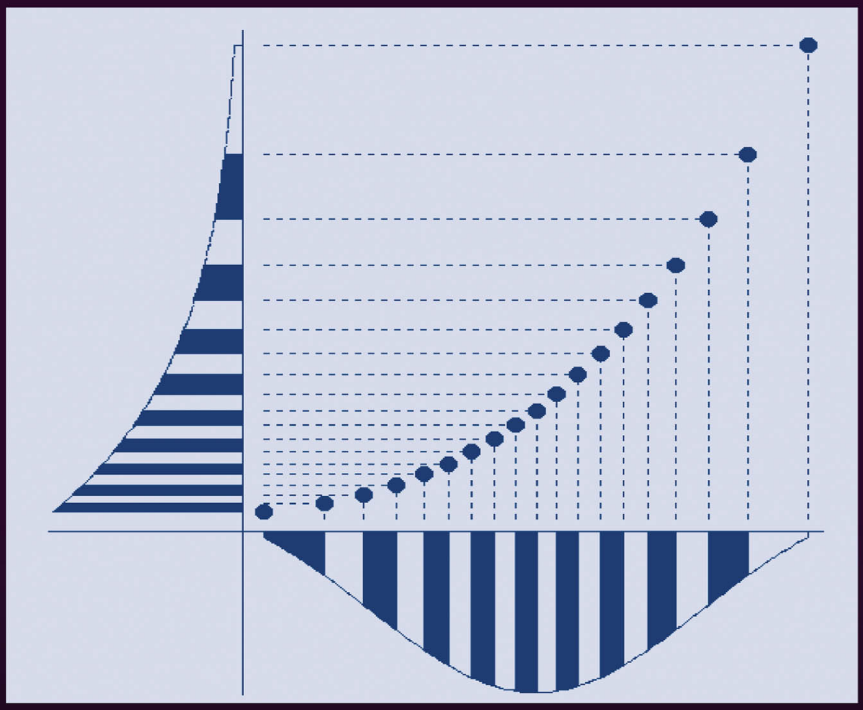
The R Series

Using R for Introductory Statistics

Second Edition



John Verzani

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Using R for Introductory Statistics

Second Edition

Chapman & Hall/CRC
The R Series

Series Editors

John M. Chambers
Department of Statistics
Stanford University
Stanford, California, USA

Torsten Hothorn
Division of Biostatistics
University of Zurich
Switzerland

Duncan Temple Lang
Department of Statistics
University of California, Davis
Davis, California, USA

Hadley Wickham
RStudio
Boston, Massachusetts, USA

Aims and Scope

This book series reflects the recent rapid growth in the development and application of R, the programming language and software environment for statistical computing and graphics. R is now widely used in academic research, education, and industry. It is constantly growing, with new versions of the core software released regularly and more than 5,000 packages available. It is difficult for the documentation to keep pace with the expansion of the software, and this vital book series provides a forum for the publication of books covering many aspects of the development and application of R.

The scope of the series is wide, covering three main threads:

- Applications of R to specific disciplines such as biology, epidemiology, genetics, engineering, finance, and the social sciences.
- Using R for the study of topics of statistical methodology, such as linear and mixed modeling, time series, Bayesian methods, and missing data.
- The development of R, including programming, building packages, and graphics.

The books will appeal to programmers and developers of R software, as well as applied statisticians and data analysts in many fields. The books will feature detailed worked examples and R code fully integrated into the text, ensuring their usefulness to researchers, practitioners and students.

Published Titles

Using R for Numerical Analysis in Science and Engineering, *Victor A. Bloomfield*

Event History Analysis with R, *Göran Broström*

Computational Actuarial Science with R, *Arthur Charpentier*

Statistical Computing in C++ and R, *Randall L. Eubank and Ana Kupresanin*

Reproducible Research with R and RStudio, *Christopher Gandrud*

Introduction to Scientific Programming and Simulation Using R, Second Edition, *Owen Jones, Robert Maillardet, and Andrew Robinson*

Displaying Time Series, Spatial, and Space-Time Data with R, *Oscar Perpiñán Lamigueiro*

Programming Graphical User Interfaces with R, *Michael F. Lawrence and John Verzani*

Analyzing Baseball Data with R, *Max Marchi and Jim Albert*

Growth Curve Analysis and Visualization Using R, *Daniel Mirman*

R Graphics, Second Edition, *Paul Murrell*

Multiple Factor Analysis by Example Using R, *Jérôme Pagès*

Customer and Business Analytics: Applied Data Mining for Business Decision Making Using R, *Daniel S. Putler and Robert E. Krider*

Implementing Reproducible Research, *Victoria Stodden, Friedrich Leisch, and Roger D. Peng*

Using R for Introductory Statistics, Second Edition, *John Verzani*

Dynamic Documents with R and knitr, *Yihui Xie*

Using R for Introductory Statistics

Second Edition

John Verzani

CUNY/College of Staten Island

New York, USA

CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the

Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20150106

International Standard Book Number-13: 978-1-4665-9073-1 (Hardback)

Library of Congress Cataloging-in-Publication Data

Verzani, John.
Using R for introductory statistics / John Verzani. -- Second edition.
pages cm. -- (Chapman & Hall/CRC the R series)
"A CRC title."
Includes bibliographical references and index.
ISBN 978-1-4665-9073-1 (hardcover : alk. paper) 1. Statistics--Data processing. 2. R (Computer program language) I. Title.

QA276.4.V47 2014
519.50285'5133--dc23 2014018934

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

About this book

This is a second edition of a book that introduces R alongside the introductory statistics curriculum. The first edition found its niche with individuals looking to get started with both areas outside of a classroom environment. It is the hope, that this second edition will be even more useful for that task.

The book was first published in 2004, when R was at version 2.0.0. Now, as of writing, R is past version 3.0.0 (3.1.0 and climbing). In that time so much has changed. For example:

- The number of R users has grown enormously. A recent survey ranked R the 15th most used programming language.
- The number of add-on packages for R has grown four- or five-fold to over 5,000. The depth and range of applications has grown considerably.
- The number of books including material on R has grown at least ten-fold.[1]
- The internet has developed many additional R communities beyond the initial mailing list. Two key additions are the question and answer site `stackoverflow.com` which has nearly 50,000 questions tagged with "`r`" and the blog aggregator `r-bloggers.com` which has over 13,000 blog entries related to R.

Basically, the amount of material out there related to learning and using R is now enormous. This book doesn't try to canvas even a sliver, rather it tries to guide the reader through the learning of the basics of R so that it is possible to take advantage of the contributions made by the R community. Though R—like other programming languages—has a reputation of having

[1] For example, there are many other texts introducing R, as this one does, that can be chosen to learn from. For example, [15], [64], [13], [14], [36], [12], [56], and `http://www.openintro.org/stat/`.

a steep learning curve, we try to break this down into small, task-oriented steps.

In this edition we place a greater emphasis on more idiomatic R. For a small example, despite the greater familiarity of using = for the assignment operator, we now use the <- operator. Another example comes in Chapter 4, where we resist the temptation to illustrate some data manipulations with the widely used `plyr` package and instead utilize similar functions from base R. For our limited demands, the corner cases that led to the desire for a `plyr`-type approach are not present, and we have the belief that it is good to start with a grounding in the functionality provided by base R.

We also try to avoid as many of the pitfalls as possible for new R users by encouraging the use of RSTUDIO, a feature-rich, cross-platform development environment for interacting with R. RStudio has very good integration with R's help system and its administrative tools; it has an integrated debugger, a powerful editor, and much more. Though relatively new to the R community, the company has already made an enormous contribution.

This book was written using the excellent `knitr` package for R. This package allows one to embed R code into a document with ease. The formatting of code blocks follows a convention championed by the `knitr` author. We think it makes the code much easier to read, and hence, reason about. It also encourages thinking of interacting with R using a script, rather than the command line directly. This style of usage is facilitated by RStudio.

In addition to changes with R, the teaching of introductory statistics (by which we mean a non-calculus approach to inferential statistics) has changed in the last decade, or so. For example, primarily due to the widespread availability of computational resources but also for pedagogical reasons, there have been pushes to include resampling approaches, permutation methods, and Bayesian analysis into the first-year course. The topics of this text hew closely to the traditional ones, be we have added a bit on these computer-intensive approaches, in particular to motivate the more traditional approach. We continue with an emphasis on realistic data and examples (which required updating some now not-so-topical examples) and we rely on visualization techniques to gather insight. Fortunately, the R language makes such inclusion quite easy.

