THE EXPERT'S VOICE® IN PROGRAMMING

Beginning R

An Introduction to Statistical Programming

POWERING THROUGH STATISTICS WITH THE FREELY-AVAILABLE R LANGUAGE

Larry Pace

Apress®

Beginning R

An Introduction to Statistical Programming

Larry Pace

Apress®

Beginning R: An Introduction to Statistical Programming

ISBN 978-1-4302-4554-4

ISBN 978-1-4302-4555-1 (eBook)

President and Publisher: Paul Manning
Lead Editor: Jonathan Gennick
Technical Reviewer: Myron Hylnka
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Kevin Shea
Copy Editor: Jill Steinberg
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code.

To my wife Shirley Pace, who has taught me the true meaning of love and loyalty.

—Larry Pace

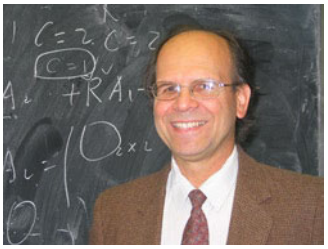
Contents at a Glance

Contents

About the Author

Larry Pace is a statistics author, educator, and consultant. He lives in the upstate area of South Carolina in the town of Anderson. He earned his Ph.D. from the University of Georgia in psychometrics (applied statistics) with a content major in industrial-organizational psychology. He has written more than 100 publications including books, articles, chapters, and book and test reviews. In addition to a 35-year academic career, Larry has worked in private industry as a personnel psychologist and organization effectiveness manager for Xerox Corporation, and as an organization development consultant for a private consulting firm. He has programmed in a variety of languages and scripting languages including FORTRAN-IV, BASIC, APL, C++, JavaScript, Visual Basic, PHP, and ASP. Larry has won numerous awards for teaching, research, and service. He is currently a Graduate Research Professor at Keiser University, where he teaches doctoral courses in statistics and research. He also teaches adjunct classes for Clemson University. When he is not reading and writing about statistics, helping others with their statistical analyses, teaching, and doing research, he likes to build spreadsheet models for various statistical analyses, tend a small vegetable garden, play his guitar, and cook on the grill. He is married to Shirley Pace, and the Paces have four grown children and two grandsons. The Paces are volunteers with Meals on Wheels and avid pet lovers with six cats and one dog, all rescued.

About the Technical Reviewer



Dr. Myron Hlynka is a professor in the Department of Mathematics and Statistics at the University of Windsor in Windsor, Ontario, Canada. He received his Ph.D. in Statistics from Penn State University in 1985. His research specialties are queueing theory and applied probability, and he maintains a well-known queueing theory web page at web2.uwindsor.ca/math/hlynka/queue.html.

Acknowledgments

No published book is ever the work of a single individual, even when only one author is listed. This book came to fruition very quickly, thanks to an excellent team and great teamwork. Jonathan Gennick of Apress approached me in the spring of 2012 with the idea of a book on beginning R. Realizing that the current R books miss the mark in significant ways, and that a book on R to help novices and intermediate programmers alike was much needed, we quickly came to an agreement. With Jonathan's expert guidance, the book went from concept to completion in only eight months. Jonathan assembled a team of professionals, and the synergy was amazing. I wish all my writing projects were as easy and fun as this one was, and as fast, too! I was privileged to have the expert technical review of Dr. Myron Hlynka of the University of Windsor, Canada. Myron caught more than a few gaffes, and made excellent suggestions for improvement in the text, almost all of which were followed. He also provided outstanding detailed reviews of the R code, making sure it worked as advertised, as well as many great ideas for improving the code and expanding the examples. The book is both better and more accurate because of Myron's eagle eye and his statistical and programming expertise. Jill Steinberg did the copyediting, and her suggestions and corrections made the book more readable, easier to understand, and more consistent. Kevin Shea, my contributing editor, kept the entire project on track, and kept me focused on both the needed details and the big picture at the same time. Thanks, Kevin! Mark Powers also pitched in from time to time, and kept the project moving along. I am grateful to the Apress production team for their excellent work in producing the final copy and the graphics for this book. I hope you enjoy reading it as much as I enjoyed writing it!

