

THE EXPERT'S VOICE® IN R PROGRAMMING

# R Recipes

A Problem-Solution Approach

Larry A. Pace

Apress®

*For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.*



# Contents at a Glance

|   |             |
|---|-------------|
| <b>About the Author .....</b>   | <b>xiii</b> |
| <b>About the Technical Reviewer .....</b>                                   | <b>xv</b>   |
| <b>Acknowledgments .....</b>  | <b>xvii</b> |
| <b>Introduction .....</b>   | <b>xix</b>  |
| <b>■ Chapter 1: Migrating to R: As Easy As 1, 2, 3 .....</b>                | <b>1</b>    |
| <b>■ Chapter 2: Input and Output .....</b>                                  | <b>17</b>   |
| <b>■ Chapter 3: Data Structures .....</b>                                   | <b>27</b>   |
| <b>■ Chapter 4: Merging and Reshaping Datasets .....</b>                    | <b>43</b>   |
| <b>■ Chapter 5: Working with Dates and Strings .....</b>                    | <b>57</b>   |
| <b>■ Chapter 6: Working with Tables.....</b>                                | <b>67</b>   |
| <b>■ Chapter 7: Summarizing and Describing Data .....</b>                   | <b>79</b>   |
| <b>■ Chapter 8: Graphics and Data Visualization .....</b>                   | <b>89</b>   |
| <b>■ Chapter 9: Probability Distributions .....</b>                         | <b>107</b>  |
| <b>■ Chapter 10: Hypothesis Tests for Means, Ranks, or Proportions.....</b> | <b>117</b>  |
| <b>■ Chapter 11: Relationships Between and Among Variables.....</b>         | <b>143</b>  |
| <b>■ Chapter 12: Contemporary Statistical Methods .....</b>                 | <b>157</b>  |
| <b>■ Chapter 13: Writing Reusable Functions .....</b>                       | <b>167</b>  |

■ **Chapter 14: Working with Financial Data** ..... **183**

■ **Chapter 15: Dealing with Big Data** ..... **201**

■ **Chapter 16: Mining the Gold in Data and Text** ..... **215**

**Index**..... **237**

# Introduction

R is an open source implementation of the programming language S, created at Bell Laboratories by John Chambers, Rick Becker, and Alan Wilks. In addition to R, S is the basis of the commercially available S-PLUS system. Widely recognized as the chief architect of S, Chambers in 1998 won the prestigious Software System Award from the Association for Computing Machinery, which said Chambers' design of the S system "forever altered how people analyze, visualize, and manipulate data."

Think of R as an integrated system or environment that allows users multiple ways to access its many functions and features. You can use R as an interactive command-line interpreted language, much like a calculator. Type a command, press Enter, and R provides the answer in the R console. R is simultaneously a functional language and an object-oriented language. In addition to thousands of contributed packages, R has programming features, just as all computer programming languages do, allowing conditionals and looping, and giving the user the facility to create custom functions and specify various input and output options.

R is widely used as a statistical computing and software environment, but the R Core Team would rather consider R an environment "within which many classical and modern statistical techniques have been implemented." In addition to its statistical prowess, R provides impressive and flexible graphics capabilities. Many users are attracted to R primarily because of its graphical features. R has basic and advanced plotting functions with many customization features.

Chambers and others at Bell Labs were developing S while I was in college and grad school, and of course I was completely oblivious to that fact, even though my major professor and I were consulting with another AT&T division at the time. I began my own statistical software journey writing programs in Fortran. I might find that a given program did not have a particular analysis I needed, such as a routine for calculating an intraclass correlation, so I would write my own program. BMDP and SAS were available in batch versions for mainframe computers when I was in graduate school—one had to learn Job Control Language (JCL) in order to tell the computer which tapes to load. I typed punch cards and used a card reader to read in JCL and data.

On a much larger and very much more sophisticated scale, this is essentially why the computer scientists at Bell Labs created S (for *statistics*). Fortran was and still is a general-purpose language, but it did not have many statistical capabilities. The design of S began with an informal meeting in 1976 at Bell Labs to discuss the design of a high-level language with an "algorithm," which meant a Fortran-callable subroutine. Like its predecessor S, R can easily and transparently access compiled code from various other languages, including Fortran and C++ among others. R can also be interfaced with a variety of other programs, such as Python and SPSS.

R works in batch mode, but its most popular use is as an interactive data analysis, calculation, and graphics system running in a windowing system. R works on Linux, PC, and Mac systems. Be forewarned that R is not a point-and-click graphical user interface (GUI) program such as SPSS or Minitab. Unlike these programs, R provides terse output, but can be queried for more information should you need it. In this book, you will see screen captures of R running in the Windows operating system.

According to my friend and colleague, computer scientist and bioinformatics expert Dr. Nathan Goodman, statistical analysis essentially boils down to four empirical problems: problems involving description, problems involving differences, problems involving relationships, and problems involving classification. I agree wholeheartedly with Nat. All the problems and solutions presented in this book fall into one or more of those general categories. The problems are manifold, but the solutions are mostly limited to these four situations.

## What this Book Covers

This book is for anyone—business professional, programmer, statistician, teacher, or student—who needs to find a way to use R to solve practical problems. Readers who have solved or attempted problems similar to the ones in this book using other tools will readily concur that each tool in one’s toolbox works better for some problems than for others. R novices will find best practices for using R’s features effectively. Intermediate-to-advanced R users and programmers will find shortcuts and applications that they may not have considered, as well as different ways to do things they might want to do.

## The Structure of this Book

The standardized format will make this a useful book for future reference. Unlike most other books, you do not have to start at the beginning and go through this book sequentially. Each chapter is a stand-alone lesson that starts with a typical problem (most of which come from true-life problems that I have faced, or ones that others have described and have given me permission to share). The datasets used with this book to illustrate the solutions should be similar to the datasets readers have worked with, or would like to work with.

Apart from a few contrived examples in the early chapters, most of the datasets and exercises come from real-world problems and data. Following a bit of background, the problem and the data are presented, and then readers learn one efficient way to solve the problem using R. Similar problems will quickly come to mind, and readers will be able to adapt what they learn here to those problems.

## Conventions Used in this Book

In this book, code and script segments will be shown this way:

```
> x <- c(1, 3, 5)
> px <- c(0.5, 0.25, 0.25)
> dist <- sample(x, size = 1000, replace = TRUE, prob = px)
>
```

Code and R functions written inline will also be formatted in the code style.

When you are instructed to perform a command within the R Console or R Editor by using the (limited) point-and-click interface, the instructions will appear as follows: File ► Workspace.

## Looking Forward

In Chapter 1, you will learn how to get R, how R works, and some of the basic things you can do with R. You will learn how to work with the R interface and the various windows you will find in R. Finally, you will learn how R deals with missing data, vectors, and matrices.



