all of which are available to download from the book's website (http://www.mango-solutions.com/wp/teach-yourself-r-in-24-hours-book). Throughout, we introduce good practices for writing code as well as provide tips and tricks from our combined experience in R development.

By the end of this book, you should have a good understanding of the fundamentals of R as well as many of the most commonly used packages. You should have a good understanding of what makes well written R code and how to implement this yourself.

How Is This Book Organized?

This book is designed to guide you through everything you need to know to get started with the R language and then introduce additional elements of the language for specific tasks.

The following is an outline of each of the hours and what to expect:

Hour 1, "The R Community"—In this hour, we start by looking at how R evolved from the S language to become the all-purpose data science programming language that it is today. The R community offers a plethora of help and support options for users. We look at some of the better-known options during this hour.

Hour 2, "The R Environment"—In this hour, we start a new R session via RStudio, type some basic commands, and explore the idea of an R "object." You will be more formally introduced to the concept of an R package.

Hour 3, "Single-Mode Data Structures"—In this hour, we describe the standard types of data found in R and introduce three key structures that can be used to store these data types: vectors, matrices, and arrays. We illustrate the ways in which these structures can be created and manage these data structures with a focus on how we can extract data from them.

Hour 4, "Multi-Mode Data Structures"—The majority of data sources contain a mixture of data types, which we need to store together in a simple, effective format. In this hour, we focus on two key data structures that allow us to store "multi-mode" data: lists and data frames. We illustrate the ways in which these structures can be created and manage these data structures with a focus on how we can extract data from them. We also look at how these two data structures can be effectively used in our day-to-day work.

Hour 5, "Dates, Times, and Factors"—In this hour, you learn more about some of the special data types in R that enable us to work with dates and times and with categorical data.

Hour 6, "Common R Utility Functions"—In this hour, we introduce you to some of the most common utility functions in R that you will find yourself using every day.

Hour 7, "Writing Functions: Part I"—One of the strengths of R is that we can extend it by writing our own functions, allowing us to create utilities that can perform a variety of tasks. In this hour, we look at ways in which we can create our own functions, specify inputs, and return results to the user. We also discuss the "if/else" structure in R and use it to control the flow of code within a function.

Hour 8, "Writing Functions: Part II"—This hour looks at a range of advanced function-writing topics, such as returning error messaging, checking whether inputs are appropriate to our functions, and the use of function "ellipses."

Hour 9, "Loops and Summaries"—In this hour, you see how we can apply simple functions and code in a more "applied" fashion. This allows us to perform tasks repeatedly over sections of our data without the need to produce verbose, repetitive code.

Hour 10, "Importing and Exporting"—In this hour, we introduce common methods for importing and exporting data. By the end of the hour you will have seen how R can be used to read and write flat files and connect to database management systems (DBMSs) as well as Microsoft Excel.

Hour 11, "Data Manipulation and Transformation"—As data scientists and statisticians, we rarely get to control the structure and format of our data. Now we will look a little closer at the structure of our data. Several approaches to data manipulation in R have evolved over time. In this hour, we start by looking at what could be called "traditional" approaches to the data manipulation tasks of sorting, setting, and merging. We then look at the popular packages **reshape**, **reshape2**, and **tidyr** for data restructuring.

Hour 12, "Efficient Data Handling in R"—We begin the hour by looking at the incredibly popular **dplyr** package. The **data.table** package is a standalone package for data manipulation that offers greater efficiency for very large data.

Hour 13, "Graphics"—After all the manipulations to our data, we want to be able to start to do something with it. In this hour, we look at how we can create graphics using the base graphics functionality, including how to send your graphics to devices such as a PDF and the standard graphics functions. We finally look at how to control the layout of graphics on the page.

Hour 14, "The ggplot2 Package for Graphics"—In this hour, we look at the hugely popular **ggplot2** package, developed by Hadley Wickham for creating high-quality graphics.

Hour 15, "Lattice Graphics"—Here we will look at a third way of creating graphics: using the **lattice** package. This graphic system is well suited to graphing highly grouped data, with the code designed to closely resemble the modeling capabilities of R.

Hour 16, "Introduction to R Models and Object Orientation"—In this hour, we see how to fit a simple linear model and assess its performance using a range of textual and graphical methods. Beyond this, we introduce "object orientation" and see how the R statistical modeling framework is built on this concept.

Hour 17, "Common R Models"—In this hour, we extend the ideas of the previous hour to other modeling approaches. Specifically, we look at Generalized Linear Models, nonlinear models, time series models, and survival models.

Hour 18, "Code Efficiency"—In this hour, we look at some of the techniques we can use to improve the efficiency and, importantly, the professionalism of our R code.

Hour 19, "Package Building"—When we put our code into a package, it forces us to ensure that our code is of a high standard and we are adhering to good practices, such as documenting our code. We focus here on making sure our code is well written and documented, the starting point for high-quality, professional code that is easy to share and reuse.

Hour 20, "Advanced Package Building"—There are a number of ways we can extend a package to make it more robust to changes and easier for users to get started with. You learn the most common of these extra components in this hour.

Hour 21, "Writing R Classes"—In this hour, we take a general look at some key features of object-oriented programming before focusing in on R's S3 implementation.

Hour 22, "Formal Class Systems"—During this hour, we look at the more formal S4 and Reference Class systems in R. Along the way, you will be introduced to concepts such as validity checking, multiple dispatch, message-passing object orientation, and mutable objects.

Hour 23, "Dynamic Reporting"—Up to this point we have seen the fundamentals of the R language as well as the aspects of R that allow us to ensure that we write high-quality, well-documented, and easily shareable code. In this hour, we take a look at one of the ways you can extend your use of R, specifically for simplifying the generation of reports that rely heavily on R-generated output.

Hour 24, "Building Web Applications with Shiny"—Although you may initially be put off by the idea of building a web application, we introduce a package that allows you to generate web applications entirely in R, writing only R code. This is currently one of the most popular packages available in R, with more and more packages being added to CRAN that use this framework.

About the Sample Code

Throughout this book, we have included examples of the concepts that are being introduced. You may notice that the code is prefixed with the symbols ">" and "+". These are the

R prompt and continuation characters and do not need to be entered when writing code. We have used the formatting conventions of `function` for a function name and **package** for a package name.

All of the code examples included in this book are available from our web page: http://www.mango-solutions.com/wp/teach-yourself-r-in-24-hours-book/

NOTE

Code-Continuation Arrows and Listing Line NumbersYou might see code-continuation arrows (➥) occasionally in this book to indicate when a line of code is too long to fit on the printed page. Also, some listings have line numbers and some do not. The listings that have line numbers have them so that we can reference code by line; the listings that do not have line numbers are not referenced by line.

Contacting the Authors

If you have any comments or questions about this book, please drop us an email at rin24hours@mango-solutions.com.

About the Authors

Andy Nicholls has a Master of Mathematics degree from the University of Bath and Master of Science in Statistics with Applications in Medicine from the University of Southampton. Andy worked as a Senior Statistician in the pharmaceutical industry for a number of years before joining Mango Solutions as an R consultant in 2011. Since joining Mango, Andy has taught more than 50 on-site R training courses and has been involved in the development of more than 30 R packages. Today, he manages Mango Solution's R consultancy team and continues to be a regular contributor to the quarterly LondonR events, by far the largest R user group in the UK, with over 1,000 meet-up members. Andy lives near the historical city of Bath, UK with his wonderful, tolerant wife and son.

Richard Pugh has a first-class Mathematics degree from the University of Bath. Richard worked as a statistician in the pharmaceutical industry before joining Insightful, the developers of S-PLUS, joining the pre-sales consulting team. Richard's role at Insightful included a variety of activities, providing a range of training and consulting services to blue-chip customers across many sectors. In 2002, Richard co-founded Mango Solutions, developing the company and leading technical efforts around R and other analytic software. Richard is now Mango's Chief Data Scientist and speaks regularly at data science and R events. Richard lives in Bradford on Avon, UK with his wife and two kids, and spends most of his "spare" (ha!) time renovating his house.