Organization The text has three main parts. The first five chapters introduce the basics of exploratory data analysis and data manipulation in R. The approach is a little slower than it need be. We postpone until Chapter 4 the details of using R's data frames. These are the primary means to store multivariate data in R, and in Chapters 4 and 5 we demonstrate many tools that can act with data frames to make data investigation very convenient. However, most of these techniques are a bit more abstract, so in the first chapters we emphasize a more direct, easier to learn approach, albeit sometimes more tedious. Most all of this material was rewritten for the second edition.

Chapters 6 through 10 cover the core of statistical inference. We added the material in Chapter 7 to introduce the major themes of inference using computation, rather than probability calculations, to give insight into questions on inference.

Chapters 11 through 13 introduce the topic of analyzing statistical models with R, covering the regression model and its specialization to analysis of variance, before ending with a brief introduction to the logistic model and non-linear models. The goal is to cover the main introduction to this topic, and to show that the basic interface R provides extends naturally to cover a wide variety of models.

The appendix on programming discusses some of the details of writing programs in the R language. In the main part of the text, user-written functions are fairly straightforward, so this material is just supplemental.

The `UsingR` package The book has an accompanying package, `UsingR`. This package is available from CRAN, R's repository of user-contributed packages. Installation should be painless. The package contains the data sets mentioned in the text (`data(package="UsingR")`), answers to selected problems (`answers()`), a few demonstrations (`demo()`), the errata (`errata()`), and sample code from the text.

Thanks The author would like to thank Chapman & Hall/CRC Press. Not just the editors who have pushed for this new edition, but the company as a whole for its work on numerous titles on R-related topics. In a similar manner, the author would like to thank `statistics.com`. They offer a variety of R-related courses, including one that features this text. The feedback from the students of that course has been important guidance in the redrafting of parts of this text. Finally and most importantly, the author would like to warmly acknowledge the continued support he has received from his family on this and other projects.

John Verzani
February, 2014

1

Getting started

1.1 What is data?

Data and their statistical summaries and interpretations are ubiquitous. For example, we found these four articles during a typical day reading the paper:

• **Example 1.1: To compile evidence to establish cause and effect**
In an opinion piece, Joe Nocera [46] discusses the prevalence of guns in the movies (in anticipation of yet another "Die Hard" movie). He quotes a spokesperson from the Motion Picture Association of America as

> "There is a predominance of findings that show there is no consistent or convincing evidence that exposure [to gun violence in movies] causes people to be more violent."

However, Nocera immediately refutes this quoting a professor from the University of Wisconsin: "There is tons of research on this."

Clearly the collection and interpretation of data is crucial when making policy decisions. This isn't an easy task, of course. A casual reader may think the above differences of opinion are a matter of political motivation, but this need not be the case. Relationships between variables can exist, even if there is not a cause and effect relationship. Trying to find convincing evidence in data often requires a careful collection of data in order for conclusions to be made. ••

• **Example 1.2: Price of a hip replacement**
In a news piece, Elisabeth Rosenthal [51] describes the research of Jaime Rosenthal who called more than 100 hospitals, covering every state in the summer of 2012 seeking the price of a hip replacement for a hypothetical, uninsured, 62-year-old female. The results were surprising:

1. Only about half the institutions could provide an estimate
2. Of those that could, the range of prices went from $11,000 to $125,798

Commentary in the article urges people to place the price data in the context of many other factors such as infection rates and unexpected deaths. However, the article summarizes the primary researcher's belief that there is little consistent correlation between higher prices and better quality in American health care.

Even in what is perhaps the most data-driven industry, there is clear need for data and context to place this data within. Further, this example hints at some other difficulties in data collection: e.g., the question of what to do with missing data, as it is often the case that some values will be unavailable. As well, the issue that the actual mechanism for computing this value at a given hospital may vary from that of another. ••

• **Example 1.3: Safety of the airline industry**
In a front page article titled "Airline Industry at Its Safest Since the Dawn of the Jet Age," authors Jad Mouawad and Christopher Drew [43] summarize the data collected by the Aviation Safety Network pointing out that 2012 had only 23 deadly accidents and 475 fatalities. This may sound high, but putting it into a rate helps give context: this is a risk of one death per 45 million flights. That is, a person could fly daily for an average of 123,000 years before being in a fatal plane crash.

The improvements in safety are not limited to advanced technologies, as the industry (regulators, pilots, and airlines) have created a culture of sharing data about flying hazards with the goal of preventing accidents.

This example shows how a focus on understanding the many factors that can contribute to a given statistic can help improve an area. It wasn't enough that the airline kept statistics, but rather that they used their findings to address shortcomings. ••

• **Example 1.4: Networking**
On the business page Andrew Sorkin [53] reports on a data base containing names of over two-million deal makers, power brokers and business executives, *and* in many cases the name of spouses, children, associates, political donations, charity work, and more. This information held by a company called Relations Science is compiled by more than 800 people.

The goal of course is to sell this information to people who plan to leverage the network of relationships. Of course, other companies, such as Facebook and LinkedIn have such information on their users, and the NSA seeming has all the data it could ever need, but in this case the information is scraped from web sites—a person need not be a member of a social network or have a security clearance.

How such large data bases get mined and what this means for personal privacy will likely continue to be a major topic of conversation for years to

come. Though the statistical techniques of working with so-called "big data" are outside the scope of this text, many of the computational skills will be developed. ••

In this sampling of articles, we see the analysis of data used in many different ways:

- Under the name "studies," data is used to make a case about social policy (in two different ways!).
- To investigate variability in prices and transparency, data is collected and summarized.
- In an industry, data demonstrates that forward looking practices can have a substantial effect.
- Data and the information it contains is mined to establish a financial advantage.

Data and its analysis is a very wide topic, so wide we couldn't begin to describe it all. In this text we narrow our focus, looking at data with an eye towards *statistical inference*. This is the process of drawing conclusions about populations based on data collected from these populations. To do this, we will use the language of probability. This will give us the flexibility to describe concrete things using data subject to random variation. Exactly how this will be used will require us to make models for our data. This text is roughly organized into three areas: the first to develop techniques for exploring data, the second the basics of statistical inference, and the third area covers the beginnings of modeling with data.

The rest of this chapter is focused on getting started with using R. We save more statistically oriented examples for Chapters 2 and beyond.

1.2 Getting started with R

This section covers the basics of getting started with R, beginning with some notes on installation and continuing with the basics of interacting with R through the command line.

Installing R

Before beginning with R, it must be installed for usage. R is available as source code from CRAN, `http://cran.r-project.org/`. However, most users probably will install R from a distributed binary. These are also available from CRAN. For example, the Microsoft Windows binary is distributed as a self-extracting `.exe` file. Simply download the file then install it as any other download. For Microsoft Windows users, the standard installation will

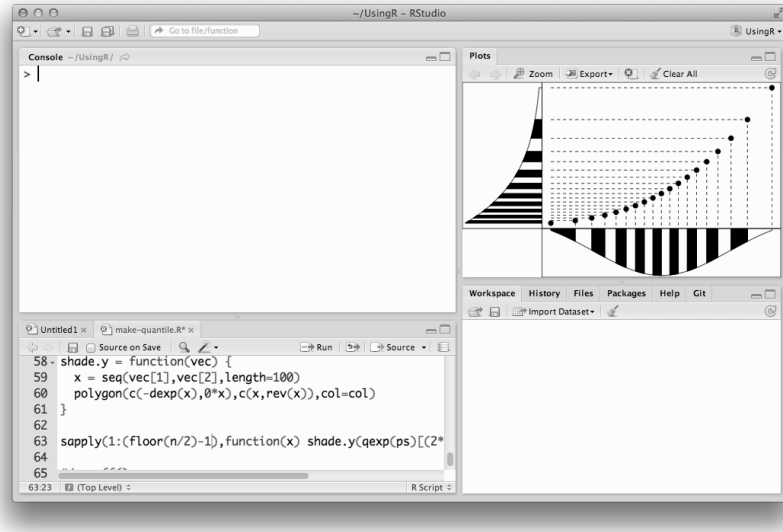


Figure 1.1: The RSTUDIO development environment for R. Visible are the console, the source code editor, the plot pane, and the workspace pane.

create a desktop icon and start menu item for opening R. If started this way, R will open to its standard Microsoft Windows GUI, but we suggest using RSTUDIO®, as described next.