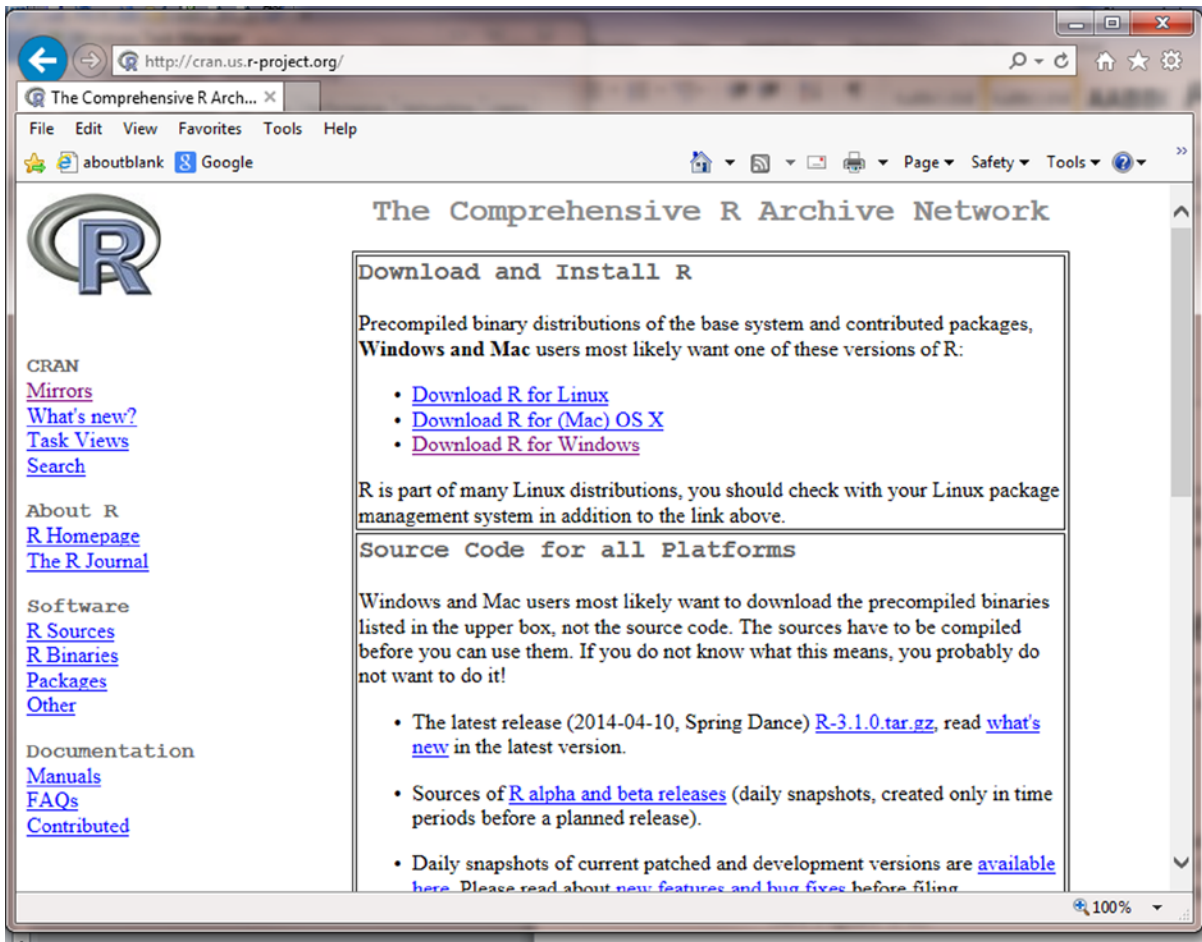
# Migrating to R: As Easy As 1, 2, 3

There are compelling reasons to use R. An enthusiastic community of users, programmers, and contributors support R and its evolution. R is accurate, produces excellent graphs, has a variety of built-in functions, and is both a functional language and an object-oriented one. R is completely free and is distributed as open-source software. Here is how to get started. It really is as easy as 1, 2, 3.

## Getting R Up and Running on Your System

The current version at the time of this writing was R 3.1.0. A recent version needs to be available on your computer in order for you to benefit from the R recipes you will learn in this book. Many users migrate to R from other statistical packages, while other users migrate to R from other programming languages. Both types of users are in for a bit of a shock. R is a programming language, but very much unlike most other ones. R is not exactly a statistics package, but rather an environment that includes many traditional statistical analyses. This is neither a statistics book nor an R programming book, though we will cover elements of both when solving problems within the recipes contained in this book.

Visit the Comprehensive R Archive Network (<http://cran.us.r-project.org/>); see the screen capture in Figure 1-1. Users of PCs and Macs can download precompiled binary files, whereas Linux users may have to do the compiling on their own. However, many Linux systems have R as part of their distributions, so Linux users may already have R preinstalled (I'll show you how to check this later in this section).

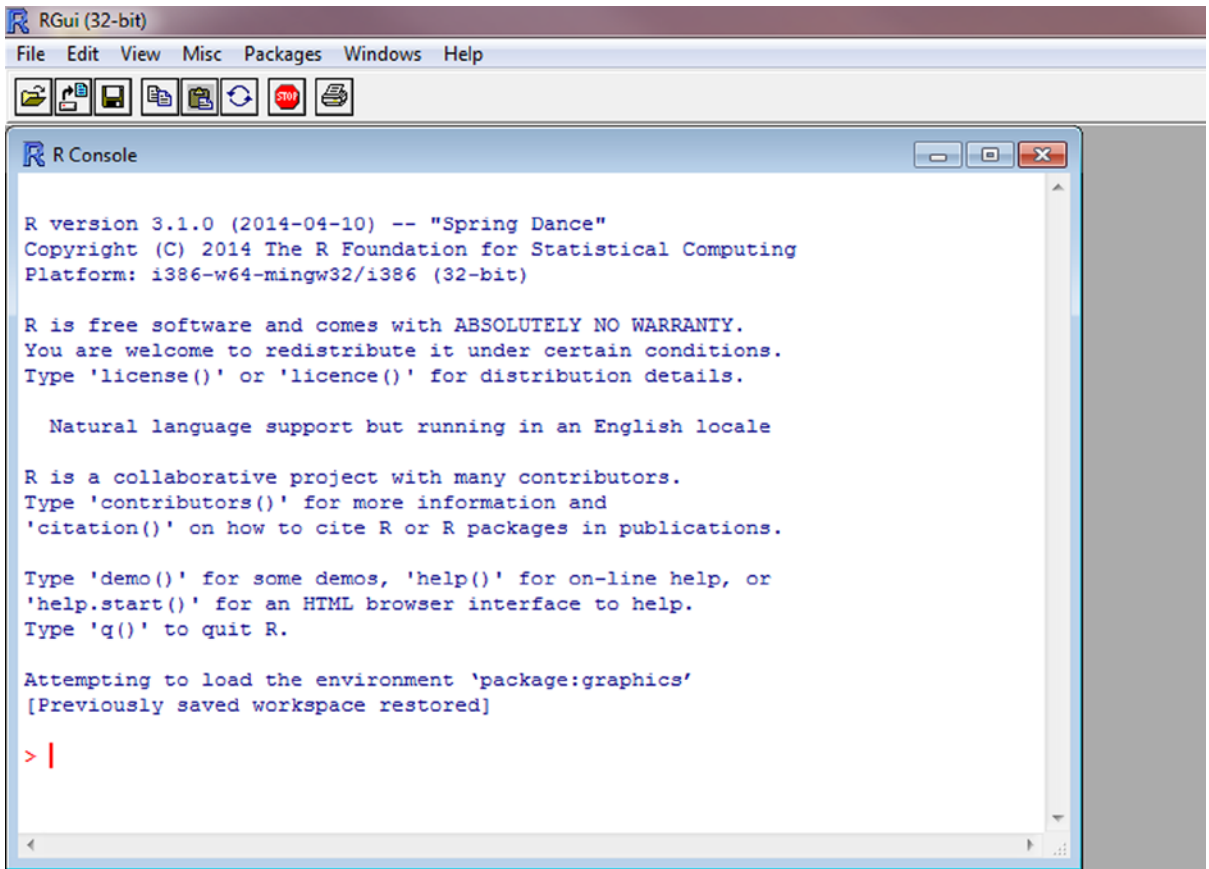


**Figure 1-1.** The Comprehensive R Archive Network

Click **Mirrors** and select the site closest to you. Download the precompiled binary files for your system or follow the instructions for compiling the source code if you need to do so. If you have never installed R, install the base distribution first. Most users of Windows will be able to use the 32-bit version of R. If you want to explore the advantages and disadvantages of using the 64-bit version (assuming you have a 64-bit Windows system), look at the information provided by the R Project to help you choose. You can also do what I did, and install both the 32-bit and the 64-bit versions.