—Larry Pace

Introduction

This is a beginning to intermediate book on the statistical language and computing environment called R. As you will learn, R is freely available and open source. Thousands of contributed packages are available from members of the R community. In this book, you learn how to get R, install it, use it as a command-line interpreted language, program in it, write custom functions, use it for the most common descriptive and inferential statistics, and write an R package. You also learn some "newer" statistical techniques including bootstrapping and simulation, as well as how to use R graphical user interfaces (GUIs) including RStudio and RCommander.

Who This Book Is For

This book is for working professionals who need to learn R to perform statistical analyses. Additionally, statistics students and professors will find this book helpful as a textbook, a supplement for a statistical computing class, or a reference for various statistical analyses. Both statisticians who want to learn R and R programmers who need a refresher on statistics will benefit from the clear examples, the hands-on nature of the book, and the conversational style in which the book is written.

How This Book Is Structured

This book is structured in 20 chapters, each of which covers the use of R for a particular purpose. In the first three chapters, you learn how to get and install R and R packages, how to program in R, and how to write custom functions. The standard descriptive statistics and graphics are covered in Chapters 4 to 7. Chapters 8 to 14 cover the customary hypothesis tests concerning means, correlation and regression, and multiple regression. Chapter 14 introduces logistic regression. Chapter 15 covers chi-square tests. Following the standard nonparametric procedures in Chapter 16, Chapters 17 and 18 introduce simulation and the "new" statistics including bootstrapping and permutation tests. The final two chapters cover making an R package and using the RCommander package as a point-and-click statistics interface.

Conventions

In this book, we use `TheSansMonoConNormalfont` to show R code both inline and as code segments. The R code is typically shown as you would see it in the R Console or the R Editor. All hyperlinks shown in this book were active at the time of printing. Hyperlinks are shown in the following fashion:

`http://www.apress.com`

When you use the mouse to select from the menus in R or an R GUI, the instructions will appear as shown below. For example, you may be directed to install a package by using the Packages menu in the RGui. The instructions will state simply to select `Packages` ➤ `Install packages...` (the ellipsis points mean that an additional dialog box or window will open when you click `Install packages`). In the current example, you will see a list of mirror sites from which you can download and install R packages.

Downloading the code

The R code and documentation for the examples shown in this book and most of the datasets used in the book are available on the Apress web site, www.apress.com. You can find a link on the book's information page under the Source Code/Downloads tab. This tab is located below the Related Titles section of the page.

Contacting the Author

I love hearing from my readers, especially fellow statistics professors. Should you have any questions or comments, an idea for improvement, or something you think I should cover in a future book—or you spot a mistake you think I should know about—you can contact me at larry@twopaces.com.

CHAPTER 1

Getting R and Getting Started

R is a flexible and powerful open-source implementation of the language S (for *statistics*) developed by John Chambers and others at Bell Labs. R has eclipsed S and the commercially available S-Plus program for many reasons. R is free, and has a variety (nearly 4,000 at last count) of contributed packages, most of which are also free. R works on Macs, PCs, and Linux systems. In this book, you will see screens of R 2.15.1 running in a Windows 7 environment, but you will be able to use everything you learn with other systems, too. Although R is initially harder to learn and use than a spreadsheet or a dedicated statistics package, you will find R is a very effective statistics tool in its own right, and is well worth the effort to learn.

Here are five compelling reasons to learn and use R.

- R is open source and completely free. It is the *de facto* standard and preferred program of many professional statisticians and researchers in a variety of fields. R community members regularly contribute packages to increase R's functionality.
- R is as good as (often better than) commercially available statistical packages like SPSS, SAS, and Minitab.
- R has extensive statistical and graphing capabilities. R provides hundreds of built-in statistical functions as well as its own built-in programming language.
- R is used in teaching and performing computational statistics. It is the language of choice for many academics who teach computational statistics.
- Getting help from the R user community is easy. There are readily available online tutorials, data sets, and discussion forums about R.