Sometimes installation is a bit more difficult than described. For example, user permissions can be an issue. The "R for Windows FAQ" document, also from CRAN, can be consulted for remedies for the more common issues.

Installing RStudio

In this book we will assume the reader has installed the RSTUDIO IDE. This open-source, integrated desktop environment makes it possible for all R users to have a common R interface, which is greatly enhanced over the R's basic command line interface. Figure 1.1 shows a sample screenshot.

Installation is straightforward in most cases. The RSTUDIO web site `http://www.rstudio.com` has links to the necessary files to download. If there are issues, the support forum (`http://support.rstudio.org/`) is available for assistance. When RSTUDIO is started, it starts R with it. Starting RSTUDIOis done in a manner consistent with other applications for your operating system. For example, the Microsoft Windows installation will add an entry to the "Start Menu" to load the program.

Figure 1.2: RStudio's console showing the issuing of the command "2 + 2" and R's response of 4.

R's command line

There are several ways to interact with R, but for us the primary one will be through the *command line*, also known as the console. The command line in RStudio is in the console pane (Figure 1.2). The command line is common to all of R's interactive interfaces. The name comes from it being the place where one types in *commands*.

In the figure we typed the command "2 + 2" then pressed the return key to send the command to R's interpreter. It responded with the answer of 4, prefixed with a `[1]`, which will make sense when we talk about data vectors in Chapter 2.

In this text, rather than show screenshots of the RStudio console, we typeset the command line. The "2 + 2" command would look like:

```
2 + 2

## [1] 4
```

Whereas, the average of five numbers might look like:

```
(1 + 3 + 2 + 12 + 8)/5

## [1] 5.2
```

The output is prefaced with R's comment character # to distinguish it from the input. Any text after a comment character is ignored by R's parser. Placing comments in front of output is not the convention with most R consoles, including RStudio, but is used here for the typesetting of R code used with this text, as we prefer not to include the prompt and need a visual clue to separate input code from output.[1]

R uses standard conventions for mathematical operations: `+`, `-`, `*`, `/`, and `^`. Here we find the distance between two points $(1,3)$ and $(2,1)$:

[1]This style also is how one would interact with the R process when typing commands into a "script file" and executing these through R's `source` function or RStudio's "run" features. Using a script makes it much easier to reconstruct one's work in a subsequent session.

```
( (2 - 1)^2 + (1 - 3)^2 )^(1/2)

## [1] 2.236
```

R uses parentheses for grouping, as is done in math texts. Parentheses are also used when calling functions, as described shortly. Square brackets are used to extract and assign values to objects that can contain more than one. Examples will start in Chapter 2, where we discuss a container for a set of data.

Combining commands We can place more than one command on the command line at once. We use a semicolon, `;`, to separate them.

The prompt The command line has two states, one being it is ready for input, the other expecting a continuation of the currently inputed line. It marks these states with a *prompt*. By default this will be `>` for a ready state and `+` for a continuation state.[2] These are not typeset in the text, as they can be distracting while reading. But be warned, the `+` prompt is indicating the previous command was not complete. If you thought it was, likely you are missing a closing parentheses.

Errors Of course, there are times where we type in a command that does not make sense to R's interpreter. This can happen, for example, when we misspell a command name or make some syntax error. Here we have two `^` symbols, one too many for R's taste:

```
2 ^^ 2

Error: unexpected '^' in "2 ^^"
```

The error messages generated by R are usually quite informative, though may seem cryptically written to the new R user. The above one is pretty clear. R may also generate warnings, which are similar to an error, but do not stop the flow of a function.[3]

Command history After one issues a command, it is recorded in R's history. Most command lines allow for scrolling through the previous commands using the up- and down-arrow keys. This can be used to edit and re-execute a previous command.

RStudio has a history pane (Figure 1.3) showing the past commands. One can double click on a command to send it back to the command line. Selecting more than one and then pressing the "To Console" toolbar item will

[2] These can be changed through the `prompt` and `continue` options, cf. `?options`.

[3] If "sourcing" in commands from a script in RStudio, the error message will conveniently contain a line number.

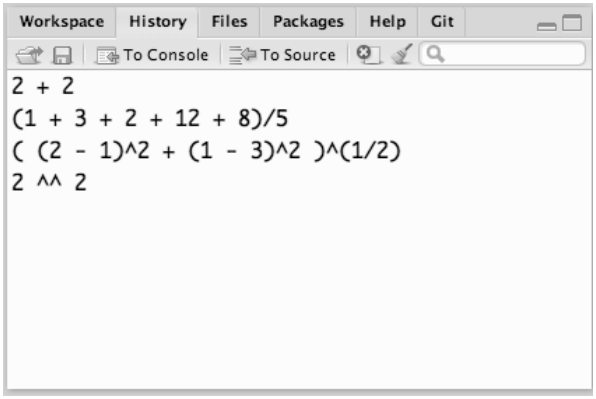


Figure 1.3: RSTUDIO's history pane showing its recording of previously issued commands.

send the collection of commands back to the console. As the history stack can grow quite large, the search panel in the history pane allows one to search through past commands. When the desired one is located, it can be viewed in its context by clicking on the small arrow on the right.

Variables

R can be used like a calculator, as above. But it really is an environment for statistical computing and graphics. The power of R goes well beyond that of a graphing calculator. One immediate difference is the ability to assign names to values.[4] In R this is done with an *assignment operator*. We use the left arrow for assignment. In RSTUDIO, there is a keyboard shortcut to insert the two-character <-, which for Windows users is alt + -.[5]

For example, here we assign a value to x and then refer to x in the subsequent command:

```
x <- 2
y <- x^2 - 2*x + 1
y                                  # assignment does not print output

## [1] 1
```

R is a dynamic language, which means we can redefine and retype values:

[4] Assignment basically gives an object a name in such a manner that R can look it up when asked. This process of lookup follows a procedure that defines R's scoping rules. The *scope* of a variable is the context in which the bound variable can be found. Some knowledge of this becomes important when programming new functions.

[5] Alternately, an equals sign may be used for assignment. This is more traditional with programming languages, but we stick with the R community's preferred convention.

```
x <- "two"                              # x has a new value
```

The value of `y`, assigned when `x=2`, does not reflect the new value assigned to `x` unless you reissue that command.

Variable names can be long or short. Here we define a variable `some_data`:

```
some_data <- 9.8
```

Case is important The case of the letters in a variable name is important. There is a distinction between `x` and `X`, or `mydata` and `myData`. This is the case with everyday language, so shouldn't be surprising, but isn't always true when using computers.

Valid names The R documentation states that a syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. While longer names can be more descriptive, shorter ones are, of course, easier to type, but harder to remember what they represent.[6]

Tab completion R command lines generally have *tab completion*. When a command is partially entered and the tab key is pressed, a list of possible completions for the current token are presented, or, if there is a unique completion, this token is filled in. This can make it much easier to use longer variable names, as one rarely needs to type the entire name. Figure 1.4 shows the options for completion of the token `boxpl`.

Built-in variables R has very few built-in variables. One is `pi` referring to the value π. Another is the variable `T` referring to the logical `TRUE` value. These names may have new values bound to them.

Functions

The R language is comprised of numerous built-in functions, providing a rich set of actions. Several of these functions are for the familiar mathematical operations:

```
x <- pi
sin(x)                                  # floating-point inaccuracy
```

[6] There are many conventions used for making longer variable names more readable. Here are some alternatives to our use of `some_data`: `some.data`, `someData`, `SomeData`. The use of a period to separate words is common, but we reserve that for programming S3 functions. The latter two examples are camel case and upper camel case. Both are widely used. We use an underscore, as it seems easier to read, but there is no consensus in the R community on this topic.