Choose your installation language and options. The defaults are fine for most users. If the R installation was successful, you will have a directory labeled R and a desktop icon for launching R. Figure 1-2 shows the opening screen of R 3.1.0 in a Windows 7 environment.





**Figure 1-2.** The R Console appears in the R GUI

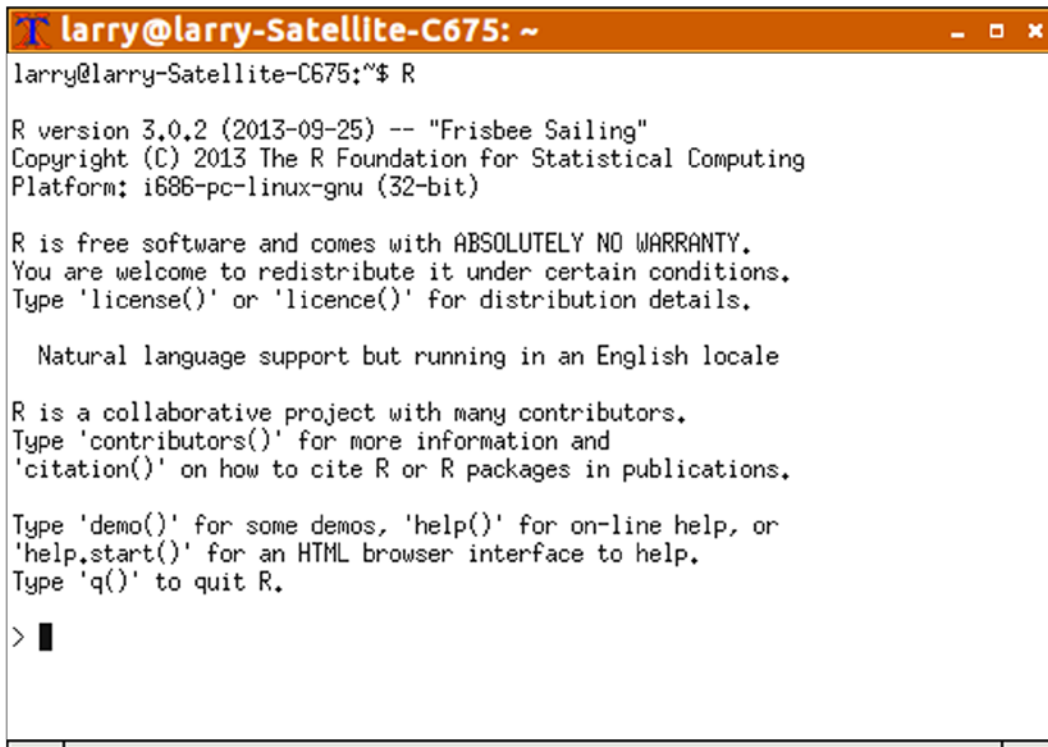
As I mentioned, Linux users may have to compile the R source code, but should first check to see if R is distributed with their version of Linux. For instance, I use Ubuntu, a distribution of Linux, on one of my computers, and the base version of R comes prepackaged with Ubuntu, as it does with most Ubuntu versions. To see if you have R base in your Linux system, use the following commands. Open a terminal session. The command prompt in Linux is the tilde character (~) followed by the dollar sign (\$).

```
~$: sudo apt-get install r-base
```

Once you have installed the base version of R, you can run R from the terminal as follows:

```
~$: R
```

Note that the Linux version of R is not likely to be the latest one, as I am currently running R 3.0.2 in Linux (see Figure 1-3).



```

larry@larry-Satellite-C675: ~
larry@larry-Satellite-C675:~$ R

R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: i686-pc-linux-gnu (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> █

```

**Figure 1-3.** R running in a Linux system (Lubuntu)

As you see in Figures 1-2 and 1-3, the command prompt in R is `>`. The following section will show you how to take R for a quick spin.

## Okay, So I Have R. What's Next?

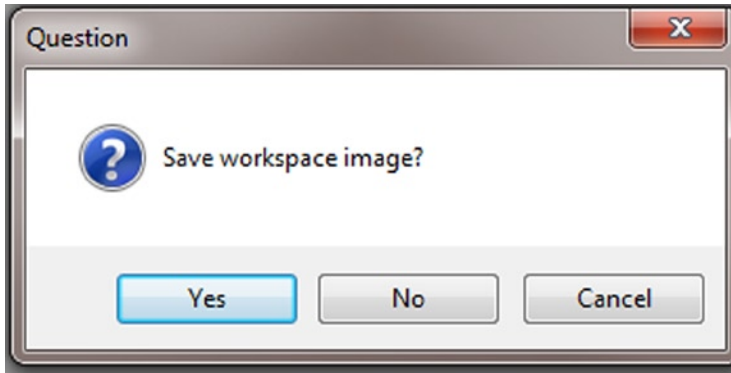
Whether you are a programmer or a statistician, or like me, a little of both, R takes some getting used to. Most statistics programs, such as SPSS, separate the data, the syntax (programming language), and the output. R takes a minimalist stance on this. If you are not using something, it is not visible to you. If you need to use something, either you must open it, as in the R Editor for writing and saving R scripts, or R will open it for you, as in the R Graphics Device when you generate a histogram or some other graphic output. So, let's see how to get around in the R interface.

A quick glance shows that the R interface is not particularly fancy, but it is highly functional. Examine the options available to you in the menu bar and the icon bar. R opens with the welcome screen shown in Figure 1-2. You can keep that if you like (I like it), or simply press **Ctrl+L** or select **Edit ► Clear Console** to clear the console. You will be working in the R Console most of the time, but you can open a window with a simple text editor for writing scripts and functions. Do this by selecting **File ► New script**. The built-in R Editor is convenient for writing longer scripts and functions, but also simply for writing R commands and editing them before you run them. Many R users prefer to use the text editor of their liking. For Windows users, Notepad is fine. When you produce a graphic object, the R Graphics Device will open. The R GUI (graphical user interface) is completely customizable as well.

Although we are showing R running in the R Console, you should be aware that there are several integrated development environments (IDEs) for R. One of the best of these is RStudio.

Do not worry about losing your output when you clear the console. This is simply the view of what you have on the screen at the moment. The output will scroll off the window when you type other commands and generate new output. Your complete R session is saved to a history file, and you can save and reload your R workspaces. The obvious advantage of saving your workspace is that you do not have to reload the data and functions you used in your R session. Everything will be there again when you reload the workspace.

You will most likely not be interested in saving your R workspace with the examples from this chapter. If you do want to save an R workspace, you will receive a prompt when you quit the session. To exit the session, enter `q()` or select **File ► Exit**. R will give you the prompt shown in Figure 1-4.



**Figure 1-4.** R prompts the user to save the workspace image

From this point forward, the R Console is shown only in special cases. The R commands and output will always appear in code font, as explained in the introduction. Launch R if it is not already running on your system. The best way to learn from this book is to have R running and to try to duplicate the screens you see in the book. If you can do that, you will learn a great deal about using R for data analysis and statistics.

First, we will do some simple math, and then we will do some more interesting and a little more complicated things. In R, one assigns values to objects with the *assignment operator*. The traditional assignment operator is `<-`. There is also a little-used right-pointing assignment operator, `->`. You can also use the equals sign for assignments. There is some advantage in that you avoid two keystrokes when you use `=` instead of `<-`. In this book, we will always use `<-` for assignments. The `=` sign is used to specify values for arguments and options in R commands. To test for equality, use `==`.