R combines aspects of functional and object-oriented programming. One of the hallmarks of R is implicit looping, which yields compact, simple code and frequently leads to faster execution. R is more than a computing language. It is a software system. It is a command-line interpreted statistical computing environment, with its own built-in scripting language. Most users imply both the language and the computing environment when they say they are "using R." You can use R in *interactive* mode, which we will consider in this introductory text, and in *batch* mode, which can automate production jobs. We will not discuss the batch mode in this book. Because we are using an interpreted language rather than a compiled one, finding and fixing your mistakes is typically much easier in R than in many other languages.

Getting and Using R

The best way to learn R is to use it. The developmental process recommended by John Chambers and the R community, and a good one to follow, is *user* to *programmer* to *contributor*. You will begin that developmental process in this book, but becoming a proficient programmer or ultimately a serious contributor is a journey that may take years.

If you do not already have R running on your system, download the precompiled binary files for your operating system from the Comprehensive R Archive Network (CRAN) web site, or preferably, from a mirror site close to you. Here is the CRAN web site:

http://cran.r-project.org/

Download the binary files and follow the installation instructions, accepting all defaults. Launch R by clicking on the R icon. For other systems, open a terminal window and type "R" on the command line. When you launch R, you will get a screen that looks something like the following. You will see the label *R Console*, and this window will be in the *RGui* (graphical user interface). Examine Figure 1-1 to see the R Console.

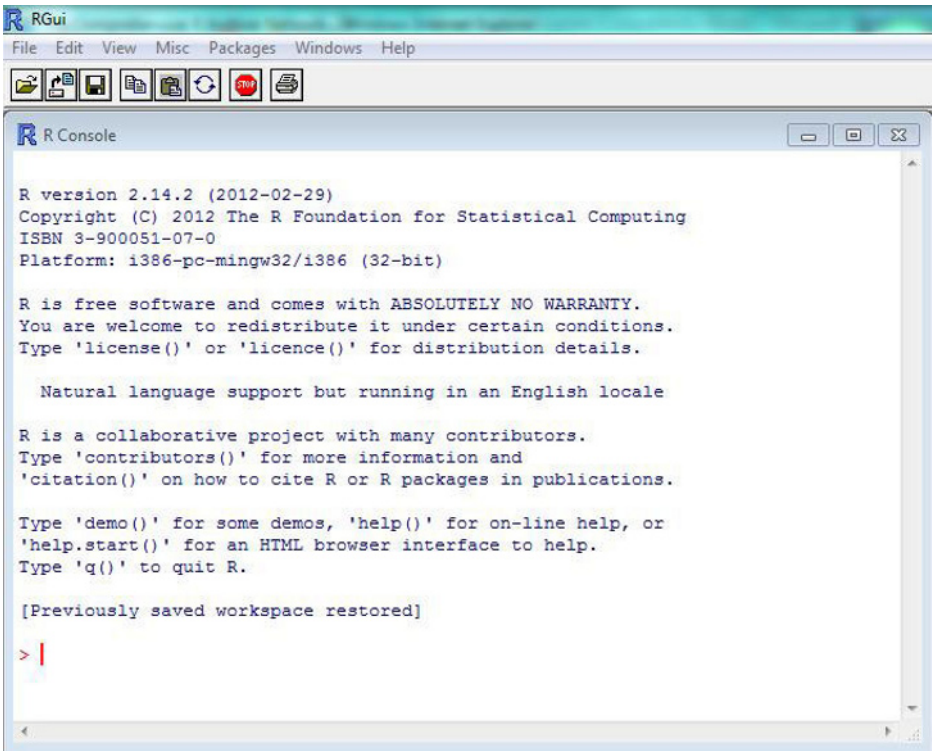


Figure 1-1. *The R Console running in the RGui in Windows 7*

Although the R greeting is helpful and informative for beginners, it also takes up a lot of screen space. You can clear the console by pressing `<Ctrl>+ L` or by selecting `Edit` ➤ `Clear console`. R's icon bar can be used to open a script, load a workspace, save the current workspace image, copy, paste, copy and paste together, halt the program (useful for scripts producing unwanted or unexpected results), and print. You can also gain access to these features using the menu bar.

■ **Tip** You can customize your R Profile file so that you can avoid the opening greeting altogether. See the R documentation for more information.