Figure 1.4: Tab completion in RSTUDIO presents the possible choices for completion, if there is more than one. When completion options are shown for a function name, a summary from the help page of each possible function is presented along with one-key access to the full page. Argument completion also shows a description of the argument.

```
## [1] 1.225e-16

sqrt(x)

## [1] 1.772
```

Functions are *called* by their name followed by a pair of parentheses. If there is more than one argument, which is often the case, these are separated by commas. An example of this would be the logarithm function which has an optional argument for the base:

```
log(x)                                  # log base e = exp(1)

## [1] 1.145

log(x, 10)                              # log base 10

## [1] 0.4971
```

One of the more commonly used functions in R has the short name c. This function is used to combine values together. Here we combine several numbers and assign them to the variable x:

```
x <- c(74, 122, 235, 111, 292)
```

A typical use of this is to create a data set, of which we discuss much more in Chapter 2. There is a range of statistical functions defined for such objects. For example, we can take the average (or mean) value:

```
mean(x)

## [1] 166.8
```

The mean function in this example takes several numbers and summarizes them with 1. It does so by adding the numbers and dividing by the number of values added. This can also be achieved with:

```
sum(x)/length(x)

## [1] 166.8
```

There are also many functions for manipulating container objects like `x`. For example, `head` and `tail` which return the first (last) `n` elements, where by default `n=6`.

Vectorized functions R has several functions which do not summarize a collection of values with a single number, but rather do the same thing for each number. Such functions are called *vectorized*. Some examples are the standard mathematical functions:

```
x + x

## [1] 148 244 470 222 584

sqrt(x)

## [1]  8.602 11.045 15.330 10.536 17.088
```

This is very natural with statistical use. Here we subtract a single value from each value of `x`:

```
x - mean(x)

## [1] -92.8 -44.8  68.2 -55.8 125.2
```

In this last example the sizes of `x` and `mean(x)` did not match. R will recycle values from the smaller one to create a new matching-sized object, then do the vectorized subtraction.

Default arguments, named arguments As mentioned, R functions can have one or more arguments. This is a good thing, as it allows the user to customize a call to a function without needing to remember many different function names. To make it much easier to use functions with many arguments, the author can provide reasonable defaults for as many arguments as they see fit. This allows the user to specify relatively few values for common cases, and adjust values as desired for other, less common cases. For example, the `mean` function has an argument to trim the data before finding the average. This is specified with a value between 0 and 0.5, with a default of 0. With this default, we've seen the familiar average is found. When we specify the other extreme value, 0.5, we actually get the median, or middle value:

```
mean(x, trim=0.5)

## [1] 122

median(x)

## [1] 122
```

The above used a *named argument*, in this case `trim=0.5`. Additional arguments can be matched by position or by keyword. In this example either could have been used. We tend to be explicit by using keywords for additional arguments, as it is easier to see what is being specified.

Generic functions Not only can R programmers create different arguments to give functions extra flexibility, R programmers can also create entirely different function definitions based on the type of these arguments. That is, the same name may refer to different function implementations. Functions for which this is implemented are termed *generic functions*. In most cases, the exact choice of definition to dispatch depends on the *class* of the first argument. We will discuss this feature at more length in Chapter 2 and further in Appendix A. For now we illustrate with an example, using R's `summary` function:

```
x <- c(74, 122, 235, 111, 292)          # numeric
y <- c(TRUE, FALSE, TRUE, TRUE)         # logical
summary(x)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      74     111     122     167     235     292

summary(y)

##    Mode   FALSE    TRUE    NA's
## logical       1       3       0
```

As seen, the summary of a collection of numbers is a statistical summary. For a collection of logical values, it is a count. The R user needs only be mindful that the function `summary` presents a reasonable summary of an object, and not worry about what specifically that summary will be.

Though this feature can cause confusion at first, it has a significant advantage in that far fewer function names need be remembered, as similarly behaved functions can be given the same name.[7]

[7] In describing functions which are generic in the text, if not noted, the most typically used implementation is described.

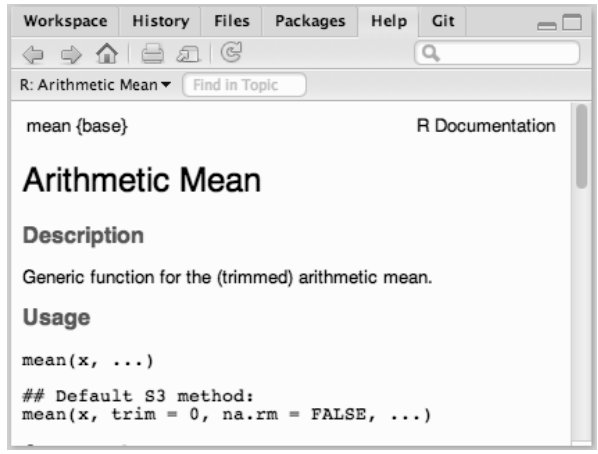


Figure 1.5: The help pane in RStudiodisplaying the help page for the `mean` function from base R. Along the top the selector on the left is used to select previously displayed topics, the middle search box searches through page, and the rightmost search box searches the help system.

Help R is comprised of a fairly small set of base functionality and is extended by adding additional packages to one's workspace. For the most part, the data sets and functions that are available in base R and its add-on packages are documented. R's help system allows one to access these help pages. The most basic access is provided by the `help` function, which has a shortcut `?`, as in `?mean`.

In RStudio, the help pane provides an interface. Figure 1.5 shows the output from issuing the `?mean` command. This command pulled up the help page for the `mean` function from base R. One can see a description and various ways it can be used. The `mean` function is a generic function, and the second usage shows what is available by default, when there is no other special implementation for the given arguments.

In Figure 1.4 we see that tab completion in RStudio for a function provides access to the help page through the f1 function key.

R provides several layers of help. Table 1.1 lists a quick summary of what various commands produce, when issued from any R console:[8]

The workspace

After interacting with R one typically has created several objects and perhaps functions. Without doing anything special, R will maintain these objects in a global *Workspace*.[9] When R searches for an object at the command line, this is the first place on its path that it will look.

[8]There is also the add-on package `SOS` to search over contributed packages.

[9]This is kept in an environment returned by `globalenv`.

Command	Description
`apropos("mean")`	List objects whose names match 'mean'.
`help("mean")`	Find help on the `mean` function. Alias is `?mean`.
`example("mean")`	Run examples found in help page for `mean`.
`help.search("mean")`	Search help data base for terms matching 'mean', searching over names, title, alias, keywords, etc. Alias is `??mean`.
`help(package="MASS")`	List general information on the specific package.
`vignette()`	List all vignettes, supply topic and/or package to narrow.

Table 1.1: Example usage of various commands to access the built-in documentation.

RStudio lists most items in the global workspace along with a short summary in the Workspace pane (Figure 1.6). Clicking an item brings up an editor or viewer, depending on the object.

From the command line, the `ls` function can be used to list the objects in the global workspace (or other environments). When used at the console, it will list the data sets and functions a user has defined.

For example, the following lists the currently defined objects in the global workspace:

```
ls()

## [1] "a"   "d"   "out" "x"   "y"
```

To get a short summary of an object, the `summary` function can be used. The `str` function can give a longer, more cryptic, summary of the structure of an object.

For example, we've seen the summary of the following produces a statistical summary. Here we see what the structure is:

```
x <- c(74, 122, 235, 111, 292)
str(x)

##  num [1:5] 74 122 235 111 292
```

You may wish to remove objects from the workspace. This can be done through the `rm` function. The following shows how to remove a single object, and how to remove all the objects in the workspace.

```
rm(x)                                   # single object
rm(list=ls())                           # all objects
```

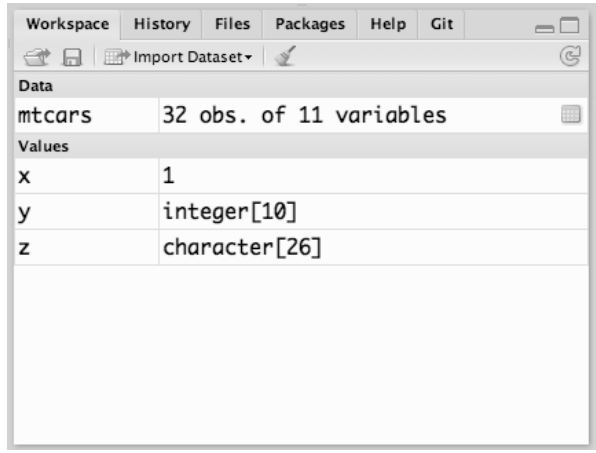


Figure 1.6: The Workspace pane offers a listing of the objects in a user's global workspace by type. Clicking an item opens an appropriate editor or viewer.