R accepts numbers, characters, variables, and even other functions as input to its functions. R is unlike other languages in several important ways. In most computer languages, a number can be assigned to a constant, usually with an equal sign, `=`. For example, in Python, you can make the assignment `x = 10`. The value of 10 is assigned to the variable `x`. The “type” of `x` is a scalar quantity (a single value) stored as an integer:

```
Python 3.3.1 (v3.3.1:d9893d13c628, Apr 6 2013, 20:25:12) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information>>> x = 10
>>> x
10
>>> type(x)
<class 'int'>
```

If you will remember some of your mathematical or computer training, recall that numerical data can be *scalars* (individual values or constants), *arrays* (or vectors) with one row or one column of numbers, or *matrices* with two or more rows and two or more columns. Many computer languages make distinctions among these data types. In some languages, which are called “strongly typed,” you must declare the variable’s type and dimensionality before you

assign a value or values to it. In other languages, known as “loosely typed,” You can assign different types of values to the same variable without having to declare the type. R works that way, and is a very loosely typed language.

To R, there are no scalar quantities. When you enter `1 + 1` and then press **Enter**, R displays `[1] 2` on the next line and gives you another command prompt. The index `[1]` indicates that to R, the integer object 2 is an integer vector of length 1. The number 2 is the first (and only) element in that vector. You can assign an R command to a variable (object), say, `x`, and R will keep that assignment until you change it. When we assign `x <- 1 + 1`, the value of 2 is assigned to the object `x`. We can now use `x` in R commands, such as `x + 1`. R's indexes start with 1 instead of 0, as some other computer languages do. If you type **numbers <- 1:10**, R will assign the numbers 1 through 10 to the integer vector called *numbers*.

```
> 1 + 1
[1] 2
> x <- 1 + 1
> x + 1
[1] 3
> x * x
[1] 4
> numbers <- 1:10
> numbers
[1] 1 2 3 4 5 6 7 8 9 10
> numbers ^ 2
[1] 1 4 9 16 25 36 49 64 81 100
> numbers * x
[1] 2 4 6 8 10 12 14 16 18 20
> sqrt(numbers)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
[9] 3.000000 3.162278
```

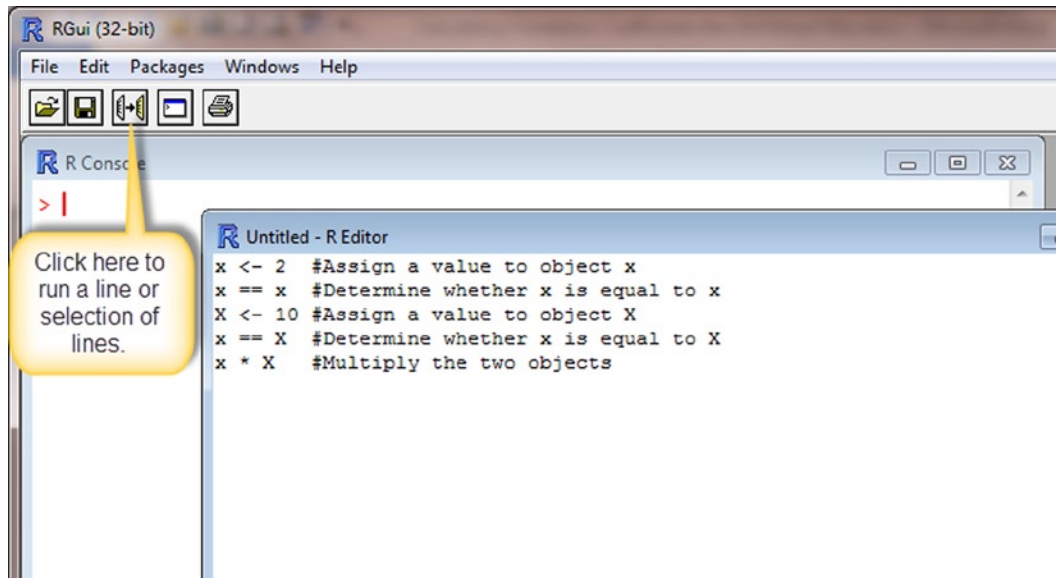
As mentioned at the beginning of this chapter, R is both functional and object-oriented. To R, everything is a function, including the basic mathematics operators. Everything is also an object in R. When you assign `x <- 1 + 1`, you have created an object called `x`. One of the most useful and powerful features of R is that many of its operators and functions are *vectorized*.

In computer science, something is vectorized if the program works on the vector in elementwise fashion, performing the same operation on each element of the vector that it would have performed on a scalar until it reaches the end of the vector. The general category of array-programming languages includes languages that generalize operations on scalars transparently to vectors, matrices, and higher-order arrays. An operation that works on an entire array is called a *vectorized operation*. Most computer languages are not vectorized to the extent R is. This makes it easy in many situations to avoid explicit loops, which are very slow in comparison to a vectorized operation. If you work in a scientific or engineering setting, you are probably familiar with MATLAB and Octave. Along with R and Python using the NumPy extension, these languages support array programming.

The only other computer language I have worked with that has the same level of vectorization is the now defunct language APL. In most languages, you would have to write a loop to square the numbers from 1 to 10. But in R, you simply use the exponent operator (^) to square all the numbers at once. The primary advantage of this is that you can frequently avoid explicit loops, as mentioned earlier.

R is case sensitive. Note that `x` and `X` are different objects in R. Although R is case sensitive, it is insensitive to spaces. I write code that uses spaces and indentation simply to make it easier for me and others to understand, and I usually comment my code fairly liberally. You would be surprised how often you can be doing something that makes perfectly good sense at the time, but looks like total gibberish when you return to it a few months later. Comments help. To insert a comment in a line of R code, simply enter `#`. The interpreter ignores anything after the `#` (pound sign or hash tag).

Here's a demonstration of the case sensitivity of R and the use of comments. Instead of working directly in the R GUI, click **File ► New Script** to open the R Editor. It is far easier to write and correct multiple lines of code in the editor (or in some other text editor) and execute the code from there than to type directly into the R Console. When you work in the R Editor, leave out the `>` command prompt. R will supply it (see Figure 1-5).



**Figure 1-5.** Use the R Editor to write multiple lines of R code

To execute your code, select one or more lines of code from the R Editor, and then click the icon for running the code in the R Console. As a shortcut, if you want to run all the code, use **Ctrl+A** to select all the code, and then press **Ctrl+R** to run the code in the R Console. Here is what you get:

```
> x <- 2 #Assign a value to object x
> x == x #Determine whether x is equal to x
[1] TRUE
> X <- 10 #Assign a value to object X
> x == X #Determine whether x is equal to X
[1] FALSE
> x * X #Multiply the two objects
[1] 20
>
```

Table 1-1 presents some useful operators, functions, and constants in R.

**Table 1-1.** *Useful Operator, Functions, and Constants in R*

| Operation/Function      | R Operator | Code Example   |
|-------------------------|------------|--|
| Addition                | +          | 1 + 1  |
| Subtraction             | -          | 2 - 1  |
| Multiplication          | *          | 3 * 2  |
| Division                | /          | 3 / 2  |
| Exponentiation          | ^          | 3 ^ 2  |
| Square root             | sqrt()     | Sqrt(81)   |
| Natural logarithm       | log()      | > exp(1)<br>[1] 2.718282<br>> log(exp(1))<br>[1] 1         |
| Common logarithm        | log10()    | > log10(100)<br>[1] 2                                      |
| Complex numbers         | complex()  | > z <- complex(real = 2, imaginary = 3)<br>> z<br>[1] 2+3i |
| Pi                      | pi         | > pi<br>[1] 3.141593                                       |
| Euler's number <i>e</i> | exp(1)     | > exp(1)<br>[1] 2.718282                                   |

Table 1-2 shows R's comparison operators. They evaluate to a logical value of TRUE or FALSE.