Many casual users begin typing expressions (*one-liners*, if you will) in the R console after the R prompt (`>`). This is fine for short commands, but quickly becomes inefficient for longer lines of code and scripts. To open

the R Editor, simply select `File > New script`. This opens a separate window into which you can type commands (see Figure 1-2). You can then execute one or more lines by selecting the code you want to use, and then pressing `<Ctrl>+ R` to run the code in the R Console. If you find yourself writing the same lines of code repeatedly, it is a good idea to save the script so that you can open it and run the lines you need without having to type the code again. You can also create custom functions in R. We will discuss the R interface, data structures, and R programming before we discuss creating custom functions.

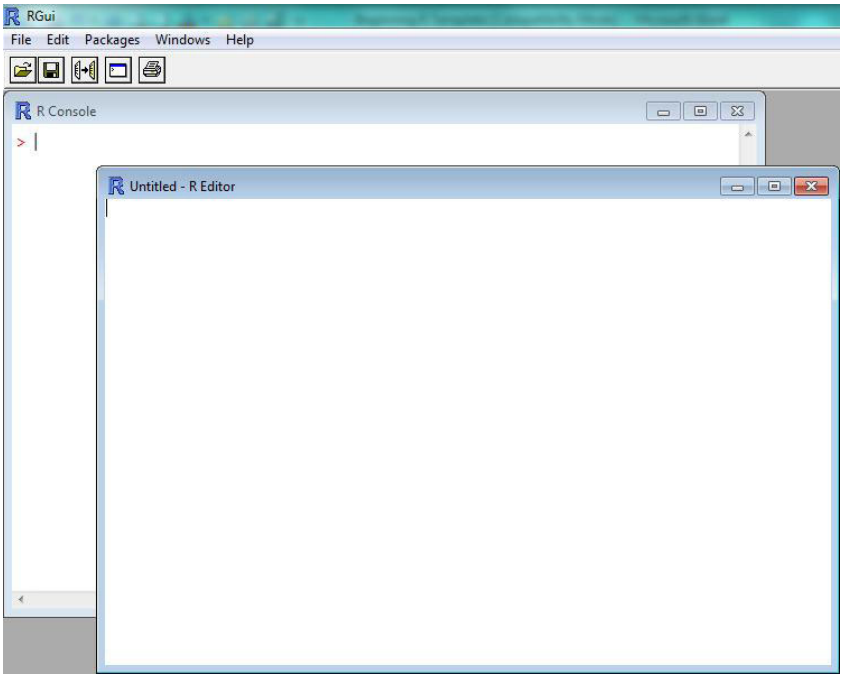


Figure 1-2. *The R Editor*

A First R Session

Now that you know about the R Console and R Editor, you will see their contents from this point forward simply shown in `this font`. Let us start with the use of R as a calculator, typing commands directly into the R Console. Launch R and type the following code, pressing `<Enter>` after each command. Technically, everything the user types is an *expression*.

```
> 2 ^ 2
[1] 4
> 2 * 2
[1] 4
> 2 / 2
[1] 1
> 2 + 2
[1] 4
> 2 - 2
[1] 0
> q()
```

Like many other programs, R uses ^ for exponentiation, * for multiplication, / for division, + for addition, and – for subtraction. R labels each output value with a number in square brackets. As far as R is concerned, there is no such thing as a *scalar quantity*; to R, an individual number is a one-element vector. The [1] is simply the index of the first element of the vector. To make things easier to understand, we will sometimes call a number like 2 a *scalar*, even though R considers it a vector.

The power of R is not in basic mathematical calculations (though it does them flawlessly), but in the ability to assign values to objects and use functions to manipulate or analyze those objects. R allows three different assignment operators, but we will use only the traditional <- for assignment.

You can use the equal sign = to assign a value to an object, but this does not always work and is easy to confuse with the test for equality, which is ==. You can also use a right-pointing assignment operator ->, but that is not something we will do in this book. When you assign an object in R, there is no need to declare or type it. Just assign and start using it. We can use x as the label for a single value, a vector, a matrix, a list, or a data frame.

