Namir's R 101 Tutorial

by

Namir Shammas

Dedication

I dedicate this tutorial, with an immense sense of gratitude and thankfulness, to the fine Boston College Jesuits who spent decades working to educate students in Baghdad College and Al-Hikmat University. They instilled in us the power to be self-taught! No words can fully thank them for that gift alone.

Table of Contents

I know what you're thinking. "Did he fire six shots or only five?" Well, to tell you the truth, in all this excitement I kind of lost track myself." But being as this is a .44 Magnum, the most powerful handgun in the world, you've got to ask yourself one question: Do I feel lucky? Well, do ya, punk?

"Dirty" Harry Callahan

Introduction

The Lineage of R

I have been very impressed with the power of the R programming language. When I first learned of the S programming language in the eighties, I desperately wanted to access S, but it was not available for the DOS-based PC machines which were popular in those days. Years later, I learned that S begat S-Plus which also ran on big machines. I was finally elated to learn that S-Plus had an open source cousin, the R language, which was available for Windows!

R at a Glance

R is a language that relies heavily on built-in functions and community-contributed function libraries. The language handles vectors, matrices, lists, and data tables in a rather advanced manner. Rather than attacking R in one Herculean effort, I chose to learn R in waves. I was very impressed that it supported very powerful functions that performed multiple linear regression, nonlinear regression, and time series analysis! While the learn-in-waves approach seemed not so taxing in one aspect, it was taxing in another aspect—I had to relearn many of the basic workings of R just about every time I took a stab at the language. Finally, I decided to write a tutorial for myself. I also decided to share it with the public in hope that it helps fellow enthusiasts and R newbie folks like me.

About This Tutorial

This tutorial is meant to be the first one about R. It covers the basics of the R language. Topics like plotting and graphics will be covered in a separate tutorial—Namir's R 102 Plotting Tutorial. Other tutorials will probably cover topics like object-oriented programming in R, personal selections of R functions that perform linear regression, nonlinear regression, time series, and multivariable optimization.

The book market has taken well to R. This is a good sign, since publishers are conservative business people. The page count of the various books on the market varies from small to medium, with a few books having high-page counts. Many small books cover the basics of R in a minimal fashion, preferring to focus on specific function libraries for R.

I chose to write a tutorial about the aspects of R that matter to me. I hope that these aspects also matter to you. A good book of R would occupy at least 500 pages. I am sharing this brief tutorial with you so you can get moving with R.

I used R versions 2.9.2 and 2.9.10 in writing this tutorial.

What I Expect You to Already Know

In this tutorial I assume that you know the following:

- How to download and install R on your machine.
- Are familiar with programming basics and know how to program in at least one language.
- How to use the R programming environment.
- How to enter basic commands at the R command prompt.
- How to download and install libraries and packages
- How to edit, save, and run scripts
- How to save your workspace
- How to use, save, and load the command history
- How to save the output of a session to a file, using the function sink().

So here we go!

The R Command Line Prompt

The Basics of Command Line Input

While R operates in a GUI environment, the R Console window allows you to type in commands at the command line prompt. This prompt appears as the > character followed by a space. It tells you that R is ready for you to enter a new command. In this tutorial, you always type what comes after the prompt (that is, the ″> ″). You can use the up and down arrow keys on your keyboard to scroll up and down previous commands. You can use the left and right arrows to edit the previous commands and re-execute them. This is a good time-saving feature.

Entering Commands that Span Multiple Lines

It is also worth mentioning that R permits you to type a command (or even a function definition) that spans multiple lines. As you enter each line, R quickly scans your input to determine if it constitutes a complete command or the end of the current command. If it is, the R interpreter processes that command and displays results or an error message if the command has incorrect data or syntax. If the line you type is an incomplete command or has not yet completed a command, the R interpreter displays a plus character followed by a space as a continuation prompt. Such a prompt tells you that R expects more input to complete the command you started keying in.