**Table 1-2.** *R Comparison Operators*

| Operator | Description              | Code Example     | Result/Comment |
|----------|--------------------------|------------------|----------------|
| >        | Greater than             | 3 > 2<br>2 > 3   | TRUE<br>FALSE  |
| <        | Less than                | 2 < 3<br>3 < 2   | TRUE<br>FALSE  |
| >=       | Greater than or equal to | 2 >= 2<br>2 >= 3 | TRUE<br>FALSE  |
| <=       | Less than or equal to    | 2 <= 2<br>3 <= 2 | TRUE<br>FALSE  |
| ==       | Equal to                 | 2 == 2<br>2 == 3 | TRUE<br>FALSE  |
| !=       | Not equal to             | 2 != 3<br>2 != 2 | TRUE<br>FALSE  |

Table 1-3 shows R's logical operators.

**Table 1-3.** *Logical Operators in R*

| Operator | Description | Code Example   | Result/Comment   |
|----------|-------------|--|--|
| &        | Logical And | <pre>&gt; x &lt;- 0:2 &gt; y &lt;- 2:0 &gt; (x &lt; 1) &amp; (y &gt; 1) [1] TRUE FALSE FALSE</pre> | This is the vectorized version. It compares two vectors element-wise and returns a vector of TRUE and/or FALSE.                              |
| &&       | Logical And | <pre>&gt; x &lt;- 0:2 &gt; y &lt;- 2:0 &gt; (x &lt; 1) &amp;&amp; (y &gt; 1) [1] TRUE</pre>        | This is the unvectorized version. It compares only the first value in each vector, left to right, and returns only the first logical result. |
|          | Logical Or  | <pre>&gt; (x &lt; 1)   (y &gt; 1) [1] TRUE FALSE FALSE</pre>                                       | This is the vectorized version. It compares two vectors element-wise and returns a vector of TRUE and/or FALSE.                              |
|          | Logical Or  | <pre>&gt; (x &lt; 1)    (y &gt; 1) [1] TRUE</pre>  | This is the unvectorized version. It compares two vectors and returns only the first logical result.   |
| !        | Logical Not | <pre>&gt; !y == x [1] TRUE FALSE TRUE</pre>  | Logical negation. Returns either a single logical value or a vector of TRUE and/or FALSE.  |

## Understanding the Data Types in R

As the preceding discussion has shown, R is strange in several ways. Remember R is both functional and object-oriented, so it has a bit of an identity crisis when it comes to dealing with data. Instead of the expected integer, floating point, array, and matrix types for expressing numerical values, R uses vectors for all these types of data. Beginning users of R are quickly lost in a swamp of objects, names, classes, and types. The best thing to do is to take the time to learn the various data types in R, and to learn how they are similar to, and often very different from, the ways you have worked with data using other languages or systems.

R has six “atomic” vector types, including *logical*, *integer*, *real*, *complex*, *string* (or character) and *raw*. Another data type in R is the list. Vectors must contain only one type of data, but lists can contain any combination of data types. A data frame is a special kind of list and the most common data object for statistical analysis. Like any list, a data frame can contain both numerical and character information. Some character information can be used for factors, and when that is the case, the data type becomes numeric. Working with factors can be a bit tricky because they are “like” vectors to some extent, but are not exactly vectors. My friends who are programmers think factors are “evil,” while statisticians like me love the fact that verbal labels can be used as factors in R, because such factors are self-labelling. It makes infinitely more sense to have a column in a data frame labelled sex with two entries, male and female, than it does to have a column labelled sex with 0s and 1s in the data frame.

In addition to vectors, lists, and data frames, R has language objects including *calls*, *expressions*, and *names*. There are *symbol objects* and *function objects*, as well as *expression objects*. There is also a special object called NULL, which is used to indicate that an object is absent. Missing data in R are indicated by NA.

We next discuss handling missing data. Then we will touch very briefly on vectors and matrices in R.

## Handling Missing Data in R

Create a simple vector using the `c()` function (some people say it means *combine*, while others say it means *concatenate*). I prefer “combine” because there is also a `cat()` function for concatenating output. For now, just type in the following and observe the results. The `na.rm = TRUE` option does not remove the missing value, but simply omits it from the calculations.

```
> x <- c(10, NA, 10, 25, 30, 15, 10, 18, 16, 15)
> x
[1] 10 NA 10 25 30 15 10 18 16 15
> mean(x)
[1] NA
> mean(x, na.rm = TRUE)
[1] 16.55556
>
```

## Working with Vectors in R

As you have learned, R treats a single number as a vector of length 1. If you create a vector of two or more objects, the vector must contain only a single data type. If you try to make a vector with multiple data types, R will coerce the vector into a single type. Chapter 3 covers how to deal with various data structure in more detail. For now, the goal is simply to show how R works with vectors.

Because you know how to use the R Editor and the R Console now, we will dispense with those formalities and just show the code and the output together. First, we will make a vector of 10 numbers, and then add a character element to the vector. R coerces the data to a character vector because we added a character object to it. I used the index `[11]` to add another element to the vector. But the vector now does not contain numbers and you cannot do math on it. Use a negative index, `[-11]`, to remove the character and the R function `as.integer()` to change the vector back to integers:

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> typeof(x)
[1] "integer"
> x[11] <- "happy"
> x
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9"
[10] "10" "happy"
> typeof(x)
[1] "character"
> x <- x[-11]
> x
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
> x <- as.integer(x)
> x
[1] 1 2 3 4 5 6 7 8 9 10
> typeof(x)
[1] "integer"
>
```



To make the example a little more interesting, let us work with some real data. The following data (thanks to Nat Goodman for the data) represent the ages in weeks of 20 randomly sampled mice from a much larger dataset.

```
> ages
[1] 10.5714 13.2857 13.5714 16.0000 10.2857 19.5714 20.0000  7.7143 20.5714
[10] 19.2857 14.0000 14.4286 19.7143 18.0000 13.2857 17.2857  5.2857 16.2857
[19] 14.1429  6.0000
> mean(ages)
[1] 14.46428
> typeof(ages)
[1] "double"
> mode(ages)
[1] "numeric"
> class(ages)
[1] "numeric"
```

R stores numeric values that are not integers in double-precision form. We can access individual elements of a vector with the index or indexes of those elements. Remember that most R functions and operators are vectorized, so that you can calculate the ages of the mice in months by dividing each age by 4. It takes only one line of code (shown in **bold**), and looping is not necessary.

```
> ages[1]
[1] 10.5714
> ages[20]
[1] 6
> ages[3:9]
[1] 13.5714 16.0000 10.2857 19.5714 20.0000  7.7143 20.5714
> months <- ages/4
> months
[1] 2.642850 3.321425 3.392850 4.000000 2.571425 4.892850 5.000000 1.928575
[9] 5.142850 4.821425 3.500000 3.607150 4.928575 4.500000 3.321425 4.321425
[17] 1.321425 4.071425 3.535725 1.500000
```

When you perform operations with vectors of different lengths, R will repeat the values of the shorter vector to match the length of the longer one. This “recycling” is sometimes very helpful as in multiplication by a scalar (vector of length 1), but sometimes produces unexpected results. If the length of the longer vector is a multiple of the shorter vector, this works well. If not, you get strange results like the following:

```
> x <- 1:2
> y <- 1:10
> z <- 1:3
> y/x
[1] 1 1 3 2 5 3 7 4 9 5
> y/z
[1] 1.0 1.0 1.0 4.0 2.5 2.0 7.0 4.0 3.0 10.0
Warning message:
In y/z : longer object length is not a multiple of shorter object length
```

## Working with Matrices in R

In another peculiarity of R, a matrix is also a vector, but a vector is not a matrix. I know this sounds like doublespeak, but read on for further explanation. A matrix is a vector with *dimensions*. You can make a vector into a one-dimensional matrix if you need to do so. Matrix operations are a snap in R. In this book, we work with two-dimensional matrices only, but higher-order matrices are possible, too.

We can create a matrix from a vector of numbers. Start with a vector of 50 random standard normal deviates (z scores if you like). R fills the matrix columnwise.