We will discuss each data type in more detail, but for now, just open a new script window and type, and then execute the following code. We will assign several different objects to x, and check the *mode* (storage class) of each object. We create a single-element vector, a numeric vector, a matrix (which is actually a kind of vector to R), a character vector, a logical vector, and a list. The three main types or modes of data in R are *numeric*, *character*, and *logical*. Vectors must be *homogeneous* (use the same data type), but lists, matrices, and data frames can all be *heterogeneous*. I do not recommend the use of heterogeneous matrices, but lists and data frames are commonly composed of different data types. Here is our code and the results of its execution. Note that the code in the R Editor does not have the prompt > in front of each line.

```
x <- 2
x
x ^ x
x ^ 2
mode(x)
x <- c(1:10)
x
x ^ x
mode(x)
dim(x) <- c(2,5)
x
mode(x)
x <- c("Hello","world","!")
x
mode(x)
x <- c(TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE)
x
mode(x)
x <- list("R","12345",FALSE)
x
mode(x)
```

Now, see what happens when we execute the code:

```
> x <- 2
> x
[1] 2
> x ^ x
[1] 4
> x ^ 2
[1] 4
> mode(x)
[1] "numeric"
```

Note the "sequence operator" will produce the same result as `c(1:10)`. You could produce a vector with the numbers 1 through 10 by using `seq(1:10)`. R provides the user the flexibility to do the same thing in many different ways. Consider the following examples:

```
> seq(1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
> x <- c(1:10)
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x ^ x
 [1]          1          4         27        256       3125      46656
 [7]     823543   16777216  387420489 10000000000
> mode(x)
[1] "numeric"
> dim(x) <- c(2,5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> mode(x)
[1] "numeric"
```

Here is the obligatory "Hello World" code that is almost universally included in programming books and classes:

```
> x <- c("Hello","world","!")
> x
[1] "Hello" "world" "!"
> mode(x)
[1] "character"
> x <- c(TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE)
> x
[1]  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE
> mode(x)
[1] "logical"
> x <- list("R","12345",FALSE)
> x
[[1]]
[1] "R"

[[2]]
[1] "12345"

[[3]]
[1] FALSE

> mode(x)
[1] "list"
```

List indexing is quite different from vector and matrix indexing, as you can see from the output above. We will address that in more detail later. For now, let us discuss moving around in R.

Moving Around in R

R saves all the commands you type during an R session, even if you have cleared the console. To see the previous lines(s) of code, use the up arrow on the keyboard. To scroll down in the lines of code, use the down arrow. The left and right arrows move the cursor in the current line, and this allows you to edit and fix code if you made a mistake. As you have learned, you can clear the console when you want a clear working area, but you will still be able to retrieve the entire command history. When you are finished with a session, you may want to save the workspace image if you did something new and useful, or discard it if you were just experimenting or created something worse than what you started with! When you exit R, using the `q()` function, or in Windows, File > Exit, the R system will save the entire command history in your current (working) directory as a text file with the extension *.Rhistory.

You can access the R history using a text editor like Notepad. Examining the history is like viewing a recording of the entire session from the perspective of the R Console. This can help you refresh your memory, and you can also copy and paste sections of the history into the R Editor or R Console to speed up your computations.

As you have already seen, the R Editor gives you more control over what you are typing than the R Console does, and for anything other than one-liners, you will find yourself using the R Editor more than the R Console.

In addition to saving the command history, R saves the functions, loaded packages, and objects you create during an R session. The benefit of saving your workspace and scripts is that you will not have to type the information again when you access this workspace image.When you exit R, you will see a prompt asking if you want to save your workspace image (see Figure 1-3).