Entering Multiple Commands on the Same Line

While we are on the subject of entering commands, I’d like to point out that R allows you to enter multiple commands on one line. Use the semicolon character to separate each command. Here is an example that uses several commands to calculate a good approximation for pi:

```
> x=355; y=113; z=x/y; z
[1] 3.141593
```

A Valuable Trick to Learn

One of the features in R that is worthwhile to learn early on is the one that allows you to store a result in a variable and view that result in a single command. This tutorial typically asks you to store the result of a function or an expression in a variable and then type in the name of that variable, in a separate command, to view the result. You can save yourself some typing by enclosing the entire first command in parentheses. R responds to this enclosure by storing the result in the variable you specify AND displaying that result! Here is an example. Type in the following commands:

```
> z=355/113
> z
[1] 3.141593
```

The first command calculates an approximation for pi and then stores that value in the variable z. The second command displays the value in variable z. You can combine the above commands into the following single command:

```
> (z=355/113)
[1] 3.141593
```

If you remember this valuable tip, use it in the tutorial to save yourself some typing.

Script Files

Regarding script files, you can open new script windows and type in script code and script code snippets for testing. You can select code lines and execute them by pressing the CTRL+R keys. A script window may hold multiple groups of code. You select the group you want and run its code. This approach is good for quick testing and retesting. If you write an R function you can also test it that way. You can then edit the function to fix errors, enhance or modify the output, or modify what the code does. You can use the CTRL+R keys to retest the function and when you are satisfied with the code save it to an .r file. You can then load the function from the .r file using the source(filename) function at the command line prompt.

R allows you to key in a multiline function from the command line. This feature works well for functions that are defined in one or two lines. The more lines you type, the more error prone the process becomes. Besides, R does not offer a way to neatly edit previous lines of code. Your best bet is to use the R script window or even your favorite text editor.

Case Sensitivity

R is a case sensitive language. You need to make sure that you enter the names of variables, functions, and other language constructs in the correct case.

About R Functions

Before you dive into learning how to work with R, you should know that it relies heavily on functions loaded from a whole collection of libraries. The typical make up of an R function is one that uses many parameters, most of which are assigned default values. Such default values represent typical or optimum settings. The advantage of this feature is that it lets you focus on supplying arguments to only the parameters that you need to work with. Consequently, the commands you type can typically be short. The more control you want over what the function does, the more arguments you need to specify when you call that function.

Getting More Help

Before I discuss any topic, I want to point out that R offers a lot of online help. To get online documentation on a specific function you can type one of the following at the command prompt:

- Type the question mark followed by the name of the function you want get help for. For example if you type **?matrix** you get the online help documentation on the matrix function.
- Type help() and include the name of the function you want to query inside the parentheses. For example typing **help(matrix)** at the command prompt provides the online help documentation for the matrix function.
- Type apropos() and include the name of a function in a pair of double quotes. R will display a list of functions related to the function that you specified.
- Type example() and include the name of a function. R executes a script that illustrates that function.

Here is an example for using the apropos() function. Type in the following command to inquire about all function that are related to the plot() function:

```
> apropos("plot")
 [1] ".__C__recordedplot"  "assocplot"           "barplot"
 [4] "barplot.default"     "biplot"              "boxplot"
 [7] "boxplot.default"     "boxplot.matrix"      "boxplot.stats"
[10] "cdplot"              "coplot"              "fourfoldplot"
[13] "interaction.plot"    "lag.plot"            "matplot"
[16] "monthplot"           "mosaicplot"          "multi.3d.plot"
[19] "multi.3d.plot2"      "plot"                "plot.default"
[22] "plot.density"        "plot.design"         "plot.ecdf"
[25] "plot.lm"             "plot.lr2"            "plot.lr3"
[28] "plot.mlm"            "plot.new"            "plot.spec"
[31] "plot.spec.coherency" "plot.spec.phase"     "plot.stepfun"
[34] "plot.ts"             "plot.TukeyHSD"       "plot.window"
[37] "plot.xy"             "preplot"             "qqplot"
[40] "recordPlot"          "replayPlot"          "save.plot"
[43] "savePlot"            "screeplot"           "spineplot"
[46] "sunflowerplot"       "termplot"            "ts.plot"
```