```
> zscores <- rnorm(50)
> zscores
[1] -1.19615960  0.95960082  0.50725210 -0.37411224  1.42044733  1.69437460
[7]  0.51677914 -0.04810441 -1.28024577 -0.48968148  1.28769546  0.93050145
[13]  0.72614070 -0.19306114 -0.56122938  0.77504861 -0.26756380 -1.11077206
[19] -0.60040090 -0.31920172  1.16802977  1.69736349  0.93134640 -1.15182325
[25]  0.12167256 -1.16038178  1.00415819  0.54469494  1.60231699 -0.11057038
[31]  0.01264523  0.57436245  0.54283138 -0.53045053  0.18115294  1.16062792
[37]  0.63649217  0.59524893 -0.52972220  0.45013366  0.31892391 -0.32371074
[43]  0.89716628 -0.15187155  0.25808226  1.73149549  1.36917698 -0.05803692
[49]  0.44942046  1.07708172

> zmatrix <- matrix(zscores, nrow = 10, ncol = 5)
> zmatrix
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] -1.19615960  1.2876955  1.1680298  0.01264523  0.31892391
[2,]  0.95960082  0.9305014  1.6973635  0.57436245 -0.32371074
[3,]  0.50725210  0.7261407  0.9313464  0.54283138  0.89716628
[4,] -0.37411224 -0.1930611 -1.1518232 -0.53045053 -0.15187155
[5,]  1.42044733 -0.5612294  0.1216726  0.18115294  0.25808226
[6,]  1.69437460  0.7750486 -1.1603818  1.16062792  1.73149549
[7,]  0.51677914 -0.2675638  1.0041582  0.63649217  1.36917698
[8,] -0.04810441 -1.1107721  0.5446949  0.59524893 -0.05803692
[9,] -1.28024577 -0.6004009  1.6023170 -0.52972220  0.44942046
[10,] -0.48968148 -0.3192017 -0.1105704  0.45013366  1.07708172
>
```

Imagine the five columns are students' standard scores on four quizzes and a final exam. You can specify names for the rows and columns of the matrix as follows:

```
> rownames(zmatrix)<-c("Jill", "Nat", "Jane", "Tim", "Larry", "Harry", "Barry", "Mary", "Gary", "Eric")
> zmatrix
      [,1]      [,2]      [,3]      [,4]      [,5]
Jill -1.19615960  1.2876955  1.1680298  0.01264523  0.31892391
Nat  0.95960082  0.9305014  1.6973635  0.57436245 -0.32371074
Jane  0.50725210  0.7261407  0.9313464  0.54283138  0.89716628
Tim -0.37411224 -0.1930611 -1.1518232 -0.53045053 -0.15187155
Larry 1.42044733 -0.5612294  0.1216726  0.18115294  0.25808226
Harry 1.69437460  0.7750486 -1.1603818  1.16062792  1.73149549
Barry 0.51677914 -0.2675638  1.0041582  0.63649217  1.36917698
Mary -0.04810441 -1.1107721  0.5446949  0.59524893 -0.05803692
Gary -1.28024577 -0.6004009  1.6023170 -0.52972220  0.44942046
Eric -0.48968148 -0.3192017 -0.1105704  0.45013366  1.07708172
```

```
> colnames(zmatrix) <- c("quiz1", "quiz2", "quiz3", "quiz4", "final")
> zmatrix
```

|       | quiz1       | quiz2      | quiz3      | quiz4       | final       |
|-------|-------------|------------|------------|-------------|-------------|
| Jill  | -1.19615960 | 1.2876955  | 1.1680298  | 0.01264523  | 0.31892391  |
| Nat   | 0.95960082  | 0.9305014  | 1.6973635  | 0.57436245  | -0.32371074 |
| Jane  | 0.50725210  | 0.7261407  | 0.9313464  | 0.54283138  | 0.89716628  |
| Tim   | -0.37411224 | -0.1930611 | -1.1518232 | -0.53045053 | -0.15187155 |
| Larry | 1.42044733  | -0.5612294 | 0.1216726  | 0.18115294  | 0.25808226  |
| Harry | 1.69437460  | 0.7750486  | -1.1603818 | 1.16062792  | 1.73149549  |
| Barry | 0.51677914  | -0.2675638 | 1.0041582  | 0.63649217  | 1.36917698  |
| Mary  | -0.04810441 | -1.1107721 | 0.5446949  | 0.59524893  | -0.05803692 |
| Gary  | -1.28024577 | -0.6004009 | 1.6023170  | -0.52972220 | 0.44942046  |
| Eric  | -0.48968148 | -0.3192017 | -0.1105704 | 0.45013366  | 1.07708172  |

Standardized scores are usually reported to two decimal places. Remove some of the extra decimals to make the next part of the code a little less cluttered. Set the number of decimals by using the `round()` function:

```
zmatrix <- round(zmatrix, digits = 2)
zmatrix
```

|       | quiz1 | quiz2 | quiz3 | quiz4 | final |
|-------|-------|-------|-------|-------|-------|
| Jill  | -1.20 | 1.29  | 1.17  | 0.01  | 0.32  |
| Nat   | 0.96  | 0.293 | 1.70  | 0.57  | -0.32 |
| Jane  | 0.51  | 0.73  | 0.93  | 0.54  | 0.90  |
| Tim   | -0.37 | -0.19 | -1.15 | -0.53 | -0.15 |
| Larry | 1.42  | -0.56 | 0.12  | 0.18  | 0.26  |
| Harry | 1.69  | 0.78  | -1.16 | 1.16  | 1.73  |
| Barry | 0.52  | -0.27 | 1.00  | 0.64  | 1.37  |
| Mary  | -0.05 | -1.11 | 0.54  | 0.60  | -0.06 |
| Gary  | -1.28 | -0.60 | 1.60  | -0.53 | 0.45  |
| Eric  | -0.49 | -0.32 | -0.11 | 0.45  | 1.08  |

If you have occasion to fill a matrix rowwise, set the `byrow` argument to `T` or `TRUE`. You can do this as follows.

```
> y <- matrix(x, nrow = 10, ncol = 10, byrow = TRUE)
> y
```

|       | [,1] | [,2] | [,3] | [,4] | [,5] | [,6] | [,7] | [,8] | [,9] | [,10] |
|-------|------|------|------|------|------|------|------|------|------|-------|
| [1,]  | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10    |
| [2,]  | 11   | 12   | 13   | 14   | 15   | 16   | 17   | 18   | 19   | 20    |
| [3,]  | 21   | 22   | 23   | 24   | 25   | 26   | 27   | 28   | 29   | 30    |
| [4,]  | 31   | 32   | 33   | 34   | 35   | 36   | 37   | 38   | 39   | 40    |
| [5,]  | 41   | 42   | 43   | 44   | 45   | 46   | 47   | 48   | 49   | 50    |
| [6,]  | 51   | 52   | 53   | 54   | 55   | 56   | 57   | 58   | 59   | 60    |
| [7,]  | 61   | 62   | 63   | 64   | 65   | 66   | 67   | 68   | 69   | 70    |
| [8,]  | 71   | 72   | 73   | 74   | 75   | 76   | 77   | 78   | 79   | 80    |
| [9,]  | 81   | 82   | 83   | 84   | 85   | 86   | 87   | 88   | 89   | 90    |
| [10,] | 91   | 92   | 93   | 94   | 95   | 96   | 97   | 98   | 99   | 100   |

R uses two indexes for the elements of a two-dimensional matrix. As with vectors, the indexes must be enclosed in square brackets. A range of values can be specified by use of the colon operator, as in `[1:2]`. You can also use a comma to indicate a whole row or a whole column of a matrix. Consider the following examples.