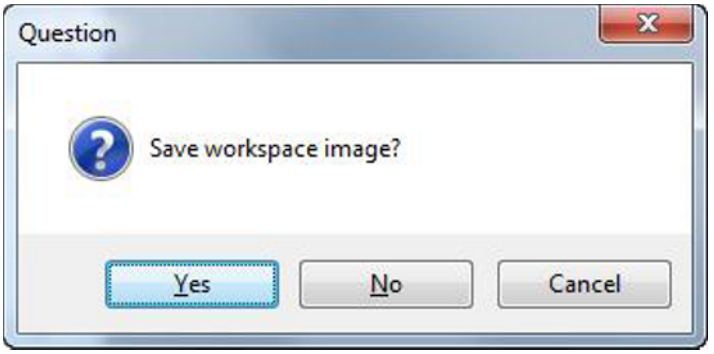


Figure 1-3. *When you exit R, you will receive this prompt*

Depending on what you did during that session, you may or may not want to do that. If you do, R will save the workspace in the *.RData format. Just as you can save scripts, output, and data with other statistics packages, you can save multiple workspace images in R. Then you can load and use the one you want. Simply find the desired workspace image, which will be an *.RData file, and click on it to launch R. The workspace image will contain all the data, functions, and loaded packages you used when you saved the workspace.

When you are in an R session, you may lose track of where you are in the computer's file system. To find out your working directory, use the `getwd()`command. You can also change the working directory by using the `setwd(dir)` command. If you need help with a given function, you can simply type `help(function)`, or use the `?function` shortcut. Either of these will open the R documentation, which is always a good place to start when you have questions.

To see a listing of the objects in your workspace, you can use the `ls()` function. To get more detail, use `ls.str()`. For example, here is the list of the objects in my current R workspace:

```
> ls()
 [1] "A"           "acctdata"     "address"      "B"            "b1"
 [6] "balance"     "c"            "CareerSat"    "chisquare"    "colnames"
[11] "confint"     "correctquant" "dataset"      "Dev"          "grades"
[16] "Group"       "hingedata"    "hours"        "i"            "Min_Wage"
```

```
[21] "n"           "names"       "newlist"     "P"           "pie_data"
[26] "quizzes"     "r"           "sorted"      "stats"       "stdev"
[31] "TAge"        "test1"       "test2"       "test3"       "testmeans"
[36] "tests"       "truequant"   "Tscore"      "Tscores"     "V"
[41] "x"           "y"           "z1"          "zAge"        "zscores"
[46] "ztest1"
```

Let us create a couple of objects and then see that they are added to the workspace (and will be saved when/if you save the workspace image).

```
> Example1 <- seq(2:50)
> Example2 <- log(Example1)
> Example2
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851 2.3978953 2.4849066 2.5649494 2.6390573
[15] 2.7080502 2.7725887 2.8332133 2.8903718 2.9444390 2.9957323 3.0445224
[22] 3.0910425 3.1354942 3.1780538 3.2188758 3.2580965 3.2958369 3.3322045
[29] 3.3672958 3.4011974 3.4339872 3.4657359 3.4965076 3.5263605 3.5553481
[36] 3.5835189 3.6109179 3.6375862 3.6635616 3.6888795 3.7135721 3.7376696
[43] 3.7612001 3.7841896 3.8066625 3.8286414 3.8501476 3.8712010 3.8918203
>
```

Now, see that `ls()` will show these two vectors as part of our workspace. (This is a different workspace image from the one shown earlier.) As this is a large workspace, I will show only the first screen (see Figure 1-4).