Aimee Gott has a PhD in Statistics from Lancaster University where she also completed her undergraduate and master's degrees. As Training Lead, Aimee has delivered over 200 days of training for Mango. She has delivered on-site training courses in Europe and the U.S. in all aspects of R, as well as shorter workshops and online webinars. Aimee oversees Mango's training course development across the data science pipeline, and regularly attends R user groups and meet-ups. In her spare time, Aimee enjoys learning European languages and documenting her travels through photography.

Dedications

This book is dedicated to my wife, for her love and support and for putting up with losing our summer to all the late nights, and to my baby boy who learnt to sit up, eat, crawl, and walk whilst this book was being written! —Andy Nicholls

This book is dedicated to my family for having to put up with me writing the book at weekends. —Richard Pugh

To Stephen, Carol, Richard, and Kirstie. —Aimee Gott

Acknowledgments

Throughout the process of writing, many people have taken time to assist us, guide us, and shape this book. First of all, thanks go to Andy Miskell, Jeff Stagg, Mike K. Smith, and Susan Duke, who took the time to review our initial outline. Thanks also go to the consultancy team at Mango and Stephen Kaluzny of TIBCO who were all able to answer our questions while writing.

We would also like to thank all those who have been involved in the production of this book. In particular we would like to thank Elaine Wiley (production editor), Trina MacDonald (acquisitions editor), Songlin Qiu (development editor), Olivia Basegio (publisher assistant), Stephanie Locke (technical editor), Bart Reed (copyeditor), and Katie Matejka (proofreader).

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: errata@informit.com

Mail: Addison-Wesley/Prentice Hall Publishing
ATTN: Reader Feedback
330 Hudson Street
7th Floor
New York, New York, 10013

Reader Services

Register your copy of Teach Yourself R in 24 Hours at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account.* Enter the product ISBN, 9780672338489, and click Submit. Once the process is complete, you will find any available bonus content under Registered Products.

*Be sure to check the box that you would like to hear from us in order to receive exclusive discounts on future editions of this product.

This page intentionally left blank

HOUR 4
Multi-Mode Data Structures

What You'll Learn in This Hour:

- What a list object is
- How to create and manipulate a data frame
- How to perform an initial investigation in the structure of our data

The majority of data sources contain a mixture of data types, which we need to store together in a simple, effective format. The "single-mode" structures introduced in the last hour are useful basic data objects, but are not sufficiently sophisticated to store data containing multiple "modes." In this hour, we focus on two key data structures that allow us to store "multi-mode" data: lists and data frames. We will illustrate the ways in which these structures can be created and managed, with a focus on how to extract data from them. We also look at how these two data structures can be effectively used in our day-to-day work.

Multi-Mode Structures

In the last hour, we examined the three structures designed to hold data in R:

- Vectors—Series of values
- Matrices—Rectangular structures with rows and columns
- Arrays—Higher dimension structures (for example, 3D and 4D arrays)

Although these objects provide us with a range of useful functionality, they are restricted in that they can only hold a single "mode" of data. This is illustrated in the following example:

```
> c(1, 2, 3, "Hello")                 # Multiple modes
[1] "1"     "2"     "3"     "Hello"
> c(1, 2, 3, TRUE, FALSE)             # Multiple modes
[1] 1 2 3 1 0
> c(1, 2, 3, TRUE, FALSE, "Hello")    # Multiple modes
[1] "1"     "2"     "3"     "TRUE"  "FALSE" "Hello"
```

As you can see, when we attempt to store more than one mode of data in a single-mode structure, the object (and its contents) will be converted to a single mode.

The preceding example uses a vector to illustrate this behavior, but let's suppose we want to store a rectangular "dataset" using a matrix. For example, we might attempt to create a matrix that contains the forecast temperatures for New York over the next five days:

```
> weather <- cbind(
+   Day   = c("Saturday", "Sunday", "Monday", "Tuesday", "Wednesday"),
+   Date  = c("Jul 4", "Jul 5", "Jul 6", "Jul 7", "Jul 8"),
+   TempF = c(75, 86, 83, 83, 87)
+ )
> weather
     Day         Date    TempF
[1,] "Saturday"  "Jul 4" "75"
[2,] "Sunday"    "Jul 5" "86"
[3,] "Monday"    "Jul 6" "83"
[4,] "Tuesday"   "Jul 7" "83"
[5,] "Wednesday" "Jul 8" "87"
```

From the quotation marks, it is clear that R has converted all the data to character values, which can be confirmed by looking at the mode of this matrix structure:

```
> mode(weather)    # The mode of the matrix
[1] "character"
```

This reinforces the need for data structures that allow us to store data of multiple modes. R provides two "multi-mode" data structures:

- Lists—Containers for any objects
- Data frames—Rectangular structures with rows and columns

Lists

The list is considered perhaps the most complex data object in R, and many R programmers will go to great lengths to avoid the use of lists in their structures. This perceived complexity, perhaps, stems from a lack of clarity over what a list "looks like." Other structures, such as vectors and matrices, are relatively easy to visualize, and are therefore easier to adopt and manage.

Despite this, lists are simple structures that can be used to perform a number of complex operations.

What Is a List?

Lists are simply containers for other objects. The objects stored in a list can be of any type (for example, “matrix” or “vector”) and any mode. Therefore, you can create a list containing the following, for example:

- A character vector
- A numeric matrix
- A logical array
- Another list

When discussing lists, some people use the analogy of a box. For example, you might do the following:

- Create an empty box.
- Put some “things” into the box.
- Look into the box to see what things are in there.
- Take things back out of the box.

In a similar way, in this section, we will look at how to do the following:

- Create an empty list.
- Put objects into the list.
- Look at the number (and names) of objects in the list.
- Extract elements from the list.

Creating an Empty List

You create a list using the `list` function. The simplest list you can create is an empty list, like this:

```
> emptyList <- list()
> emptyList
list()
```

Later, you will see how to add elements to this empty list.

Creating a Non-Empty List

More commonly, you'll create a list and add initial elements to it at the same time. You achieve this by specifying a comma-separated set of objects within the `list` function:

```
> aVector <- c(5, 7, 8, 2, 4, 3, 9, 0, 1, 2)
> aMatrix <- matrix( LETTERS[1:6], nrow = 3)
> unnamedList <- list(aVector, aMatrix)
> unnamedList
[[1]]
 [1] 5 7 8 2 4 3 9 0 1 2

[[2]]
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

In this example, we created two objects (`aVector` and `aMatrix`) and then created a list (`unnamedList`) containing copies of these objects.

NOTE

Original Objects

When you create lists in this way, you take copies of the objects (`aVector` and `aMatrix` in this example). The original objects are not impacted by this action (that is, they are not edited, moved, changed, or deleted).

If you only need the objects within the list, you could create the objects as you specify the list, like this:

```
> unnamedList <- list(c(5, 7, 8, 2, 4, 3, 9, 0, 1, 2),
+                       matrix( LETTERS[1:6], nrow = 3))
> unnamedList
[[1]]
 [1] 5 7 8 2 4 3 9 0 1 2

[[2]]
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

Creating a List with Element Names

When you create a list, you can optionally assign names to the elements. This helps you when you're referencing elements in the list later.

```
> namedList <- list(VEC = aVector, MAT = aMatrix)
> namedList
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

As before, you can also create the (named) objects as you're creating the list:

```
> namedList <- list(VEC = c(5, 7, 8, 2, 4, 3, 9, 0, 1, 2),
+                   MAT = matrix( LETTERS[1:6], nrow = 3))
> namedList
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