```
> y[,1:5]
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]   11   12   13   14   15
[3,]   21   22   23   24   25
[4,]   31   32   33   34   35
[5,]   41   42   43   44   45
[6,]   51   52   53   54   55
[7,]   61   62   63   64   65
[8,]   71   72   73   74   75
[9,]   81   82   83   84   85
[10,]  91   92   93   94   95
> y[1:5,]
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9   10
[2,]   11   12   13   14   15   16   17   18   19   20
[3,]   21   22   23   24   25   26   27   28   29   30
[4,]   31   32   33   34   35   36   37   38   39   40
[5,]   41   42   43   44   45   46   47   48   49   50
> y[5,5]
[1] 45
> y[10,10]
```

[1] 100R can do many useful things with matrices. For example, calculate the variance-covariance matrix by using the `var()` function:

```
> varcovar <- var(zmatrix)
> varcovar
      quiz1      quiz2      quiz3      quiz4      final
quiz1  1.0544544  0.11489111 -0.3285267  0.3838900  0.19691333
quiz2  0.1148911  0.63790667  0.1006644  0.1074422  0.07846222
quiz3 -0.3285267  0.10066444  1.0574489 -0.0665400 -0.19844667
quiz4  0.3838900  0.10744222 -0.0665400  0.2859656  0.20044222
final  0.1969133  0.07846222 -0.1984467  0.2004422  0.47039556
```

Invert a matrix by using the `solve()` function:

```
> inverse <- solve(varcovar)
> inverse
      quiz1      quiz2      quiz3      quiz4      final
quiz1  2.23763294 -0.02683957  0.6305501 -3.3937313  0.7799048
quiz2 -0.02683957  1.72182793 -0.2326712 -0.5743348 -0.1293917
quiz3  0.63055010 -0.23267125  1.2358101 -0.9683403  0.7088308
quiz4 -3.39373126 -0.57433478 -0.9683403  10.3613632 -3.3071827
final  0.77990476 -0.12939168  0.7088308 -3.3071827  3.5292487
```

Do matrix multiplication by using the `%%` operator. Just to make things clear, the matrix product of a matrix and its inverse is an identity matrix with 1's on the diagonal and 0's in the off-diagonals. Showing the result with fewer decimals makes this more obvious. For some reason, many of my otherwise very bright students do not “get” scientific notation at all.

```
> identity <- varcovar %*% inverse
> identity
```

|       | quiz1         | quiz2         | quiz3         | quiz4        | final         |
|-------|---------------|---------------|---------------|--------------|---------------|
| quiz1 | 1.000000e+00  | 5.038152e-18  | 3.282422e-17  | 2.602627e-16 | 5.529431e-18  |
| quiz2 | -8.009544e-18 | 1.000000e+00  | -2.323920e-17 | 1.080679e-16 | -4.710858e-17 |
| quiz3 | -7.697835e-17 | 7.521991e-17  | 1.000000e+00  | 9.513874e-17 | -9.215718e-17 |
| quiz4 | 1.076477e-16  | 1.993407e-17  | 3.182133e-17  | 1.000000e+00 | -4.325967e-17 |
| final | -4.770490e-18 | -6.986328e-18 | -1.832302e-17 | 1.560167e-16 | 1.000000e+00  |

```
> identity <- round(identity, 2)
> identity
```

|       | quiz1 | quiz2 | quiz3 | quiz4 | final |
|-------|-------|-------|-------|-------|-------|
| quiz1 | 1     | 0     | 0     | 0     | 0     |
| quiz2 | 0     | 1     | 0     | 0     | 0     |
| quiz3 | 0     | 0     | 1     | 0     | 0     |
| quiz4 | 0     | 0     | 0     | 1     | 0     |
| final | 0     | 0     | 0     | 0     | 1     |

## Looking Backward and Forward

In Chapter 1, you learned three important things: how to get R, how to use R, and how to work with missing data and various types of data in R. These are foundational skills. In Chapter 2, you will learn more about input and output in R. Chapter 3 will fill in the gaps concerning various data structures, returning to vectors and matrices, as well as learning how to work with lists and data frames.



# Input and Output

R provides many input and output capabilities. This chapter contains recipes on how to read data into R, as well as how to use several handy input and output functions. Although most R users are more concerned with input, there are times when you need to write to a file. You will find recipes for that in this chapter as well.

Oracle boasts that Java is everywhere, and that is certainly true, as Java is in everything from automobiles to cell phones and computers. R is not everywhere, but it is everywhere you need it to be for data analysis and statistics.

## Recipe 2-1. Inputting and Outputting Data

### Problem

To work with data, you need to get it into your R program. You may want to obtain that data from user input or from a file. Once you have done some processing you may want to output some data.

### Solution

Besides typing data into the console, you can use the script editor. The output for your R session appears in the R Console or the R Graphics Device. The basic commands for reading data from a file are `read.table()` and `read.csv()`.

---

■ **Note** Here CSV refers to comma-separated values.

---

You can write to a file using `write.table()`. In addition to these standard ways to get data into and out of R, there are some other helpful tools as well. You can use data frames, which are a special kind of list. As with any list, you can have multiple data types, and for statistical applications, the data frame is the most common data structure in R. You can get data and scripts from the Internet, and you can write functions that query users for keyboard input.

Before we discuss these I/O (input/output) options, let's see how you can get information regarding files and directories in R. File and directory information can be very helpful. The functions `getwd()` and `setwd()` are used to identify the current working directory and to change the working directory. For files in your working directory, simply use the file name. For files in a different directory, you must give the path to the file in addition to the name.

The function `file.info()` provides details of a particular file. If you need to know whether a particular file is present in a directory, use `file.exists()`. Using the function `objects()` or `ls()` will show all the objects in your workspace. Type **`dir()`** for a list of all the files in the current directory. Finally, you can see a complete list of file- and directory-related functions by entering the command `?files`.

To organize the discussion, I'll cover keyboard and monitor I/O; reading, cleaning, and writing data files; reading and writing text files; and R connections, in that order.

## Keyboard and Monitor Access

You can use the `scan()` function to read in a vector from a file or the keyboard. If you would rather enter the elements of a vector one at a time with a new line for input, just type **`x <- scan()`** and press the **Enter** key. R gives you the index, and you supply the value. See the following example. When you are finished entering data, just hit the **Enter** key with an empty index.

```
> xvector <- scan()
1: 19
2: 20
3: 31
4: 25
5: 36
6: 43
7: 53
8: 62
9: 40
10: 29
11:
Read 10 items
> xvector
[1] 19 20 31 25 36 43 53 62 40 29
```

Humans are better and faster at entering data in a column than they are at entering data in a row. You may like this way of entering vectors more than using the `c()` function.

If your data are in a file in the current working directory, you can enter a vector by using the file name as the argument for `scan()`. For example, assume you have a vector stored in a file called `yvector.txt`.