The latter, uses `ls` to return the names of the current objects. As these are character data, the `list` argument is employed. In RSTUDIO this last action is initiated by the "broom" toolbar icon on the Workspace pane.

Sessions The global workspace and history file contain your currently defined objects and the steps for how they were created. Both are useful to keep, and R can do so from session to session. When one quits R, a prompt to "Save workspace image" is given. The default choice will write the contents of the workspace to a file to be read back in when R is started again.[10] This means that your objects are persistent from session to session.

Projects RSTUDIO users have more options than just keeping track of a history file and global environment from session to session. The project framework allows an RSTUDIO user to specify a directory and its files and subfolders as part of a project. In addition to providing a means to store the session information, projects make it very easy to search over all accompanying files and allows these files to easily be put under version control. Both of these are quite useful when programming with R, though we don't make use of them in this text.

[10] R provides the functions `save` and `load` to write (and read) representations of R objects to a file. The saved workspace is written to a file `.RData` in the current working directory. When R is restarted in that directory this file is loaded in as part of the usual startup process. The help page `?Startup` documents the startup process.

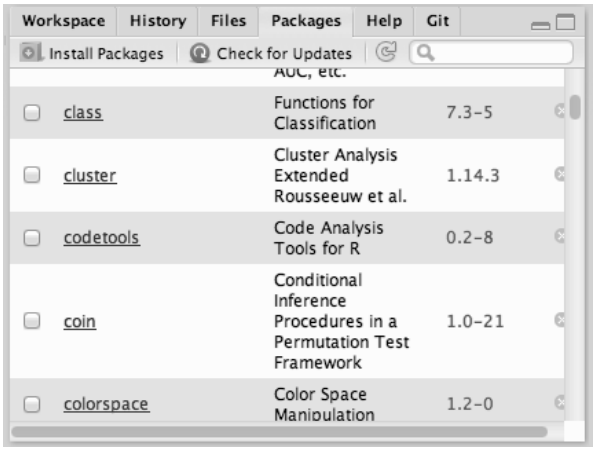


Figure 1.7: RStudio package interface allows one to easily load or unload an installed package, as well one can install packages from CRAN and other sources.

External packages

As mentioned, R can be extended through external *packages* which one can install into a local R environment. There are literally thousands of such packages available.

Packages are primarily available through CRAN, R's worldwide repository of packages and R source. Several packages are also available through the BioConductor project `http://www.bioconductor.org`, r-forge `https://r-forge.r-project.org/`, GitHub `https://github.com/languages/R`, Google Code Page `https://code.google.com/`, and other sites.

RStudio provides a Package pane for interacting with packages (Figure 1.7). From here one can load and unload currently installed packages by toggling the checkboxes on the left of the package name. Once loaded the (exported) functions and data sets of the package are available for use.

Packages can also be installed onto a user's system. The interface for this requires three pieces of information:

- The package name. As there are so many add-on packages, this is provided through an entry box with autocompletion.
- The repository to install the package from. The default is one of CRAN's repositories. It could also be used to indicate a locally downloaded file or another repository.
- The library of packages to install the package into. When loading an installed package, R searches over available package libraries. Often this can be left to the default, but if there are permission issues or other complications, this may need to be set. For details see `?.libPaths`.

Packages may have dependencies on other packages. The default settings are to automatically install any dependent packages.

Like R, packages are versioned. The "Check for Updates" tool button will search for new versions of currently installed packages and gives the user a chance to update those that are out of date. This is all very similar to how a smartphone keeps track of its installed applications and their versions.

For non-RSTUDIO usage, the following functions perform the core functionality: to load an installed package, there are `require` and `library`; to install a package from CRAN there is `install.packages`; to list the packages available through CRAN, there is `available.packages`; and to update any installed packages to the latest version from CRAN, there is `update.packages`;

For example, the `UsingR` package accompanies this book. To install it one could issue the command:

```
install.packages("UsingR")              # done once
```

If one is not already set, a query, as to which CRAN repository to use for downloading files, will be made. The `UsingR` package has several dependencies.[11] The defaults for the above call will download and install those not currently installed into the user's package library at the same time.

Once downloaded, the function `require` (or alternatively, `library`) is used to attach the package to the workspace:

```
require("UsingR")                       # done each session
```

Data sets

Many packages include accompanying data sets. The `UsingR` package has several that we will see utilized in the text. This package also calls in, among others, the `HistData` package that provides data sets from the history of statistics and data visualization. In addition, base R has a `datasets` package that is loaded automatically, unless one requests something different.

For the most part, the data sets in a package are available in the user's search path, though they don't appear in the Workspace pane by default. For example, the `rivers` data set is part of the `datasets` package. Here we show the first 6 values:

```
head(rivers)                            # head displays first 6 only

## [1] 735 320 325 392 524 450
```

[11] The package depends on the `MASS` [57], `ggplot` [61], `lubridate` [27], `Hmisc` [34], `coin` [31] [32], `aplpack` [63], `vcd` [41], `LearnEDA` [4], `quantreg` [38], and `HistData` [24] external packages.

The data function The `rivers` object cannot be edited directly, any edits will produce a copy in the user's workspace. (This copy will then also display in the Workspace pane of RStudio.) A copy will also be made if one brings the data set into the workspace with the `data` function:

```
data(rivers)                              # create local copy of data
```

The `data` function can also be used to search a package for available data sets, e.g., `data(package="UsingR")`.

The `Cavendish` (`HistData`)[12] data set contains data from a series of experiments carried out by Cavendish in 1798 to estimate the gravitational constant, G. We can look at its first 6 values with:

```
require("HistData")                       # only needs to be done once

## Loading required package: HistData

head(Cavendish)

##   density density2 density3
## 1    5.50     5.50       NA
## 2    5.61     5.61       NA
## 3    4.88     5.88       NA
## 4    5.07     5.07       NA
## 5    5.26     5.26       NA
## 6    5.55     5.55       NA
```

Data frames The output above is different from what we have seen so far. This data set is stored as a *data frame*:

```
str(Cavendish)

## 'data.frame': 29 obs. of  3 variables:
##  $ density : num  5.5 5.61 4.88 5.07 5.26 5.55 5.36 5.29 5.58 5.65 ...
##  $ density2: num  5.5 5.61 5.88 5.07 5.26 5.55 5.36 5.29 5.58 5.65 ...
##  $ density3: num  NA NA NA NA NA NA 5.36 5.29 5.58 5.65 ...
```

A data frame is R's way of organizing several related variables into one object. Data frames are rectangular sets of data with each column being a variable and each row representing a case. We discuss much more about data frames in Chapter 4. For now, we just want to indicate how one accesses a variable from a data frame.

[12] We use this typesetting convention to refer to data sets in packages that are not loaded by base R.

The output of `str(Cavendish)` shows there are three variables in this data frame: `density`, `density2`, and `density3`. We can reference the values in, say, `density2` through the syntax `dataframe_name$variable_name`, as in:

```
head(Cavendish$density2)

## [1] 5.50 5.61 5.88 5.07 5.26 5.55
```

Later, we will see other ways to do this task and why we use a dollar sign here, but this is perhaps the most common. For now, we see that we can treat this data just like a data set we may have typed in:

```
summary(Cavendish$density2)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    5.07    5.34    5.47    5.48    5.62    5.88
```

Problems

1.1 Use R as you would a calculator to find numeric answers to the following:

1. $1 + 2(3 + 4)$
2. $4^3 + 3^{2+1}$
3. $\sqrt{(4+3)(2+1)}$
4. $\left(\dfrac{1+2}{3+4}\right)^2$

1.2 Rewrite these R expressions as math expressions, using parentheses to show the order in which R performs the computations:

1. `2 + 3 - 4`
2. `2 + 3 * 4`
3. `2/3/4`
4. `2^3^4`

1.3 Use R to compute the following

$$\frac{1 + 2 \cdot 3^4}{5/6 - 7}.$$

1.4 Use R to compute the following

$$\frac{0.25 - 0.2}{\sqrt{0.2 \cdot (1 - 0.2)/100}}.$$

1.5 Assign the numbers 2 through 5 to different variables, then use the variables to multiply all the values.

1.6 The `rivers` data set is loaded when R is. View the data by typing its name and then the return key. What is the last value listed?