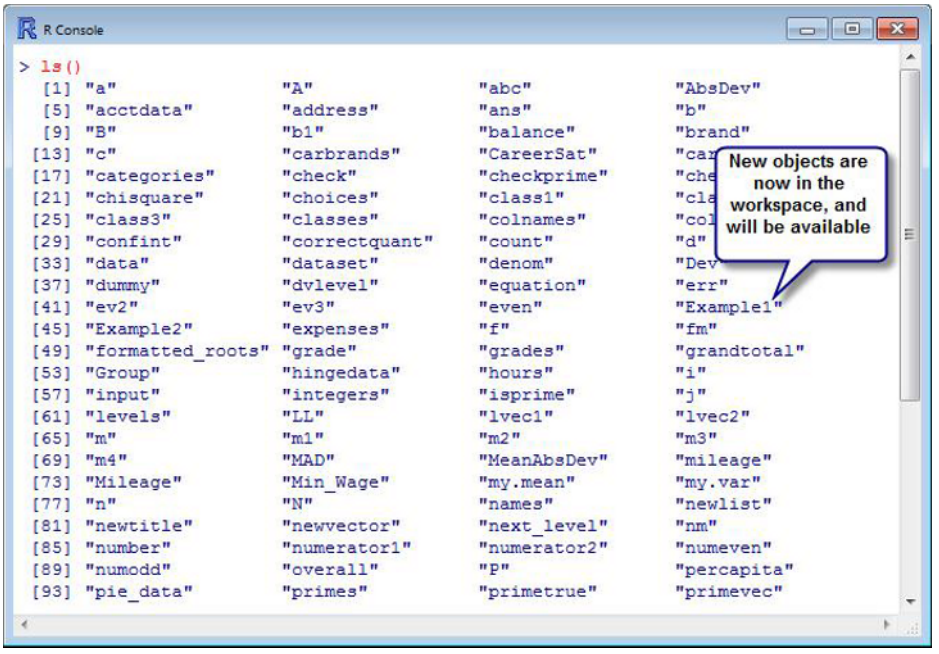


Figure 1-4. *Newly-created objects are automatically stored in the workspace*

Now, see what happens when we invoke the `ls.str()` function for a "verbose" description of the objects in the workspace. Both our examples are included. The `ls.str()` function gives an alphabetical listing of all the objects currently saved in the workspace. We will look at the descriptions of only the two examples we just created.

```
> ls.str()

ls.str()
Example1 :  int [1:49] 1 2 3 4 5 6 7 8 9 10 ...
Example2 :  num [1:49] 0 0.693 1.099 1.386 1.609 ...
```

Working with Data in R

As the creator of the S language, John Chambers, says, in most serious modern applications, real data usually comes from a process external to our analysis. Although we can enter small amounts of data directly in R, ultimately we will need to import larger data sets from applications such as spreadsheets, text files, or databases. Let us discuss and illustrate the various R data types in more detail, and see how to work with them most effectively.

Vectors

The most common data structure in R is the *vector*. As we have discussed, vectors must be *homogeneous*—that is, the type of data in a given vector must all be the same. Vectors can be numeric, logical, or character. If you try to mix data types, you will find that R forces (*coerces*, if you will) the data into one mode.

Creating a Vector

See what happens when you try to create a vector as follows. R obliges, but the mode is character, not numeric.

```
> x <- c(1, 2, 3, 4, "Pi")
> x
[1] "1"  "2"  "3"  "4"  "Pi"
> mode(x)
[1] "character"
```

Let us back up and learn how to create numeric vectors. When we need to mix data types, we need lists or data frames. Character vectors need all character elements, and numeric vectors need all numeric elements. We will work with both character and numeric vectors in this book, but let us work with numeric ones here. R has a *recycling* property that is sometimes quite useful, but which also sometimes produces unexpected or unwanted results. Let's go back to our sequence operator and make a numeric vector with the sequence 1 to 10. We assign vectors using the `c` (for *combine*) function. Though some books call this *concatenation*, there is a different `cat()` function for concatenating output. We will discuss the `cat()` function in the next chapter. Let us now understand how R deals with vectors that have different numbers of elements. For example, see what happens when we add a vector to a different single-element vector. Because we are adding a "scalar" (really a single-element vector), R's recycling property makes it easy to "vectorize" the addition of one value to each element of the other vector.

```
> x <- c(1:10)
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> length(x)
[1] 10
> y <- 10
> length(y)
[1] 1
> x+y
 [1] 11 12 13 14 15 16 17 18 19 20
```

Many R books and tutorials refer to the last command above as *scalar addition*, and that much is true. But what really matters here is that the command and its output are technically an implementation of R's *recycling* rule we discussed above. When one vector is shorter than the other, the shorter vector is recycled when you apply mathematical operations to the two vectors.