To illustrate the example() function, type in the following command to obtain an example for the function matrix():

```
> example(matrix)

matrix> is.matrix(as.matrix(1:10))
[1] TRUE

matrix> !is.matrix(warpbreaks)# data.frame, NOT matrix!
[1] TRUE

matrix> warpbreaks[1:10,]
   breaks wool tension
1      26    A       L
2      30    A       L
3      54    A       L
4      25    A       L
5      70    A       L
6      52    A       L
```

```
7      51    A       L
8      26    A       L
9      67    A       L
10     18    A       M

matrix> as.matrix(warpbreaks[1:10,]) #using as.matrix.data.frame(.) method
   breaks wool tension
1  "26"   "A"  "L"
2  "30"   "A"  "L"
3  "54"   "A"  "L"
4  "25"   "A"  "L"
5  "70"   "A"  "L"
6  "52"   "A"  "L"
7  "51"   "A"  "L"
8  "26"   "A"  "L"
9  "67"   "A"  "L"
10 "18"   "A"  "M"

matrix> # Example of setting row and column names
matrix> mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol=3, byrow=TRUE,
matrix+                dimnames = list(c("row1", "row2"),
matrix+                                c("C.1", "C.2", "C.3")))

matrix> mdat
     C.1 C.2 C.3
row1   1   2   3
row2  11  12  13
```

Do not hesitate to use the online help if anything in this tutorial sounds vague to you.

Data Types

Data Type Summary

R supports various data types, most notably the following:

- Logical (Boolean) values (TRUE and FALSE)
- Integers
- Floating point numbers.
- Complex numbers
- Vectors
- Matrices of two or more dimensions
- Character strings
- Lists
- Factors (enumerated types)
- Data frames
- Time series
- Dates

We will encounter the different data types as we progress in the tutorial.

In addition to the above data types, numerous R functions handle vectors, matrices, data frames, and lists with missing data. R uses the designation NA (short for *Not Available*) to represent a missing value. Do not confuse NA with NaN which is a designation for *Not A Number*. R uses NaN to flag the result of illegal mathematical operations.

The as.xxx() and is.xxx() Functions Family

For now it is worthwhile to list some of the family of as.*xxxx*() function in R that performs data type conversion. Table 1 shows the functions that perform data type conversion.

Table 1. Data type conversion functions in R.

Function	***Conversion***
as.array(x)	Converts to an array
as.character(x)	Converts to a string
as.complex(x)	Converts to a complex number
as.dataframe(x)	Converts to a data frame
as.Date(x)	Converts to a date
as.integer(x)	Converts to an integer
as.list(x)	Converts to a list
as.logical(x)	Converts to a Boolean type
as.matrix(x)	Converts into a matrix
as.numeric(x)	Converts into a numeric value
as.real(x)	Same as as.single(x)
as.single(x)	Converts to a double-precision floating point number
as.table(x)	Converts into a table
as.vector(x)	Converts into a vector

R also provides with a similar family of is.*xxxx*() functions that tests for a data type. For example, is.integer(x) tests if object x stores an integer, is.logical(x) tests if object x stores a logical value, and so on. A special mention goes to the logical function is.na() as one that tests for missing data. R uses NA to represent missing data. The NA stands for *Not Available*. R also offers NULL to signal that the object has no value.

Basic Mathematical Operators

The following table shows the basic mathematical operators supported by R.

Table 2. Basic mathematical operators in R.

Operator	***Purpose***	***Example***
+	Add	1 + 3
-	Subtract or negate	4 – 3

Operator	*Purpose*	*Example*
*	Multiply	5 * 3
/	Divide	355 / 113
%/%	Integer division	355 %/% 113
%%	Modulo	133 %% 3
^	Raise to power	2^4

Variables

Naming Variables

The names of variables are case sensitive and include:

- Lowercase characters
- Uppercase characters
- The dot character
- The underscore character, which cannot be the leading character in the name of a variable or a function.
- Digits, which cannot be the leading characters in the name of a variable or function.

Examples of valid variables are:

```
X
y
x1
x1.best
sort.order
descending.sort.order
h2o
Best.fit
best.fit
best_fit
best.linear.fit
A.inv
```

It is worth pointing that in the above examples, the variables Best.Fit and best.fit refer to different variables, since R is case sensitive.