Creating a List: A Summary

You have now seen a few different ways of creating a list. It is worth recapping the ways in which we created the lists with some code examples:

```
> # Create an empty list
> emptyList <- list()

> # 2 Ways of Creating an unnamed list containing a vector and a matrix
> unnamedList <- list(aVector, aMatrix)
> unnamedList <- list(c(5, 7, 8, 2, 4, 3, 9, 0, 1, 2),
+                     matrix( LETTERS[1:6], nrow = 3))

> # 2 Ways of Creating a named list containing a vector and a matrix
> namedList <- list(VEC = aVector, MAT = aMatrix)
> namedList <- list(VEC = c(5, 7, 8, 2, 4, 3, 9, 0, 1, 2),
+                  MAT = matrix( LETTERS[1:6], nrow = 3))
```

In these examples, we created three lists that we will use as examples over the next few sections:

```
> emptyList         # An empty list
list()

> unnamedList       # A list with unnamed elements
[[1]]
 [1] 5 7 8 2 4 3 9 0 1 2

[[2]]
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"

> namedList         # A list with element names
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

NOTE

Printing Style

Notice the difference in printing when a list has element names versus when there are no element names: Elements are indexed with double square brackets (for example, `[[1]]`) for “unnamed” lists, and with dollar symbols (for example, `$VEC`) for “named” lists. This gives you a hint as to how you’ll be able to reference the elements of a list later.

List Attributes

As with single-mode structures, a set of functions allows you to query some of the list attributes. Specifically, you can use the `length` function to query the number of elements in the list, and the `names` function to return the element names.

The `length` function returns the number of elements in the list, as shown here:

```
> length(emptyList)
[1] 0
> length(unnamedList)
[1] 2
> length(namedList)
[1] 2
```

The names function returns the names of the elements in the list, or NULL if there are no elements or no element names assigned:

```
> names(emptyList)
NULL
> names(unnamedList)
NULL
> names(namedList)
[1] "VEC" "MAT"
```

With single-mode data structures, we additionally used the mode function to return the type of data they held. Because lists are multi-mode structures, there is no longer a single mode of data being stored, so the word “list” is returned:

```
> mode(emptyList)
[1] "list"
> mode(unnamedList)
[1] "list"
> mode(namedList)
[1] "list"
```

Subscripting Lists

Two types of list subscripting can be performed:

- You can create a subset of the list, returning a shorter list.
- You can reference a single element within the list.

Subsetting the List

You can use square brackets to select a subset of an existing list. The return object will itself be a list.

```
LIST [ Input specifying the subset of list to return ]
```

As with vectors, you can put one of five input types in the square brackets, as shown in Table 4.1.

TABLE 4.1 Possible List Subscripting Inputs

Input	Effect
Blank	All values of the list are returned.
A vector of positive integers	Used as an index of list elements to return.
A vector of negative integers	Used as an index of list elements to omit.

Input	Effect
A vector of logical values	Only corresponding `TRUE` elements are returned.
A vector of character values	Refers to the names of elements to return.

To illustrate the subsetting of lists, we will use the `namedList` object created earlier.

Blank Subscripts

If you use a blank subscript, the whole of the list is returned:

```
> namedList [ ]  # Blank subscript
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

Positive Integer Subscripts

If you use a vector of positive integers, it is used as an index of elements to return:

```
> subList <- namedList [ 1 ]    # Return first element
> subList                       # Print the new object
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

> length(subList)               # Number of elements in the list
[1] 1
> class(subList)                # Check the "class" of the object
[1] "list"
```

As you can see from this example, the return object (saved as `subList` here) is itself a list. You can also use the `class` function to check the type of object, and it confirms `subList` is a list object.

NOTE

An Object's Class

This is the first time in this book you've seen the `class` function used. It returns the type of objects, whereas the `mode` function returns the type of data held in an object. Let's illustrate this distinction with a numeric matrix:

```
> aMatrix <- matrix(1:6, nrow = 2)      # Create a numeric matrix
> aMatrix                               # Print the matrix
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> mode(aMatrix)                         # Mode of data held in this object
[1] "numeric"

> class(aMatrix)                        # Type (or "class") of object
[1] "matrix"
```

Negative Integer Subscripts

You can provide a vector of negative integers to specify the index of list elements to omit:

```
> namedList
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
> namedList [ -1 ]   # Return all but the first element
$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

Logical Value Subscripts

You can provide a vector of logical integers to specify the list elements to return and omit:

```
> namedList
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"

> namedList [ c(T, F) ]  # Vector of logical values
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2
```

Character Value Subscripts

If your list has element names, you can provide a vector of character values to identify the (named) elements you wish to return:

```
> namedList
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"

> namedList [ "MAT" ]         # Vector of Character values
$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"
```

Reference List Elements

In the last section, you saw that you can reference a list using square brackets to "subset" the list (that is, return a list containing only a subset of the original elements). More commonly, you'll want to reference a specific element within your list.

You can reference elements of a list in two ways:

- You can use "double" square brackets.
- If there are element names, you can use the $ symbol.

Double Square Bracket Referencing

You can directly reference an element of a list using double square brackets. Although there are a number of uses of the double square brackets, the most common use is to supply a single integer index to refer to the element to extract:

```
> namedList              # The original list
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
```

```
[2,] "B"  "E"
[3,] "C"  "F"

> namedList[[1]]         # The first element
 [1] 5 7 8 2 4 3 9 0 1 2
> namedList[[2]]         # The second element
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"

> mode(namedList[[2]])   # The mode of the second element
[1] "character"
```

When you use double square brackets in this way, you are directly referencing the objects contained within the list, as supported by the result of the `mode` function call. This is in contrast to the use of the single square bracket earlier, where we extracted a subset of the list itself:

```
> namedList [1]          # Return a list containing 1 element
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

> namedList [[1]]        # Return the first element of the list (a vector)
 [1] 5 7 8 2 4 3 9 0 1 2
```

Referencing Named Elements with $

If the elements of your list are named, you can use the `$` symbol to directly reference them. As such, the following lines of code are equivalent ways of referencing the first (the "VEC") element of our `namedList` object:

```
> namedList              # Print the original list
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"

> namedList[[1]]         # Return the first element
 [1] 5 7 8 2 4 3 9 0 1 2
> namedList$VEC          # Return the "VEC" element
 [1] 5 7 8 2 4 3 9 0 1 2
```

Double Square Brackets versus $

The $ symbol provides a more intuitive way of referencing named list elements, which is also more aesthetically pleasing than the use of double square brackets. We tend to use double square brackets when there are no element names assigned, and use $ when names exist. Here's an example:

```
> unnamedList        # List with no element names
[[1]]
 [1] 5 7 8 2 4 3 9 0 1 2

[[2]]
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"

> unnamedList[[1]]   # First element
 [1] 5 7 8 2 4 3 9 0 1 2

> namedList          # List with element names
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2]
[1,] "A"  "D"
[2,] "B"  "E"
[3,] "C"  "F"

> namedList$VEC      # The "VEC" element
 [1] 5 7 8 2 4 3 9 0 1 2
```

TIP

Shortened $ Referencing

When you use the $ symbol, you only need to provide enough of the name so that R understands which element you are referring to. This is illustrated in the following example:

```
> aList <- list( first = 1, second = 2, third = 3, fourth = 4 )
> aList$s   # Returns the second
[1] 2
> aList$fi  # Returns the first
[1] 1
```

```
> aList$fo  # Returns the fourth
[1] 4
```

Although it is possible to use shortened referencing in this way, it can lead to less maintainable and readable code, and should be avoided where possible when creating scripts.

Adding List Elements

You can add elements to a list in one of two ways:

- By directly adding an element with a specific name or in a specific position
- By combing lists together

Directly Adding a List Element

You can add a single element to a list by assigning it into a specific index or name. The syntax mirrors that of the "Double Square Brackets versus $" section earlier. For example, let's add a single element to our empty list:

```
> emptyList                        # Empty list
[[1]]
[1] "A" "B" "C" "D" "E"

> emptyList[[1]] <- LETTERS[1:5]   # Add an element

> emptyList                        # Updated (non)empty list
[[1]]
[1] "A" "B" "C" "D" "E"
```

Instead of using the double square brackets, we can use the `$` symbol to add a "named" element to a list:

```
> emptyList <- list()              # Recreate the empty list
> emptyList                        # Empty list
list()
> emptyList$ABC <- LETTERS[1:5]    # Add an element
> emptyList                        # Updated (non)empty list
$ABC
[1] "A" "B" "C" "D" "E"
```

NOTE

Adding Nonconsecutive Elements

The preceding examples uses either square brackets or the `$` symbol to add elements to the "first" position of an empty list. If we add an element to a later index, R interpolates a number of `NULL` elements to fill any gaps in the list:

```
> emptyList <- list()                # Recreate the empty list
> emptyList                          # Empty list
list()
> emptyList[[3]] <- "Hello"          # Assign to third element
> emptyList
[[1]]
NULL

[[2]]
NULL

[[3]]
[1] "Hello"
```

Combining Lists

You can grow lists by combining them together using the `c` function, as shown here:

```
> list1 <- list(A = 1, B = 2)   # Create list1
> list2 <- list(C = 3, D = 4)   # Create list2
> c(list1, list2)               # Combine the lists
$A
[1] 1

$B
[1] 2

$C
[1] 3

$D
[1] 4
```

A Summary of List Syntax

As you have seen so far in this hour, the way we use lists varies slightly based on whether the elements of the list are named. At this point, it is worth reviewing the syntax to create and manage "unnamed" and "named" list structures.

Overview of Unnamed Lists

An overview of the key syntax covered is shown here, using a list without named elements as an example. First, let's create a list and look at the list attributes:

```
> unnamedList <- list(aVector, aMatrix)    # Create the list

> unnamedList                              # Print the list
[[1]]
 [1] 5 7 8 2 4 3 9 0 1 2

[[2]]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> length(unnamedList)                      # Number of elements
[1] 2

> names(unnamedList)                       # No element names
NULL
```

We can subset the list or extract list elements using single/double square brackets:

```
> unnamedList[1]                           # Subset the list
[[1]]
 [1] 5 7 8 2 4 3 9 0 1 2

> unnamedList[[1]]                         # Return the first element
 [1] 5 7 8 2 4 3 9 0 1 2

> unnamedList[[3]] <- 1:5                  # Add a new element

> unnamedList
[[1]]
 [1] 5 7 8 2 4 3 9 0 1 2

[[2]]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

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

Overview of Named Lists

Let's look at a similar example using a list with element names. First, let's create the list and view the list attributes:

```
> namedList <- list(VEC = aVector, MAT = aMatrix)   # Create the list

> namedList                                         # Print the list
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> length(namedList)                     # Number of elements
[1] 2

> names(namedList)                      # Element names
[1] "VEC" "MAT"
```

We can subset the list using single square brackets, or reference elements directly with the $ symbol:

```
> namedList[1]                          # Subset the list
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

> namedList$VEC                         # Return the first element
 [1] 5 7 8 2 4 3 9 0 1 2

> namedList$NEW <- 1:5                  # Add a new element

> namedList
$VEC
 [1] 5 7 8 2 4 3 9 0 1 2

$MAT
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

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

Motivation for Lists

A good understanding of lists helps you to accomplish a number of useful tasks in R. To illustrate this, we will briefly look at two use cases that rely on list structures. Note that this section includes syntax that will be covered later in this book, but we include it here to illustrate "the art of the possible" at this stage.

Flexible Simulation

Consider a situation where we want to simulate a number of extreme values (for example, large financial losses by day, or particularly high values of some measure for each patient in a drug study). For each iteration, we may simulate any number of numeric values from a given distribution.

A list provides a flexible structure to hold all the simulated data. Consider the following code example:

```
> nExtremes <- rpois(100, 3)               # Simulate number of extreme values by
                                             day from a Poisson distribution
> nExtremes[1:5]                           # First 5 numbers
[1] 0 3 5 7 3

> # Define function that simulates "N" extreme values
> exFun <- function(N) round(rweibull(N, shape = 5, scale = 1000))
> extremeValues <- lapply(nExtremes, exFun) # Apply the function to our simulated
                                              numbers

> extremeValues[1:5]                       # First 5 simulated outputs
[[1]]
numeric(0)

[[2]]
[1] 1305  948 1077

[[3]]
[1] 676 516 865 614 970

[[4]]
[1]  618 1217  818 1173 1205 1105  519

[[5]]
[1] 1026  933  657
```

From this example, note that the first simulated output generated no "extreme" values, resulting in the output containing an empty numeric vector (signified by `numeric(0)`). The "unnamed" list structure allows us to hold, in the same structure:

- This empty vector (indicating no "extreme values" for a particular day)
- Large vectors holding a number of simulated outputs (for days where many "extreme values" were simulated)

Given that we have stored this information in a list, we can query it to summarize the average number and average of extreme values:

```
> median(sapply(extremeValues, length))     # Average number of simulated extremes
[1] 3
> median(sapply(extremeValues, sum))        # Average extreme value
[1] 2634
```

TIP

The apply Functions

In the preceding examples, we used functions such as `lapply` (which applies a function to each element of a list) and `sapply` (which performs the same action but simplifies the outputs). We cover the apply family of functions later in Hour 9, "Loops and Summaries."

Extracting Elements from Named Lists

In R, most objects are, fundamentally, lists. For example, let's use the `t.test` function to perform a simple T-test. We will take the example straight from the `t.test` help file:

```
> theTest <- t.test(1:10, y = c(7:20))      # Perform a T-Test
> theTest                                   # Print the output

          Welch Two Sample t-test

data:  1:10 and c(7:20)
t = -5.4349, df = 21.982, p-value = 1.855e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -11.052802  -4.947198
sample estimates:
mean of x mean of y
      5.5      13.5
```

The output is printed as a nicely formatted text summary informing us of the significant T-test. But what if we wanted to use one of the elements of this output in further work (for example, the p-value). Consulting the help file, we see the return value is described as follows:

Value

A list with class `htest` containing the following components:

- **`statistic`** The value of the t-statistic.
- **`parameter`** The degrees of freedom for the t-statistic.
- **`p.value`** The p-value for the test.
- **`conf.int`** A confidence interval for the mean appropriate to the specified alternative hypothesis.

- **`estimate`** The estimated mean or difference in means, depending on whether it was a one-sample test or a two-sample test.
- **`null.value`** The specified hypothesized value of the mean or mean difference, depending on whether it was a one-sample test or a two-sample test.
- **`alternative`** A character string describing the alternative hypothesis.
- **`method`** A character string indicating what type of t-test was performed.
- **`data.name`** A character string giving the name(s) of the data.

The key thing to note here is that the return object is "a list." Given that the output is a list, we can query the named elements of this list and see that the result matches the description of elements in the help file:

```
> names(theTest)        # Names of list elements
[1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
[6] "null.value"  "alternative" "method"      "data.name"
```

Given that this is a named list, and we know the names of the elements, we can use the `$` symbol to directly reference the information we need:

```
> theTest$p.value       # Reference the p-value
[1] 1.855282e-05
```

Using this approach, we can reference a wide range of elements from R outputs.

NOTE

Print Methods

In the preceding example, we created a complex object (fundamentally, a named list) that printed in a neat manner:

```
> theTest                                    # Print the output

          Welch Two Sample t-test

data:  1:10 and c(7:20)
t = -5.4349, df = 21.982, p-value = 1.855e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -11.052802  -4.947198
sample estimates:
mean of x mean of y
      5.5      13.5
```

The neat printout is generated by a print "method" associated with outputs from `t.test`. If we want to see the "raw" underlying structure, we can use the `print.default` function, which confirms that the structure is list based:

```
> print.default(theTest)
$statistic
       t
-5.43493

$parameter
      df
21.98221

$p.value
[1] 1.855282e-05
...
```

Data Frames

In the last section, we introduced the "list" structure, which allows you to store a set of objects of any mode. A data frame is, like many R objects, a named list. However, a data frame enforces a number of constraints on this named list structure. In particular, a data frame is constrained to be a named list that can only hold vectors of the same length.

Creating a Data Frame

We create a data frame by specifying a set of named vectors to the `data.frame`. For example, let's create a data frame containing New York temperature forecasts over the next five days:

```
> weather <- data.frame(                  # Create a data frame
+   Day   = c("Saturday", "Sunday", "Monday", "Tuesday", "Wednesday"),
+   Date  = c("Jul 4", "Jul 5", "Jul 6", "Jul 7", "Jul 8"),
+   TempF = c(75, 86, 83, 83, 87)
+ )
> weather                                 # Print the data frame
        Day  Date TempF
1  Saturday Jul 4    75
2    Sunday Jul 5    86
3    Monday Jul 6    83
4   Tuesday Jul 7    83
5 Wednesday Jul 8    87
```

NOTE

Print Methods

As discussed earlier, the neat printing of this object is caused by a print "method" for data frames. We can see the raw structure using `print.default`, which again confirms that a data frame is fundamentally a named list of vectors:

```
> print.default(weather)
$Day
[1] Saturday  Sunday    Monday    Tuesday   Wednesday
Levels: Monday Saturday Sunday Tuesday Wednesday

$Date
[1] Jul 4 Jul 5 Jul 6 Jul 7 Jul 8
Levels: Jul 4 Jul 5 Jul 6 Jul 7 Jul 8

$TempF
[1] 75 86 83 83 87

attr(,"class")
[1] "data.frame"
```

CAUTION

Nonmatching Vector Lengths

If we try to create a data frame using vectors with nonmatching lengths, we get an error message:

```
> data.frame(X = 1:5, Y = 1:2)
Error in data.frame(X = 1:5, Y = 1:2) :
  arguments imply differing number of rows: 5, 2
```

Querying Data Frame Attributes

Because a data frame is simply a named list, the functions we used to query list attributes will work the same way:

- The `length` function returns the number of elements of the list (that is, the number of columns).
- The `names` function returns the element (column) names.

The following example illustrates the use of these functions:

```
> length(weather)           # Number of columns
[1] 3
> names(weather)            # Column names
[1] "Day"   "Date"   "TempF"
```

Selecting Columns from the Data Frame

As with lists, we can reference a single element (vector) from our data frame using either double squared brackets or the `$` symbol:

```
> weather         # The whole data frame
        Day  Date TempF
1  Saturday Jul 4    75
2    Sunday Jul 5    86
3    Monday Jul 6    83
4   Tuesday Jul 7    83
5 Wednesday Jul 8    87

> weather[[3]]    # The "third" column
[1] 75 86 83 83 87
> weather$TempF   # The "TempF" column
[1] 75 86 83 83 87
```

Selecting Columns from the Data Frame

Because we can reference columns in this way, we can also use these approaches to add new columns. For example, let's add a new column called `TempC` to our data containing the temperature in degrees Celsius:

```
> weather$TempC <- round( (weather$TempF - 32) * 5/9 )
> weather
        Day  Date TempF TempC
1  Saturday Jul 4    75    24
2    Sunday Jul 5    86    30
3    Monday Jul 6    83    28
4   Tuesday Jul 7    83    28
5 Wednesday Jul 8    87    31
```

Subscripting Columns

Because the columns of data frames are vectors, we can subscript them using the approaches from Hour 3, "Single-Mode Data Structures." Specifically, we can subscript the columns using square brackets:

```
DATA$COLUMN [ Input specifying the subset to return ]
```

As before, we can reference using blank, positive, negative, or logical inputs. Character inputs do not make sense for referencing columns because the individual elements within columns are not associated with element names.

Blank, Positive, and Negative Subscripts

If we use a blank subscript, all the values of the vector are returned:

```
> weather
        Day  Date TempF TempC
1  Saturday Jul 4    75    24
2    Sunday Jul 5    86    30
3    Monday Jul 6    83    28
4   Tuesday Jul 7    83    28
5 Wednesday Jul 8    87    31

> weather$TempF [ ]  # All values of TempF column
[1] 75 86 83 83 87
```

If we use a vector of positive integers, it refers to the elements of the column (vector) to return:

```
> weather$TempF [ 1:3 ]  # First 3 values of the TempF column
[1] 75 86 83
```

If we use a vector of negative integers, it refers to the elements of the column (vector) to omit:

```
> weather$TempF [ -(1:3) ]  # Omit the first 3 values of the TempF column
[1] 83 87
```

Logical Subscripts

As you saw in the last hour, we can provide a vector of logical values to reference a vector, and only the corresponding `TRUE` values are returned. Here's an example:

```
> weather$TempF
[1] 75 86 83 83 87
> weather$TempF [ c(F, T, F, F, T) ]    # Logical subscript
[1] 86 87
```

Of course, we usually generate the logical vector with a logical statement involving a vector. For example, we could return all the `TempF` values greater than 85 using this statement:

```
> weather$TempF [ weather$TempF > 85 ]  # Logical subscript
[1] 86 87
```

Instead, we could reference a column of a data frame based on logical statements involving one or more other columns (because all columns are constrained to be the same length):

```
> weather$Day [ weather$TempF > 85 ]    # Logical subscript
[1] Sunday    Wednesday
Levels: Monday Saturday Sunday Tuesday Wednesday
```

NOTE

Factor Levels

In the last example, you can see that the days where the forecast is greater than 85°F are Sunday and Wednesday. However, you should note two things about the output:

- There are no quotation marks around the returned values (Sunday and Wednesday).
- Additional "Levels" information has been printed.

This strange output is produced because, when you create a data frame using character columns, those columns are converted to "factors," which are "category" columns that are automatically derived from character vectors when used in a data frame. You'll see more on factors later in Hour 5, "Dates, Times, and Factors."

Referencing as a Matrix

Although a data frame is structured as a named list, its rectangular output is more similar to the matrix structure you saw earlier. As such, R allows us to reference the data frame as if it was a matrix.

Matrix Dimensions

Because we can treat a data frame as a matrix, we can use the `nrow` and `ncol` functions to return the number of rows and columns:

```
> nrow(weather)   # Number of rows
[1] 5
> ncol(weather)   # Number of columns
[1] 4
```

Subscripting as a Matrix

In Hour 3, you saw that you can subscript a matrix using square brackets and two inputs (one for the rows, one for the columns). We can use the same approach to subscript a data frame, where each input can be one of the standard five input types:

```
DATA.FRAME [ Rows to return , Columns to return]
```

Blanks, Positives, and Negatives

We can use blank subscripts to return all rows and columns from a data frame:

```
> weather[ , ]           # Blank, Blank
        Day  Date TempF TempC
1  Saturday Jul 4    75    24
2    Sunday Jul 5    86    30
3    Monday Jul 6    83    28
4   Tuesday Jul 7    83    28
5 Wednesday Jul 8    87    31
```

If we use vectors of positive integers, they are used to provide an index of the rows/columns to return. This example uses positive integers to return the first four rows and the first three columns:

```
> weather[ 1:4, 1:3 ]    # +ive, +ive
       Day  Date TempF
1 Saturday Jul 4    75
2   Sunday Jul 5    86
3   Monday Jul 6    83
4  Tuesday Jul 7    83
```

We can use vectors of negative integers to indicate the rows and columns to omit in the return result, as shown in this example:

```
> weather[ -1, -3 ]      # -ive, -ive
        Day  Date TempC
2    Sunday Jul 5    30
3    Monday Jul 6    28
4   Tuesday Jul 7    28
5 Wednesday Jul 8    31
```

In the preceding examples, we have used the same input type for both rows and columns. However, we can mix up the input types, as illustrated in this example, where we select the first four rows and all the columns:

```
> weather[ 1:4, ]        # +ive, Blank
       Day  Date TempF TempC
1 Saturday Jul 4    75    24
2   Sunday Jul 5    86    30
3   Monday Jul 6    83    28
4  Tuesday Jul 7    83    28
```

Logical Subscripts

We often use logical subscripts to reference specific rows of the data to return. To perform this action, we need to provide a logical value for each row of the data:

```
> weather                         # The original data
        Day  Date TempF TempC
1  Saturday Jul 4    75    24
2    Sunday Jul 5    86    30
3    Monday Jul 6    83    28
4   Tuesday Jul 7    83    28
5 Wednesday Jul 8    87    31

> weather[ c(F, T, F, F, T), ]  # Logical, Blank
        Day  Date TempF TempC
2    Sunday Jul 5    86    30
5 Wednesday Jul 8    87    31
```

As before, we more commonly apply a logical statement to a column (vector) contained in the data frame to generate the logical vector:

```
> weather[ weather$TempF > 85, ]        # Logical, Blank
        Day  Date TempF TempC
2    Sunday Jul 5    86    30
5 Wednesday Jul 8    87    31

> weather[ weather$Day != "Sunday", ]  # Logical, Blank
        Day  Date TempF TempC
1  Saturday Jul 4    75    24
3    Monday Jul 6    83    28
4   Tuesday Jul 7    83    28
5 Wednesday Jul 8    87    31
```

Character Subscripts

We often use vectors of character strings to specify the columns we wish to return. Although a data frame has “row names,” we tend not to reference rows using character strings. This example selects the `Day` and `TempC` columns from the data, filtering so that only rows with temperatures greater than 85°F are returned:

```
> weather[ weather$TempF > 85, c("Day", "TempC")]  # Logical, Character
        Day TempC
2    Sunday    30
5 Wednesday    31
```

Summary of Subscripting Data Frames

At this point, it is worth a quick review of some of the key syntax used to select subsets of a data frame. In particular, consider the following lines of code:

```
> weather$Day [ weather$TempF > 85 ]                  # Days where TempF > 85
[1] Sunday    Wednesday
Levels: Monday Saturday Sunday Tuesday Wednesday

> weather [ weather$TempF > 85 , ]                    # All data where TempF > 85
        Day  Date TempF TempC
2    Sunday Jul 5    86    30
5 Wednesday Jul 8    87    31

> weather [ weather$TempF > 85 , c("Day", "TempF") ]  # 2 columns where TempF > 85
        Day TempF
2    Sunday    86
5 Wednesday    87
```

In the first example, we are subscripting `weather$Day`. This is a vector, so we provide a single input (a logical vector in this case). It returns the two values of the `Day` column where the corresponding `TempF` column is greater than 85.

In the second example, we are now referencing data from the whole weather dataset. As such, we need two subscripts (one for rows, one for columns). In this example, we use a logical vector for the rows and blank for the columns, returning all columns but only rows where `TempF` is greater than 85. Attention should be paid to the use of the comma in the first example versus the second example, driven by the fact that we are referencing data from a vector (first example) versus the whole data frame (second example).

The third example extends the second example to pick only columns `Day` and `TempF` using a character vector for the column input.

Exploring Your Data

Later in this book, you'll see a range of functionality for manipulating data frames. For now, it is useful for you to look at a few simple functions that will help you to quickly understand the data stored in a data frame.

The Top and Bottom of Your Data

A function called `head` allows you to return the first few rows of the data. This is particularly useful when you have a large data frame and only want to get a high-level understanding of the structure of the data frame. The `head` function accepts any data frame and will return (by default) only the first six rows. For this example, we use the built-in `iris` data frame (for more information, open the help file for the `iris` data frame using the `?iris` command):

```
> nrow(iris)         # Number of rows in iris
[1] 150
> head(iris)         # Return only the first 6 rows
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

This immediately gives us a view on the structure of the data. We can see that the `iris` data frame has five columns: `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`. All columns seem to be numeric, except the `Species` column, which appears to be character (or a "factor," as briefly discussed earlier).

The second argument to the `head` function is the number of rows to return. Therefore, we could look at more or fewer rows if we wish:

```
> head(iris, 2)   # Return only the first 3 rows
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
```

If instead we wanted to look at the last few rows, we could use the `tail` function. This works in the same way as the `head` function, with the data frame as the first input and (optionally) the number of rows to return as the second input:

```
> tail(iris)      # Return only the last 6 rows
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
> tail(iris, 2)   # Return only the last 2 rows
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

Viewing Your Data

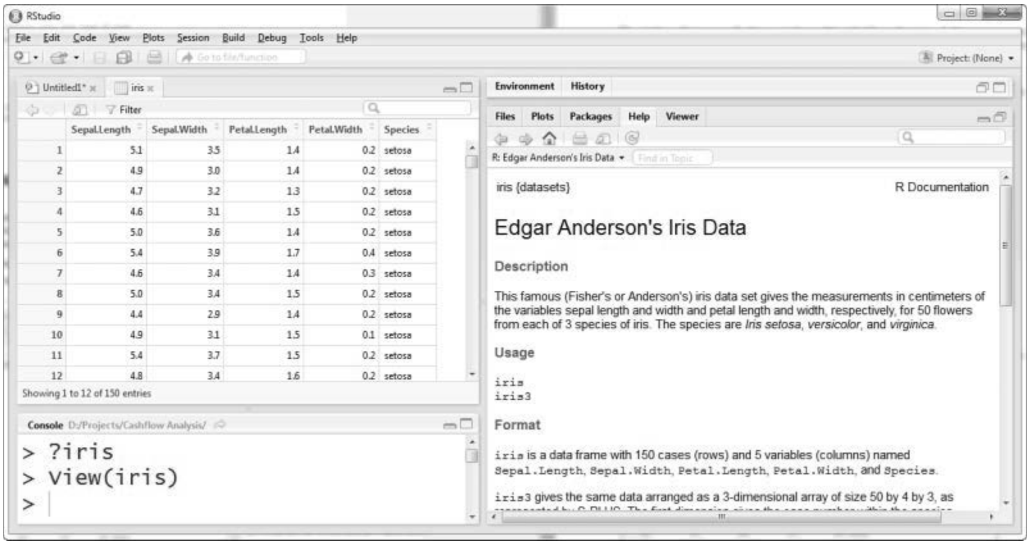
If you are using the RStudio interface, you can use the `View` function to open the data in a viewing grid. This feature in RStudio is evolving quickly, so readers of this book may find the functionality richer than that presented here (the version of RStudio being used is 0.99.441). See Figure 4.1 for an example.

If we use the `View` function, our data frame is opened in the data grid viewer in RStudio:

```
> View(iris)     # Open the iris data in the data grid viewer
```

This window allows us to scroll around our data, and tells us the range of data we are viewing (for example, in Figure 4.1 the message at the bottom of the viewer tells us that we are looking at rows "1 to 19 of 150").

The search bar (top right of the window) allows us to input search criteria that will be used to search the entire dataset. This is used to interactively filter the data based on a partial matching of the search term. As a quick example, look at the result of typing **4.5** in the search bar, as shown in Figure 4.2.


RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Project: (None)
Untitled1*
iris
Filter
Environment History
Files Plots Packages Help Viewer
R: Edgar Anderson's Iris Data
Find in Topic
Console D:/Projects/Cashflow Analysis/

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa

Showing 1 to 12 of 150 entries