1.7 The `exec.pay` (`UsingR`) data set is available from the command line after loading the package `UsingR`. Load the package, and inspect the data set. Scan the values to find the largest one.

1.8 For the `exec.pay` (`UsingR`) data set, apply the functions `mean`, `min`, and `max`. What are the values found?

1.9 The basic `mean` function has an additional argument `trim`. When given, the specified proportion of the data is trimmed from the sorted data before the mean is taken. Compare the difference between `mean(exec.pay)` and `mean(exec.pay, trim=0.10)`.

1.10 The `Orange` data set is stored as a data frame with three variables. What are the three variables?

1.11 Compute the average age of the trees in the `Orange` data set using `mean`.

1.12 Compute the largest circumference of the trees in the `Orange` data set.

2

Univariate data

We discuss in this chapter single variable (*univariate*) data sets and various summaries for such data. Univariate data are the building blocks for multivariate data sets, but we resist the temptation to start there, preferring to take our time in the development.

First, what do we mean by a data set? Let's think about it in terms of a data collection process. We may wish to understand measurement or characteristics of several different cases.

A *case* is one of several different possible items of interest. A typical example would be the individuals in some population (a classroom, likely voters). In some texts [42] this is how cases are defined, but we prefer a more generic term to avoid confusion with examples such as hospitals in a state or country, or gas stations in the country.

A *variable* is some measurement or characteristic of a case. For example, with students in a classroom the last test grade; for likely voters their party affiliation, if any; and for gas stations, their current price per gallon.

A *univariate data set* is then a set of measurements for some variable from a collection of cases. We use the subscript notation to represent such a data set:

$$x_1, x_2, \ldots, x_n.$$

The subscript gives an implicit order to the data, which is basically a way to keep track of which case the measurement is for.

Levels of measurement In 1946, Stanley Smith Stevens [54] posited an influential description of various types of data. His ordering consisted of data being:

nominal Such data is qualitative or descriptive, but not numeric. An example might be the name of a person or the town they are from, or the number on a bib a runner wears in a race.

ordinal Ordinal data is data with some order, so that we can sort the data from largest to smallest, say. An example might be the place a runner takes in a race.

interval Interval data is ordinal data where the difference between two values has some interpretation. The clock time a person finishes might be an example. If we know runner A finishes at noon, and runner B at 1PM, then we know that runner B took longer. Since we haven't specified when they started, we don't know what percent longer though.

ratio Ratio data has a meaningful 0. If we record not the time of finishing, but the time since starting, then 0 has a meaning and we can take a ratio of the total time for runner A and B to compare the two.

Though this taxonomy has permeated many textbooks, especially in the social sciences, we prefer to characterize data to match how we work with it on the computer. We have:

factor data When we look at many variables, some may simply record categories used to group the data. In R we will use *factors* to store these variables. An example might be the browser a user has used to view a web site, as gleaned from a web site log. Such information is important to web programmers.

character data Some categorical data are factors, but others are really just identifiers, and are not used for grouping. An example might be a user's IP address. This is basically a unique code identifying a computer, like an address. The distinction between factor and character data can be thought of as the distinction between *categorizing* a case or *characterizing* a case. While both factor and categorical data are "nominal" we keep the distinction as we will interact with such data in R differently.

discrete data Discrete data comes from measurements where there are essentially only distinct and separate possible values that can be counted. For example, the number of visits a person makes to our web site will always be integer data, as will other counting data.

continuous data Continuous data is that which could conceivably come from a continuum of values. The recording of the time in milliseconds of a vist to a web site might be such data. A useful distinction is that for discrete data we expect that cases will share values, whereas for continuous data this will be impossible, or at least very unlikely. There is no fine line though. We can always turn continuous data into discrete data just by truncating (e.g., recording the minute, not the millisecond of a visit) or by binning. Rather than draw distinctions for numeric data between ordinal, interval or ratio, it is more important for statistical theory—in finding a model for the recorded data—to know if the data is discrete or continuous.

date and time data Though we just saw that time data can be considered continuous or discrete depending on resolution, for computers there are often separate ways entirely to handle date and time data. Issues

that complicate matters are as familiar as leap days and time zones, but there are even more subtleties. For example, scale. People in finance want millisecond data, but over long time ranges this recording can literally run out of numbers on a computer. Astronomers need precise measurements for durations down to leap seconds. R has several ways to work with such data, that go beyond just storing the values as simple numbers.

hierarchical data The traditional idea of a data set being several measurements for different cases is widely established. Such data nicely fits into a spreadsheet in a rectangular manner, and in tables of data bases. However, this structure doesn't fit well for every data set of interest. For example, data on networks. We don't discuss such examples in the text, but include it here to point out that this list is far from comprehensive.

This chapter will begin with some R basics for working with data sets and then proceed to look at various summaries—numeric and graphical—of different types of data.

2.1 Data vectors

Our notation for a data set, $x_1, x_2, \ldots, x_n$ suggests already a few things: there are n items, and, by using a common name, they are all measurements of the same type. R's basic data structure is perfectly suited for this. The `c` function can be used to bundle our data set together.

Suppose the number of whale beachings in Texas during the 1990s was

```
74 122 235 111 292 111 211 133 156 79
```

We can combine these values into a data set with R through:

```
whale <- c(74, 122, 235, 111, 292, 111, 211, 133, 156, 79)
```

We just separate values with a comma. We refer to the `whale` object as a *data vector*.[1] It has the same properties as a data set: a size and a common type for the values.

The size of the data set is the "n", and is retrieved in R with the `length` function:[2]

```
length(whale)

## [1] 10
```

[1] Technically, what we call "data vectors" are more commonly referred to as just "vectors," a more common mathematical usage. They are generalized by adding dimensions to create matrices and arrays.

[2] Both `c` and `length` have a much wider usage. Here we describe their specialization to data vectors.

The number of elements is of interest, and R shows a count when it prints a vector. This is the `[1]` that appears before the output. As even single numbers (scalars) are treated as vectors of length 1, these too are prefixed, as above. When the output spans more than one row, each row is prefixed by the index of the first member of the row.

There are, of course, many other useful functions in R to extract information from a data vector. For example, as seen, we can get the total number using `sum`:

```
sum(whale)

## [1] 1524
```

The average number can be found by combining these two:

```
sum(whale)/length(whale)

## [1] 152.4
```

Or, more commonly, this is done directly through the `mean` function:

```
mean(whale)

## [1] 152.4
```

Vectorization As mentioned, the arithmetic operations and the mathematical functions are vectorized, in that they will be called for each element in a data vector. Where `sum`, `length`, and `mean` are reductions, in that when they are called with a vector they reduce it down to a number, vectorized functions maintain the size of the vectors involved, possibly after recycling to reach a consistent length. Above, the division by `length(whale)` used vectorization.

Here are more examples, all returning a vector of length 10:

```
whale - mean(whale)                    # mean(whale) recycled

##  [1] -78.4 -30.4  82.6 -41.4 139.6 -41.4  58.6 -19.4   3.6 -73.4

whale^2 / length(whale)

##  [1]  547.6 1488.4 5522.5 1232.1 8526.4 1232.1 4452.1 1768.9
##  [9] 2433.6  624.1

sqrt(whale)

##  [1]  8.602 11.045 15.330 10.536 17.088 10.536 14.526 11.533
##  [9] 12.490  8.888
```

Missing values, NA Some data sets are not complete. In the initial examples, we discussed a *New York Times* article on a study [33] to look at the price of hip replacements. Simulated data from 15 hospitals is given here:

```
10,500 45,000 74,100 NA     83,500 86,000 38,200 NA
44,300 12,500 55,700 43,900 71,900 NA     62,000
```

Here we use, as the authors did, the value "NA" (not available) for data that was not available. In this case, the surveyed hospitals could not provide the information on the cost of a total hip replacement.