Performing Vector Arithmetic

R is very comfortable with adding two vectors, but notice what sometimes happens when they are of different lengths, and one is not a single-element vector. Look at the output from the following examples to see what happens when you add two vectors. In some cases, it works great, but in others, you get warned that the longer object is not a multiple of the length of the shorter object!

```
> y <- c(0, 1)
> y
[1] 0 1
> x+y
 [1]  1  3  3  5  5  7  7  9  9 11
> y <- c(1,3,5)
> x+y
 [1]  2  5  8  5  8 11  8 11 14 11
Warning message:
In x+y : longer object length is not a multiple of shorter object length
```

What is happening in the code above is that the shorter vector is being recycled as the operation continues. In the first example, zero is added to each odd number, while 1 is added to each even number. R recycles the elements in the shorter vector, as it needs, to make the operation work. Sometimes R's recycling feature is useful, but often it is not. If the vectors are *mismatched* (that is, if the length of the longer vector is not an exact multiple of the shorter vector's length), R will give you a warning, but will still recycle the shorter vector until there are enough elements to complete the operation.

Before we discuss vector arithmetic in more detail, let us look at a few more computations using our example vector x:

```
> 2+3 * x       #Note the order of operations
 [1]  5  8 11 14 17 20 23 26 29 32
> (2+3) * x     #See the difference
 [1]  5 10 15 20 25 30 35 40 45 50
> sqrt(x)         #Square roots
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278
> x %% 4          #This is the integer divide (modulo) operation
 [1] 1 2 3 0 1 2 3 0 1 2
> y <- 3+2i     #R does complex numbers
> Re(y)           #The real part of the complex number
[1] 3
> Im(y)           #The imaginary part of the complex number
[1] 2
> x * y
 [1]  3+ 2i  6+ 4i  9+ 6i 12+ 8i 15+10i 18+12i 21+14i 24+16i 27+18i 30+20i
```

Now that you understand working with numeric vectors, we are ready to explore additional vector arithmetic. First, create a vector:

```
> x <- c(1:10)       #Create a vector
```

Note that the `c()` function is not really needed here, as my technical reviewer pointed out! You can simply use:

```
> x <- 1:10
```

And get the same result. Following are some other possibilities, along with the results for x, y, and z.

```
> y <- seq(10)       #Create a sequence
> z <- rep(1,10)     #Create a repetitive pattern
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> y
 [1]  1  2  3  4  5  6  7  8  9 10
> z
 [1] 1 1 1 1 1 1 1 1 1 1
```

As mentioned previously, R often allows users to avoid explicit looping by the use of *vectorized* operations. Looping *implicitly* through a vector is many times faster than looping explicitly, and makes the resulting R code more compact. As you will learn in the following chapter, you can loop explicitly when you need to, but should try to avoid this if possible. Vectorized operations on a single vector include many built-in functions, making R powerful and efficient. It is possible to apply many functions to data frames as well, though not all functions "do" data frames, as you will see. We can work around this by using other features and functions of R.

Although it takes some getting used to, R's treatment of vectors is logical. As we discussed earlier, a vector must have a single mode. You can check the mode of an object using the `mode()` function or using the `typeof()` function. As you have seen already in the output, R, unlike some other languages, begins its indexing with 1, not 0.

Adding Elements to a Vector

When you add elements, you are reassigning the vector. For example, see what happens when we append the numbers 11:15 to our x vector:

```
> x <- c([1:10])
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Now, let us reassign x by adding the numbers 11 through 15 to it. We are taking x and then appending (concatenating, in computer jargon) the sequence 11, 12, 13, 14, 15 to the 10-element vector to produce a 15-element vector we now have stored as x.

```
> x <- c(x, 11:15)
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

As you already know, we can use the sequence operator to create a vector. We can obtain the length of the vector with the `length()` function, the sum with the `sum()` function, and various statistics with other built-in functions to be discussed and illustrated later. To make our example a little more interesting, let us imagine a discrete probability distribution. We have a vector of the values of the variable and another vector of their probabilities. Here is a very interesting distribution known as Benford's Distribution, which is based on Benford's Law. The distribution gives the probability of first digits in numbers occurring in many (but not all) kinds of data. Some data, such as financial data, are well described by Benford's Law, making it useful for the investigation of fraud. We will return to this example later when we have the background to dig deeper, but for now, just examine the distribution itself. We will call the first digit `V`. Table 1-1 lists the first digits and their probabilities.