```
> ?iris
> View(iris)
>
```

iris {datasets} R Documentation

Edgar Anderson's Iris Data

Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.

Usage

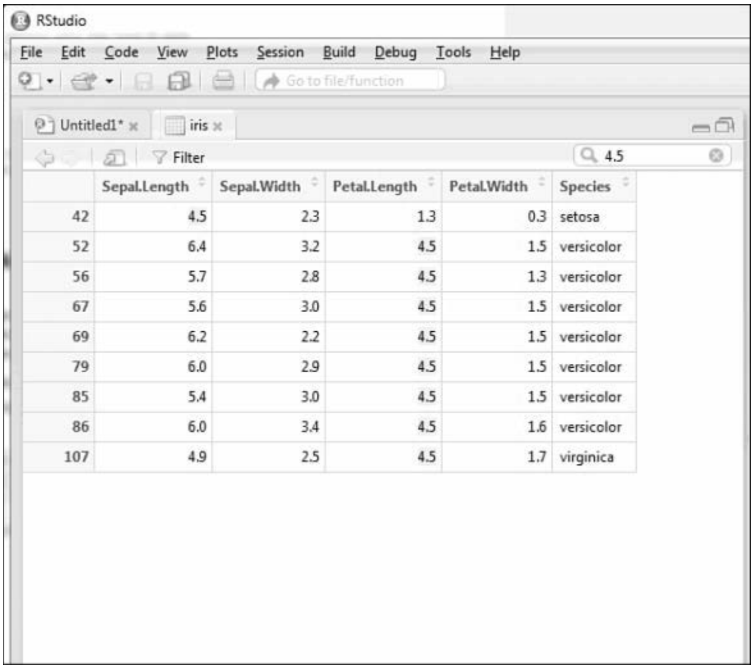
```
iris
iris3
```

Format

`iris` is a data frame with 150 cases (rows) and 5 variables (columns) named `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`.

`iris3` gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as

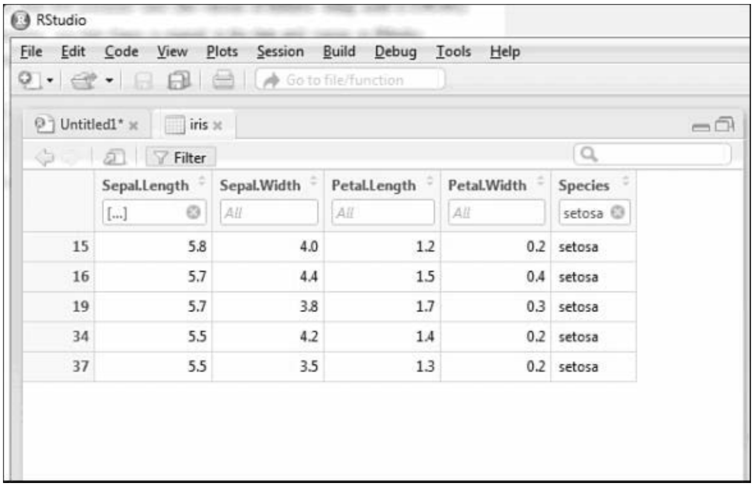
FIGURE 4.1
The iris dataset viewed in the RStudio data grid viewer



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
42	4.5	2.3	1.3	0.3	setosa
52	6.4	3.2	4.5	1.5	versicolor
56	5.7	2.8	4.5	1.3	versicolor
67	5.6	3.0	4.5	1.5	versicolor
69	6.2	2.2	4.5	1.5	versicolor
79	6.0	2.9	4.5	1.5	versicolor
85	5.4	3.0	4.5	1.5	versicolor
86	6.0	3.4	4.5	1.6	versicolor
107	4.9	2.5	4.5	1.7	virginica

FIGURE 4.2
Using the search bar in the data grid viewer

If we click the Filter icon from the top of the data grid viewer window, we will see a number of filtering fields appear, which we can use to interactively subset the data in a more data-driven manner. This example uses the filter feature to look only at rows for the "setosa" species with `Sepal.Length` greater than 5.5 (see Figure 4.3).



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
	[...]	All	All	All	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
19	5.7	3.8	1.7	0.3	setosa
34	5.5	4.2	1.4	0.2	setosa
37	5.5	3.5	1.3	0.2	setosa

FIGURE 4.3
Filtering data in the data grid viewer

Summarizing Your Data

We can use the `summary` function to produce a range of statistical summary outputs to summarize our data. The `summary` function accepts a data frame and produces a textual summary of each column of the data:

```
> summary(iris)   # Produce a textual summary
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

Note that the summaries produced are suitable for each column type (statistical summary for numeric columns, frequency count for factor columns).

Visualizing Your Data

In this book, you will see a number of functions for creating sophisticated graphical outputs. However, let's look at one simple function that creates an immediate visualization of the structure of our data.

We can create a scatter-plot matrix plot of our data frame using the `pairs` function as follows:

```
> pairs(iris)    # Scatter-plot matrix of iris
```

In the graphic shown in Figure 4.4, each variable in the data is plotted against each other. For example, the plot in the top-right corner is a plot of `Sepal.Length` (y axis) against `Species` (x axis).

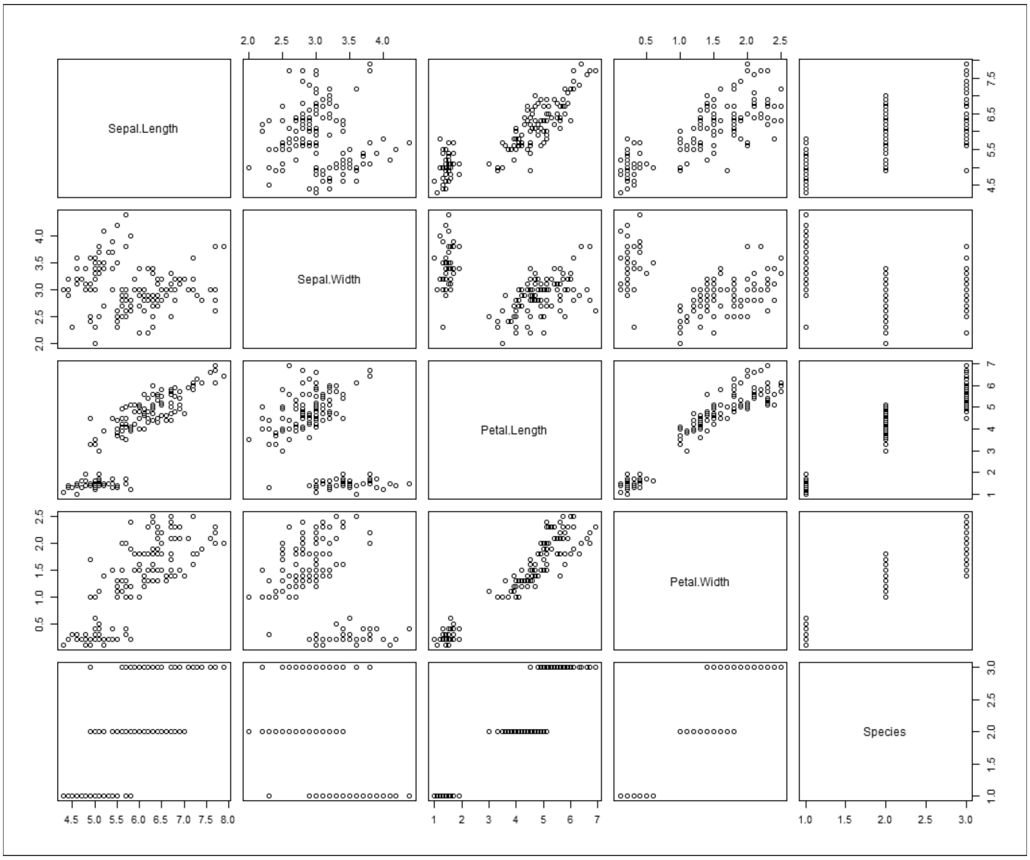


FIGURE 4.4
Scatter-plot Matrix of the iris data frame

From this plot we can quickly identify a number of characteristics of our data:

- We see that the data has five columns, whose names are printed on the diagonal of the plot.
- We can again see that `Species` is a factor column, whereas the rest are numeric.
- If we look at the plots on the right side of the chart, we can see each numeric variable plotted against `Species` and note that the numeric data would seem to vary across each level of `Species`.
- Columns `Petal.Length` and `Petal.Width` would seem to be highly correlated.