Don't confuse the decimal in the name of R variables and functions with the member access operator. Such operator is commonly used in *OTHER* object-oriented programming languages to access member functions or data members of objects. You can think of the dot character in R as the underscore which is common in many other programming languages.

Assigning Values to Variables

You can assign values to variables using the following operators:

- The <- operator
- The -> operator
- The = operator

The <- operator seems to be the most commonly used assignment operator I have seen in R books and articles. In fact, when you look at code with the <- operator you can safely assume that you are looking at code in R (or S-Plus). In this tutorial I will use the = operator most of the time.

Here are examples for each operator:

```
> x <- 355/113
> 355/113 -> x
> x = 355/113
```

Remember that the "> " characters represent the R command line prompt. You always type what appears after the "> ".

Removing Some or All Variables from the Workspace

The function rm() removes the variables specified in its list of arguments. This function can take any number of arguments. For example, to remove the variables x, y, and z, type in the following command:

```
> rm(x, y, z)
```

You can use the list of all variables that you can obtain with the function ls(), to erase all of the variables from the workspace using function rm(). Here is the command that uses functions rm() and ls():

```
> rm(list = ls())
```

If you execute the above rm() command by accident and want to restore the variables and functions in your environment, exit the R environment WITHOUT saving the workspace. This includes refusing to save the workspace when the R environment prompts you to do so, as you exit. Once you are out of R, load it again and you will have your variables and functions back. This scheme works only if you have saved the R workspace in a previous session. The R environment automatically reloads the last saved workspace. You can load other previously saved workspaces using the **File | Load Worskspace …** menu. Saving the workspace for important work is highly recommended.

Vectors

Vectors in General

Vectors (or one-dimensional arrays) contain a collection of similarly typed data. R supports vectors of integers, real numbers, complex numbers, logical values, strings, and other data types.

The most common way to create a vector is R is to use the combine function c()—one of the most popular functions in R. The arguments for this function appear as a comma-delimited sequence of values and/or variables. The simplest case for using c() to create a vector looks like:

```
> x <- c(12.2, 32, 7, 45.32)
```

You can use function c() to combine explicit numeric values with the values already stored in one or more vectors, such as:

```
> y = c(98.43, 76.32, x)
```

Typing y at the R prompt yields:

```
[1] 98.43 76.32 12.20 32.00 7.00 45.32
```

You can also create vectors made of other vectors, such as:

```
> X1 = c(x, y, x)
```

Which creates the vector X1 that contains the values in the smaller vectors x, y, and x again. You can also type:

```
> X2 = c(x, 1.234, x)
```

The above command creates the vector X2 that contains twice the sequence of numbers in the smaller vector x, separated by the number 1.234.

Typing X1 at the prompt yields:

```
[1] 12.20 32.00  7.00 45.32 98.43 76.32 12.20 32.00  7.00 45.32 12.20 32.00
[13]  7.00 45.32
```

Typing X2 at the prompt yields:

```
[1] 12.200 32.000 7.000 45.320 1.234 12.200 32.000  7.000 45.320
```

You can create (or better yet, initialize) an empty vector using the function c() with no arguments. Here is an example:

```
> ev = c()
```

The above command creates an empty vector ev. You can add to that vector any values using the function c(). Here is an example:

```
> ev = c(ev, 11, 22)
> ev = c(ev, 1:10)
> ev
[1] 11 22 1 2 3 4 5 6 7 8 9 10
```

And so on! You can think of the combine function c() as a vector concatenation function.

Mixing Types in Function c()

In the case of mixing values from different data types, the function c() works at finding the most suitable common data type and apply that type in creating an array of values with a consistent data type. Consider the case of mixing a string with integers, real numbers, complex numbers, and logical values. The function c() returns an array of strings. For example mixing the string "5" with the numbers 1, 2, 3, 4, the complex number 1+2i, and the logical TRUE in function c() yields an array of strings:

```
> xx = c(1,2,3,4,"5", 1+2i, TRUE)
> xx
[1] "1" "2" "3" "4" "5" "1+2i" "TRUE"
```

What if you remove the element "5" from the above call to function c()? What is the result? Typing the modified command yields an array of complex numbers:

```
> xx = c(1,2,3,4, 1+2i, TRUE)
> xx
[1] 1+0i 2+0i 3+0i 4+0i 1+2i 1+0i
```

The function c() determines that the complex type is the most common type for the combined values. Even the value TRUE becomes the complex number 1+0i!