```
> scan("yvector.txt")
Read 10 items
[1] 22 18 32 39 42 73 37 55 34 34
```

The `readline()` function works in a similar fashion to get information from the keyboard. For example, you may have a code fragment like the following:

```
> yourName <- readline("Type in Your First and Last Name: ")
Type in Your First and Last Name: Larry Pace
> yourName
[1] "Larry Pace"
```

In the interactive mode, you can print the value of an object to the screen simply by typing the name of the object and pressing **Enter**. You can also use the `print()` function, but it is not necessary at the top level of the interactive session. However, if you want to write a function that prints to the console, just typing the name of the object will no longer work. In that case, you will have to use the `print()` function. Examine the following code. I wrote the function in the script editor to make things a little easier to control. I cover writing R functions in more depth in Chapter 11.

```
> cubes
function(x) {
  print(x^3)
}
```

```
> x <- 1:20
> cubes(x)
[1] 1 8 27 64 125 216 343 512 729 1000 1331 1728 2197 2744 3375
[16] 4096 4913 5832 6859 8000
```

## Reading and Writing Data Files

R can deal with data files of various types. Tab-delimited and CSV are two of the most common file types. If you load the `foreign` package, you can read in additional data types, such as SPSS and SAS files.

### Reading Data Files

To illustrate, I will get some data in SPSS format from the General Social Survey (GSS) and then open it in R. The GSS dataset is used by researchers in business, economics, marketing, sociology, political science, and psychology. The most recent GSS data are from 2012. You can download the data from [www3.norc.org/GSS+Website/Download/](http://www3.norc.umd.edu/GSS+Website/Download/) in either SPSS format or Stata format.

Because Stata does a better job than SPSS at coding the missing data in the GSS dataset, I saved the Stata (\*.DTA) format into my directory and then opened the dataset in SPSS. This fixed the problem of dealing with missing data, but my data are far from ready for analysis yet. If you do not have SPSS, you can download the open-source program PSPP, which can read and write SPSS files, and can do most of the analyses available in SPSS. The point of this illustration is simply that there are data out there in cyberspace that you can import into R, but you may often have to make a pit stop at SPSS, Stata, PSPP, Excel, or some other program before the data are ready for R. If you have an “orderly” SPSS dataset with variable names that are legal in R, you can open that file directly into R with no difficulty using the `foreign()` package.

When I read the SPSS data file into R, I see I still have some work to do:

```
> require(foreign)
Loading required package: foreign
> gss2012 <- read.spss("GSS2012.sav")
There were 11 warnings (use warnings() to see them)
> warnings()
Warning messages:
1: In read.spss("GSS2012.sav") :
  GSS2012.sav: Unrecognized record type 7, subtype 18 encountered in system file
2: In `levels<-(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
3: In `levels<-(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
4: In `levels<-(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
5: In `levels<-(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
6: In `levels<-(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
7: In `levels<-(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
8: In `levels<-(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
```



```

9: In `levels<-`(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
10: In `levels<-`(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated
11: In `levels<-`(`*tmp*`, value = if (nl == nl) as.character(labels) else paste0(labels, ... :
  duplicated levels in factors are deprecated

```

Although this dataset with 4820 records and 1067 variables is large by the standards of the majority of researchers, the data are not “big” in the modern sense. As you can see by the preceding warning messages, the next problem is that the data must be cleaned up a bit before I can do any serious data analysis. Dealing with dirty data is a real-world problem that is not sufficiently addressed in most statistics textbooks, in which professors like me make up examples that are easy to work with, and which almost never have missing data. Recipe 2-2 deals with cleaning up data.

---

■ **Note** R nearly choked on the GSS data. We will talk about how to handle very large datasets in Chapter 13.

---

## Writing Data Files

The `write.table()` function is the analog of the `read.table()` function. The `write.table()` function writes a data frame. The function `cat()` can also be used to write to a file (or to the screen), by successive parts. What this means is that you concatenate the arguments to the `cat()` function, separating them by commas. You can use any R data type for this purpose. The following code illustrates this:

```

> cats <- c("Tom","Felix","Mittens","Socks","Boots","Fluffy")
> ages <- c(12,10,8,2,5,3)
> pets <- data.frame(cats, ages, stringsAsFactors = FALSE)
> pets
   cats ages stringsAsFactors
1   Tom  12             FALSE
2  Felix  10             FALSE
3 Mittens   8             FALSE
4  Socks   2             FALSE
5  Boots   5             FALSE
6 Fluffy   3             FALSE
> write.table(pets, "myCats")
> cat("Tom\n", file = "catFile")
> cat("Felix\n", file = "catFile", append = TRUE)
> ## verify the file writes by using the file.exists() function
> file.exists("myCats")
[1] TRUE
> file.exists("catFile")
[1] TRUE

```

## Recipe 2-2. Cleaning Up Data

### Problem

Real-world data often need cleaning. For example, the GSS codebook uses several different codes for missing data. The easiest way to handle the recoding in this particular case is to clean the dataset in SPSS (see Recipe 2-1 for more on GSS). After the cleaning, the data will be more orderly. In many cases, cleaning data in R is more efficient, and in many others, it might be more efficient to use the search-and-replace functionality of a word processor or a spreadsheet program. As always, choose the most appropriate tool from the toolbox. If the dataset is small, you can make minor edits using the R Data Editor, not to be confused with the script editor.

### Solution

When you have serious data recoding and cleaning to do (I call it “data surgery”), I suggest you make use of the `plyr` package in R. Think of a pair of pliers. The `plyr` package is a SAC (split-apply-combine) tool, and does a great job for such purposes.

To illustrate some real-world data cleaning issues, let us use a manageable (and I hope interesting to you) set of data, compliments of Dr. Nat Goodman. The data consist of various measurements of mutant and normal mice. The mutated mice were created to carry the genome sequence for Huntington’s disease. Several different strains of mice were used because inbred mice are as alike genetically as human twins are. For this example, we will work with only two strains of mice.

The following is the head (the first few records) of the mouse data (which we can view with the `head()` function). Each mouse has a unique identifier, the strain, the nominal genome sequence, and the actual genome sequence. The sequence CAG repeated seven times represents a normal mouse. CAG sequences of 40 or more are associated with Huntington’s disease in mice. The other variables are self-descriptive. The age is the mouse’s age in weeks. This dataset represents a small portion of a much larger dataset.

As you have seen previously, when you read in CSV files, you do not have to specify that the first row contains the variable names. The “header” is expected in a CSV file. However, many tab-delimited files do not have a row of column headers. If your tab-delimited file does have a row of variable names as the first row, you must set the header option to T or TRUE, as shown in the following code segment.

```
> mouseWeights <- read.table("Mouse_Weights.txt", header = TRUE)
> head(mouseWeights)
  mouse_id strain cag_nominal cag_actual sex   age body_weight brain_weight
1  hd1769   B6      Q111      113    F 3.7143    11.18      0.380
2  hd1777   B6      Q111      137    F 4.0000    12.50      0.434
3  hd1778   B6       WT        7    F 4.0000    13.30      0.406
4  hd1782   B6      Q111      136    F 4.0000    11.66      0.426
5  hd1806   B6       WT        7    M 4.0000    14.33      0.464
6  hd1808   B6      Q111      113    M 4.0000    13.72      0.414
```

When we examine the data, we see that there are some problems. We find that some mice have an impossible body weight of zero grams. Other mice have an equally impossible brain weight of zero grams.

```
summary(mouseWeights)
  mouse_id   strain cag_nominal cag_actual sex   age
hd1094 :    1   B6 :376      Q111:172  Min.    : 7.00  F:315  Min.    : 3.714
hd1095 :    1  CD1:268      Q50 :166  1st Qu.: 7.00  M:329  1st Qu.: 8.000
hd1104 :    1           Q92 : 51    Median :48.00      Median :12.143
hd1107 :    1           WT  :255    Mean   :57.93      Mean   :12.138
hd1109 :    1           3rd Qu.:113.00      3rd Qu.:16.286
hd1110 :    1           Max.    :154.00      Max.    :20.571
```