Summary

In this hour, we focused on two structures that store "multi-mode" data (that is, data containing more than one data type). First, we looked at lists, which allow us to store any number of objects of varying modes. Then, we looked at data frames as a special "type" of list that stores rectangular datasets in an effective manner.

Although lists are very powerful structures, when we import data into R (which you'll see in Hour 10, "Importing and Exporting"), it will be stored as a data frame. Therefore, you need to be very comfortable manipulating this structure in particular. You should practice the syntax relating the subscripting of data frames using square brackets and the `$` symbol, because this is a fundamental skill useful across all R tasks.

Q&A

Q. Can we create nested lists?

A. Yes. Because lists can store any type of object, they can themselves store other lists. Here's an example:

```
> nestedList <- list(A = 1, B = list(C = 3, D = 4))  # Create a nested list
> nestedList                                          # Print the nested list
$A
[1] 1

$B
$B$C
[1] 3

$B$D
[1] 4
```

```
> nestedList$B$C                 # Extract the C element within the B element
[1] 3
```

Q. What other inputs can we use within the double square brackets?

A. In the last hour, you saw that you can use integers to directly reference elements of a list. Refer to the help file (opened using `?"[["`) for a complete list of possible inputs. However, it is worth nothing that you can use single-character strings to reference columns. Here's an example:

```
> weather           # The full dataset
        Day  Date TempF TempC
1  Saturday Jul 4    75    24
2    Sunday Jul 5    86    30
3    Monday Jul 6    83    28
4   Tuesday Jul 7    83    28
5 Wednesday Jul 8    87    31
> col <- "TempC"    # The column we want to select
> weather[[col]]    # Return the TempC column
[1] 24 30 28 28 31
```

Q. What is the difference between `DF[ ]` and `DF[ , ]`?

A. As shown previously, you subscript data from a data frame using square brackets. Here's an example:

```
> weather [ , c("Day", "TempC") ]   # All rows, 2 columns
        Day TempC
1  Saturday    24
2    Sunday    30
3    Monday    28
4   Tuesday    28
5 Wednesday    31
```

In this example, we provide two subscripts for the data frame: blank for the rows (so all rows are returned) and a character vector to select two columns. The subscripts are separated by a comma. If we omit the comma, we appear to get the same result:

```
> weather [ c("Day", "TempC") ]     # 2 vector elements
        Day TempC
1  Saturday    24
2    Sunday    30
3    Monday    28
4   Tuesday    28
5 Wednesday    31
```

Here, we are using the fact that a data frame is actually a named list of vectors. In this case, we are creating a "sub-list" containing only the two columns specified.

Q. Why, when I select a single column, is it returned as a vector?

A. When you select a single column via the square brackets approach, it is indeed returned as a vector:

```
> weather [ , c("Day", "TempC") ]    # 2 columns - returns a data frame
        Day TempC
1  Saturday    24
2    Sunday    30
3    Monday    28
4   Tuesday    28
5 Wednesday    31
> weather [ , "TempC" ]              # 1 column - returns a vector
[1] 24 30 28 28 31
```

In this case, the last line is equivalent to `weather$TempC`. When you select a single column of data, R simplifies the output in a way that's similar to how you saw matrix dimensions dropped in Hour 3. If you specifically want to retain the dimensional structure, you can use the argument `drop` within the square brackets, as follows:

```
> weather [ , "TempC", drop = F ]    # 1 column - retain dimensions
  TempC
1    24
2    30
3    28
4    28
5    31
```

As you can see from the output, the use of `drop = F` retains the structure, returning a 5×1 data frame.

Workshop

The workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the "Answers" section that follows.

Quiz

1. What is a "list" object?
2. How do we reference elements from a list?
3. What is the "mode" of a list?
4. What's the difference between a list and a data frame?
5. Name two ways we can return the number of columns of a data frame.

6. If we run the following code, what would the contents and structure of `result1` and `result2` contain?

```
> myDf <- data.frame(X = -2:2, Y = 1:5)
> result1 <- myDf$Y [ myDf$X > 0 ]
> result2 <- myDf [ myDf$X > 0, ]
```

7. What is the difference between the `head` and `tail` functions?

Answers

1. A "list" is a simple R object that can contain any number of objects of any "class."
2. We can reference elements of a list using the "double square brackets" notation. Most commonly, we provide the index of the element we want to return from the list (for example, `myList[[2]]` for the second element). If a list has element names, we can alternatively use the dollar notation, specifying the name of the list element (for example, `myList$X` to return the `X` element of `myList`).
3. Because a list is a "multi-mode" object, it has no explicit "mode." If you ask for a list's mode, it simply returns "list."
4. A list can contain any number of objects of any class—its elements may be named or unnamed. A data frame is a "named" list that is restricted to contain only same-length vectors—when printing a data frame, it uses a specific method so the data is presented in a more formatted manner.
5. We can use the `length` function to return the number of columns in a data frame, because this returns the number of vector elements in the underlying "list" structure. Alternatively, because we can treat a data frame as a matrix, we can use the `ncol` function to achieve the same result.
6. The `result1` object will contain a vector of those values from the Y column where the corresponding X column is greater than 0—specifically, this will be a vector containing values 4 and 5. The `result2` object will contain a data frame with two rows, corresponding to the rows where X is greater than 0 (so rows 4 and 5 of the original data frame).
7. The `head` function returns the first six rows (by default) of a data frame. The `tail` function returns the last six rows (by default) of a data frame.

Activities

1. Create a "named" list containing a numeric vector with 10 values (called X) and a character vector with 10 values (called Y) and a sequence of values from 1 to 10 (called Z). Use this list:
   - Print the number of elements and the element names.
   - Select the X element.
   - Select the Y element.

- Select values of the X element that are greater than the median of X.
- Select values of the Y element where the corresponding X element is greater than the median of X.

2. Adapt your code to instead create a data frame containing two columns (X = a numeric vector with 10 elements, Y = a character column containing 10 elements, Z = integers 1 to 10). Use this structure:
    - Print the number of columns and the column names.
    - Select the X column.
    - Select the Y column.
    - Select values of the X column that are greater than the median of X.
    - Select values of the Y column where the corresponding X value is greater than the median of X.
3. Further subset the data in the data frame created in the last exercise as follows:
    - Select all rows of the data where Z is greater than 5.
    - Select all rows of the data where Z is greater than 3 and X is greater than the median of X.
    - Select just the X and Z columns from the data where Z is greater than 5.
4. Print the built-in `mtcars` data frame. Look at the help file for `mtcars` to understand the origin of the data. Use this data frame:
    - Print only the first five rows.
    - Print the last five rows.
    - How many rows and columns does the data have?
    - Look at the data in the RStudio data viewer (if you are using RStudio).
    - Print the `mpg` column of the data.
    - Print the `mpg` column of the data where the corresponding `cyl` column is 6.
    - Print all rows of the data where `cyl` is 6.
    - Print all rows of the data where `mpg` is greater than 25, but only for the `mpg` and `cyl` columns.
    - Create a scatter-plot matrix of your data.
    - Create a scatter-plot matrix of your data, but only using the first six columns of the data.

Index