Vectors Having Numerical Sequences

R allows you to create vectors that have numerical sequences, and long ones at that. This feature saves you from excessive typing, not to mention saving you from likely typo errors.

The simplest format for creating a straightforward sequence uses the following general syntax:

first.value:last.value

For example, you can create the vector X3 that has the sequence of integers from 1 to 25 by typing the following command:

```
> X3 = 1:25
```

If you type X3 at the prompt you get the values in that vector:

```
[1] 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

You can just as easily create the vector X4 that has the sequence of -1 to -25 by typing:

```
> X4 = -1:-25
```

To view the contents of vector X4, type its name at the prompt. You get:

```
[1] -1  -2  -3  -4  -5  -6  -7  -8  -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -
19
[20] -20 -21 -22 -23 -24 -25
```

R also offers the function seq() to create more elaborate numerical sequences that increment by values other than 1. Such increments include negative and non-integer increments. The function can also let you specify how many values to generate. The function seq() has two flavors, each of which takes three arguments:

seq(from, to, by)

seq(from, to, length.out)

The first form of the seq() function allows you to specify the range of values (**from** and **to**) and the increment value using the parameter **by**. The range of values as well as the increment determine how many elements the function seq() creates.

The second flavor of function seq() allows you to explicitly specify the number of elements to create by using an argument for the parameter **length.out**.

You can create the vector X5 that has the sequence of 1, 3, 5, …, 25 by typing the following command:

```
> X5 = seq(1, 25, 2)
```

To view the contents of vector X5, type its name at the prompt and you get:

```
[1]  1  3  5  7  9 11 13 15 17 19 21 23 25
```

You can also specify the name of the parameters of seq() and place them in any order (since you tell R which argument it is and what value is assign to it). For example, you can create the vector X5b which has the same contents as vector X5 by typing:

```
> X5b = seq(by=2, to=25, from=1)
```

When you type X5b at the prompt, you get:

```
[1]  1  3  5  7  9 11 13 15 17 19 21 23 25
```

Which is the same sequence of values stored in variable X5.

You can also create the above array of values using the following form of the function seq():

```
> X5b = seq(from=1, to=25, length=13)
```

You can also use the rep() function to create a repeated sequence of numbers for a desired length. The syntax for the rep() function is:

rep(object, replications)

For example, to create a sequence of five ones, you type;

```
> rep(1, 5)
```

And you get:

```
[1] 1 1 1 1 1
```

The first argument for function rep() need not be limited to scalars. You can include a range of values and ask function rep() to repeat that range. Here is an example where you can repeat four times the range 1, 2, and 3:

```
> rep(1:3,4)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

You can also use the combine function c() as the first argument for function rep(). Here is an example that repeats three times the sequence of 1, 10, and 100:

```
> rep(c(1,10,100),3)
[1]   1  10 100   1  10 100   1  10 100
```

The function length() returns the number of elements in a vector. For example, if you type the following at the prompt:

```
> length(X3)
```

R displays the following result:

```
[1] 25
```

Accessing Vectors

R supports the typical way of accessing specific vector elements, and more!

By accessing vectors I mean the ability to retrieve values from particular vector elements AND to assign values to specific vector elements. This feature also applies to matrices and higher-dimensional arrays.

You can use an index in square brackets to retrieve a value from a vector or assign a new value. For example to access the second element in vector X1, you type:

```
> X1[2]
```

```
[1] 32
```

To assign the value of 30 to that same element you type:

```
> X1[2] = 30.00
```

When you type X1 at the prompt you get:

```
 [1] 12.20 30.00  7.00 45.32 12.20 32.00  7.00 45.32 12.20 32.00  7.00 45.32
```

The above output shows that the second element has the new value of 30.00.