Such situations are quite common. R provides the special variable `NA` to represent values that are missing. Using this, to enter the above data set into R could be done with:

```
hip_cost <- c(10500, 45000, 74100, NA, 83500, 86000, 38200, NA,
              44300, 12500, 55700, 43900, 71900, NA, 62000)
```

The value `NA` is interpreted as the value is missing, but could possibly be there. As such, it would be incorrect to assume it has no value—it is just unknown. This effects how the data is used. For example, we could try and take the total costs, as perhaps we work for an insurer and are attempting to estimate a total exposure:

```
sum(hip_cost)

## [1] NA
```

We see then this sum is also not available. (It can be said that `NA` values poison subsequent operations.) Many R functions have an argument to specify what to do with missing data. For `sum` this argument is `na.rm` (remove `NA` values). The default is on the side of caution, but we can specify `TRUE` to change that:

```
sum(hip_cost, na.rm = TRUE)

## [1] 627600
```

The `mean` function has the same argument. A consumer may want to know if a quoted charge is out of line. The following shows the average costs for those hospitals that reported a value is just over $52,000:

```
mean(hip_cost, na.rm=TRUE)

## [1] 52300
```

For multivariate data sets, and the functions that interact with them, we will see there are generally more options for dealing with `NA` values.

NULL Somewhat related to NA is the value NULL. NULL is a reserved value usually indicating that some requested action is undefined or unavailable.

NaN, Inf The value NaN looks like NA, but is different. This value arises from arithmetic operations that are undefined, such as 0/0, or unrepresentable. The value Inf stands for infinity, and comes from evaluations such as 1/0. Inf can be positive or negative and can be used as expected with R's ordering operators, like <. (Comparisons with NaN produce NA.) These values are part of the specification for floating point numbers implemented by R.

Attributes: names Data vectors may have attributes. The "names" attribute allows the association of a name with each case. For example, the built-in precip data set lists the average rainfall in inches for 70 cities in the United States. Here we list the first 6 using the head function:[3]

```
head(precip)

##      Mobile      Juneau     Phoenix Little Rock Los Angeles
##        67.0        54.7         7.0        48.5        14.0
##  Sacramento
##        17.2
```

The city names are printed above the values. Clearly this is useful information when looking at the data, as otherwise we'd need to be familiar with how the index matches up with the city. In fact, operations may make tracking the index mentally quite challenging. Here we sort the data biggest to smallest to find the top 6 rainiest cities:

```
head( sort(precip, decreasing=TRUE) )

##       Mobile        Miami     San Juan  New Orleans       Juneau
##         67.0         59.8         59.2         56.8         54.7
## Jacksonville
##         54.5
```

The names function returns the names associated with a data vector. Here we show the first 6 names—without the associated data:

```
head(names(precip))

## [1] "Mobile"      "Juneau"      "Phoenix"     "Little Rock"
## [5] "Los Angeles" "Sacramento"
```

[3]The head function is used to show the first k elements, with $k = 6$ by default; the tail function is used to show the last k elements, again with $k = 6$ by default. The headtail function is provided by UsingR to show the first and last k elements with ellipses separating them with a default of $k = 3$.

Creating named components The setting of names is so common, that there are several means to do so. For example, we can specify the values through `c` using what is essentially a `key=value` syntax:

```
test_scores <- c(Alice = 87, Bob = 72, Shirley = 99)
```

Quoting the key values is optional, though necessary if a name contains spaces.

Alternatively, the `setNames` function can be used to set the names. This is more commonly used with programming, as you'll see the syntax is a bit more verbose:

```
test_scores <- setNames(c(87, 82, 99), c("Alice", "Bob", "Shirley"))
```

Assignment functions In R there are several functions which take on two forms, one to "get" and one to "set" values. The names of these functions come in pairs, for example `names` and `names<-`. The latter is used for assignment. Though they look similar when used, the setting operator appears on the left-hand side of the assignment, as below.[4]

```
test_scores <- c(87, 782, 99)
names(test_scores) <- c("Alice", "Bob", "Shirley")
test_scores

##   Alice     Bob Shirley
##      87     782      99
```

We used a character vector for the names. The assignment functions are typically called through `fun(x) <- value` to set the value for `x`. This notation can be a bit confusing at first, as it doesn't fall into the typical `x <- value`.

Coercion As mentioned a univariate data set, being comprised of similar measurements, should have values of the same basic type. This need not be the case when entered into a data vector:

```
x <- c(1, "two", "III")
x

## [1] "1"   "two" "III"
```

Note how `x` is printed: all the values are in quotes. What happened? Data vectors too must all be of the same type when stored in R.[5] Here R silently

[4] The assignment functions mutate the values assigned to a variable, unlike most other functions in R which do not.

[5] In R, a list is a generalized vector where each component may have a different type.

coerced, or converted, the numeric value 1 into the string "1". Basically, if we try to combine different types of objects through c, R will promote each element to a common type, which often will be a character type.

One common mistake is to use the character string "NA" instead of the NA variable itself when specifying missing values. Such a mistake will silently convert all values to character.

The coercion can also be done through the "as" functions. Here, we show how to coerce back and forth between numbers and characters:

```
as.numeric("1")

## [1] 1

as.character(1)

## [1] "1"
```

There are several other specific "as" functions and one simply named as to do general coercions.

scan There are many alternatives to c for entering data into R. We wait until Chapter 4 to discuss most of these, as they often result in a data frame, which are discussed therein. However, one way to read data in from the command line or a file that returns a data vector is the scan function. This function has many arguments, we mention just its first which allows us to indicate if the function should scan the keyboard input or scan the contents of a file. If left empty, it will scan the keyboard until an empty line is specified. With scan we can separate values by spaces:

```
scan()
1: 74 122 235 111 292 111 211 133 156 79
11:

Read 10 items
 [1]  74 122 235 111 292 111 211 133 156  79
```

In the above, we copied and pasted in the values and then added a new line to indicate we were done. This output the 10 values we specified, but did not save them, as we did not assign the values to a name.

Alternatively, we can scan values from a file. For example, suppose the file whale.txt contained the same data. This command will read in the file and store its scanned values in the whale variable.

```
whale <- scan("whale.txt")              # or scan(file.choose())
```

We specified a file name on the working directory. If that is not where the data file is, it can be convenient to browse for the values. The `file.choose` function will do just that.

Structured data

There are some convenient functions for generating structured data. For simple sets of contiguous integers the colon operator, `:`, may be used:

```
1:5                                  # 1 2 3 4 5

## [1] 1 2 3 4 5

1:length(whale)                      # 1 2 3 ... 10

##  [1]  1  2  3  4  5  6  7  8  9 10

0:(length(whale) - 1)                # 0 1 2 ... 9

##  [1] 0 1 2 3 4 5 6 7 8 9
```

The value returned by the command `a:b` is a sequence from `a` to `b` with a step size of 1 and not exceeding `b`. We can have `b` being less than `a`, in which case the sequence counts down. The second example shows a common construct used when we want to reference all values of a data vector by its indices.[6] The third example illustrates that parentheses are needed to do arithmetic on the values of `a` and `b`, as the colon operator has higher precedence, it will be done before the subtraction unless otherwise directed through parentheses.

Sequences increasing by 1 are *arithmetic sequences*, which more generally can increase (or decrease) by a given step size, say `h` (which could be negative). To generate the values a, $a + h$, $\ldots$, $b = a + nh$, the `seq` function is convenient. Such a sequence is returned through `seq(a, b, by=h)`, or using positional arguments just `seq(a, b, h)`. To count by 10s to 100 we could have:

```
seq(0, 100, by=10)                   # count by 10s

##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

The value of `b` is a suggestion and if the argument for `by` does not allow `b-a` to be written as a multiple of `h`, `b` won't be included (or exceeded) in the returned vector.

[6] For programming, it is suggested to use the function `seq_along` which is basically `1:length(x)`, but handles more gracefully the case where the data vector is of 0-length.

To return a sequence of a given length between `a` and `b` can be done by choosing the `by` value appropriately, but is much easier done by specifying the desired length, through the `length.out` argument:

```
seq(0, 100, length.out=11)              # counts by 10s as well

##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

The `rep` function The `rep` function can be used to repeat values. The basic form is to repeat the first argument, the number of times specified in the second:

```
rep(5, times=10)                        # 10 5s

##  [1] 5 5 5 5 5 5 5 5 5 5
```

That can be convenient. The values can be vectorized, for much more complicated patterns. To get the pattern 1, 2, 3 four times we have:

```
rep(1:3, times=4)

##  [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

Whereas to get the pattern 1, 1, 1, 2, 2, 3 we have:

```
rep(c(1,2,3), times=c(3,2,1))           # or rep(1:3, 3:1)

## [1] 1 1 1 2 2 3
```

When the `times` argument is a vector of length matching the length of `x`, then it specifies how many times each corresponding value in `x` should be repeated.

Indexing

The subscript in our data set notation, $x_1, x_2, \ldots, x_n$, indicate that data has a notion of size and order. R's data vectors allow us to access and assign to parts of a data vector using indices. As in our notation, the first element is indexed by a 1.[7] There are several different ways that we can extract parts of a data vector: by numeric index, by name, by a matching-sized logical vector (c.f. Table 2.1). We will come back to the latter.

[7] Several computer languages use 0-based indices.

Numeric indices The notation for indexing is the square bracket `[` in contrast to how functions utilize parentheses.[8] Inside the paired brackets go the specific index. The simplest case is just a single number:

```
whale                                   # all the values

##  [1]  74 122 235 111 292 111 211 133 156  79

whale[1]                                # first element

## [1] 74

whale[2]                                # second element

## [1] 122

whale[11]                               # 11th element is not there

## [1] NA
```

The last command, `whale[11]`, illustrates what happens when an attempt to access an index beyond the length of the data vector. R does not raise an error, but rather returns `NA`.

We are not limited to single indices, the index vector can be as long as desired:

```
whale[c(1,3,5,7,9)]

## [1]  74 235 292 211 156
```

Here we see that the indexing operation is vectorized, and all the corresponding values are returned.

There is a convenient convention of *negative indexing* which extracts *all but* the specified values. This can be quite convenient. The one caveat, we cannot use both negative and positive indices at once. This call will return all but the first element of `whale`:

```
whale[-1]

## [1] 122 235 111 292 111 211 133 156  79
```

Indexing by `0` returns a 0-length data vector, as does indexing by a 0-length vector. If no index is specified, as in `whale[]`, the entire data vector is returned.

[8] We will see that lists also use a double square bracket, `[[`.

Indexing with names For data vectors with a names attribute, we can reference values by their name, instead of position. For example, the average amount of rain in Seattle and New York is given by:

```
precip[c("Seattle Tacoma", "New York")] # which is rainier?

## Seattle Tacoma       New York
##           38.8           40.2
```

The `match` function is used to find where a value is in a data set. To find out the corresponding indices for these names, one could use the following construct and then index by number:

```
match(c("Seattle Tacoma", "New York"), names(precip))

## [1] 65 42
```

This illustrates the greater ease of using the names directly. There is no negative indexing with names.[9] As well, when a name does not exist, the value of `NA` is given:

```
precip["Seattle"]                   # needs "Seattle Tacoma" to match

## <NA>
##   NA
```

Later, we will see that we can index by a logical vector in addition to numeric indices or names.

Assignment through indexing The assignment function `[<-` allows us to assign to parts of a data vector. The simplest case is when the length of the indexing vector matches the length of the assigned values. For example, to change the first value of a data set can be done through:

```
x <- c(1, 2, 3)
x[1] <- 11                              # 11 is now first value
x

## [1] 11  2  3
```

The index can be a vector, so the following works:

[9] The `subset` function which does have some features for negative indexing with names will be discussed in Chapter 4.

Command	Description
`x[1]`	The first element of `x`.
`x[]`	All elements of `x`.
`x[length(x)]`	The last element of `x`.
`x[c(2,3)]`	The second and third elements of `x`.
`x[-c(2,3)]`	All but the second and third elements of `x`.
`x[0]`	0-length vector of same type as `x`.
`x[1] <- 5`	Assign a value of 5 to first element of `x`.
`x[c(2,3)] <- c(4,5)`	Assign values to second and third elements of `x`. In assignment, recycling of the right-hand side may occur. Assignment can grow the length of a data vector.

Table 2.1: Various uses of indexing by numeric indices.

```
x[2:3] <- c(12, 13)
x

## [1] 11 12 13
```

We can extend the size of a data vector through assignment. "Holes" will be filled in by `NA`s:

```
x[6] <- 6
x

## [1] 11 12 13 NA NA  6
```

To reduce the size of a data vector we simply reassign the variable using a subset of the data vector, e.g. `x <- x[1:2]`.

Recycling When the right-hand side has fewer elements than the data vector referred to on the left-hand side, R will recycle the value on the right-hand side. This makes it easy to assign many values at once. For instance to set the last two values to `0` in `x`, we have:

```
x[2:3] <- 0
x

## [1] 11  0  0 NA NA  6
```

The vector `0` is recycled to `c(0,0)` then assigned. When the right-hand side is a single value, recycling is easy to understand. When it is not, then the right-hand side is repeated a sufficient number of times:

```
x <- 1:10
x[] <- 1:3                                  # 10 on left, 3 on right

## Warning: number of items to replace is not a multiple of replacement
length

x

##  [1] 1 2 3 1 2 3 1 2 3 1
```

R gives a warning about the 3 (the length of the right-hand side) not being a multiple of 10 (the number of items referenced through `x[]`, but does as requested).

Recycling isn't magic, just very convenient. If could be done manually with the `rep` function—just repeat the vector a given number of times. For recycling that is not a multiple of the desired length, we can use a construct like the following. The `%%` operator finds the remainder after division, and then these values are shifted by 1 to get the corresponding index:

```
n <- 10                                     # get 10 elements of x
x[1 + 0:(n-1) %% length(x)]                 # use remainder for indices

##  [1] 1 2 3 1 2 3 1 2 3 1
```

Data types

So far, we have used numeric and character data in the examples. Of course, there can be many other types of data possible. To organize this, R assigns a *class* attribute to most R objects and otherwise creates an implicit class for an object. The class of an object is used to determine how it should be printed.[10] The `class` function will return the class of an object. For most objects, this is a single character, but may be a character vector.

Numeric data types

The two main classes for numeric data are `numeric` and `integer`, though there are others, e.g. `complex`. Most of the time numbers are `numeric`. For example, to see that all of these objects are numeric:

```
c(class(1), class(pi), class(seq(1, 5, by=1)))

## [1] "numeric" "numeric" "numeric"
```

[10] Indeed, the class of an object decides which function definition should be used for many different functions, not just `print` and `show`.

To make an integer value, we need to work a bit: we can preallocate space for an integer data set of length `n` with `integer(n)`; we can use the suffix `L` to force a number to be treated as an integer (e.g., `1L`); we can coerce numeric values of integer type through the `as.integer` function.

To the casual R user, the distinction is not important. Integers are stored differently. They are precisely known, but have a limited range, roughly between $\pm 2 \cdot 10^9$. Numeric values are stored using floating point representation. This format can store much larger integer values and has a much wider range of numbers it can represent.[11]

However, floating point representation cannot store all numbers exactly. For the data sets in this text, we won't need to think about this. Though if we try to be too literal, the impact can pop up. For example, the square root of 2 is irrational and the floating point representation is an approximation. It shows in the second of these expressions:

```
sqrt(2) * sqrt(2)                       # looks right

## [1] 2

sqrt(2) * sqrt(2) - 2                   # a difference

## [1] 4.441e-16
```

In the last command, we see that there is a small difference between the two numbers—out in the 16th decimal point. This difference can be an issue when checking equality of values.

Categorical data types

R has two distinct classes for working with categorical data: `factor` and `character`. As mentioned, to distinguish the two: factors are used to classify values, character data is used to characterize values.

Character data Character data is created just by quoting values. Quotes can be matching pairs of single or double quotes, though double quotes are preferred and used to display character values. Within a quoted value a quote symbol can be used, but it must be escaped by prefixing it with a backslash (cf. `?Quotes` for more details.)

[11] We can think of each floating point number as stored with three parts: a sign (± 1), an exponent (e, with $k = K - e$ for some K), and a precision, p. Then we can use scientific notation to represent each number through $\pm 1 \cdot p10^k$. In R—like other computer languages—scientific notation is printed with an `e` to indicate the exponent, e.g., `3.141593e+13` or `-3.183099e-14`. The format also allows for the representation of $\pm\infty$: `Inf`, `-Inf`; a value for "not a number:" `NaN`; and plus or minus 0.