R supports array slicing. It allows you to access multiple elements when you use an index that is itself a vector! For example to simply display the 2nd, 4th, and 6th elements of vector X1, you can first create the vector of indices i:

```
> i = c(2,4,6)
```

Then, when you type X1[i] at the prompt, R displays the following result:

```
[1] 30.00 45.32 32.00
```

Of course, you can skip using the vector i and directly access the 2nd, 4th, and 6th elements of vector X1 in the following single command:

```
> X1[c(2,4,6)]
[1] 30.00 45.32 32.00
```

R also allows you to use negative indices to EXCLUDE elements at specific indices. For example, to exclude the 2nd, 4th, and 6th elements of vector X1 from the list of display elements, you first create the vector of indices j:

```
> j = c(-2,-4,-6)
```

Then, when you type X1[j] at the prompt, R displays the following result:

```
[1] 12.20  7.00 12.20  7.00 45.32 12.20 32.00  7.00 45.32
```

The above output shows the elements of vector X1, except for the 2nd, 4th, and 6th elements.

In addition to using the function c() you can use the syntax from:to to specify a range of contiguous elements to include or exclude. Remember that accessing a portion of a vector works with retrieving values or overwriting existing ones.

You can use the same approach of selecting or excluding vector elements in vector operations.

Naming Vectors Elements

You can associate names with the elements of a vector. You can then employ these names to access the associated vector elements. Here is an example for associating names to a vector:

```
> nv = c(55, 67, 41, 71)
> names(nv) = c("HP55", "HP67", "HP41C", "HP71B")
```

When you type the name of the vector nv you get the names and the values associated with the vector:

```
> nv
 HP55  HP67 HP41C HP71B 
   55    67    41    71 
```

You can access the third vector element by using the name associated with that element:

```
> nv["HP41C"]
HP41C 
   41 
```

Using named vector elements you can emulate enumerated types in programming languages like C and C++. In such emulation, the element name corresponds to an enumerated identifier and the element's value corresponds to the value associated with the enumerated identifier. As the above example shows, these values need not be contiguous or fall in a simple sequence.

Vector Operations

R supports vector operations in a manner similar to Matlab.

Vectors and Scalars

You can create a new vector by using an existing vector and a scalar. For example, you can create the vector Y3 by adding 2 to each element of vector X3:

```
> Y3 = 2 + X3
```

Typing Y3 yields the following output:

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

You can also create the vector Y4 by doubling the values in vector X3:

```
> Y4 = 2 * X3
```

Typing Y4 yields the following output:

```
[1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
50
```

And finally, you can create vector Y5 by multiplying each value in vector X3 by 2 and subtracting 5 from each result:

```
> Y5 = 2 * X3 - 5
```

Typing Y5 yields the following output:

```
[1] -3 -1  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43
45
```

The above examples make the point that R allows you to create/update vectors using expressions that involve other vectors. The beauty of this feature is that R, like Matlab, allows you to do vector math without using loops. In fact using vector math is faster than using loops that requires more time to access individual vector elements.

Functions of Vectors

You can also apply functions to vectors in order to obtain new vectors. For example, you can create vector S3 by taking the square root of each element in vector X3:

```
> S3 = sqrt(X3)
```

Typing S3 yields the following output:

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278 3.316625 3.464102 3.605551 3.741657 3.872983 4.000000
[17] 4.123106 4.242641 4.358899 4.472136 4.582576 4.690416 4.795832 4.898979
[25] 5.000000
```

You can create vector S4 as an array of sine values based on the values of vector X3:

```
> S4 = sin(X3*pi/180)
```

Typing S4 at the prompt yields the following output:

```
[1] 0.01745241 0.03489950 0.05233596 0.06975647 0.08715574 0.10452846
 [7] 0.12186934 0.13917310 0.15643447 0.17364818 0.19080900 0.20791169
[13] 0.22495105 0.24192190 0.25881905 0.27563736 0.29237170 0.30901699
[19] 0.32556815 0.34202014 0.35836795 0.37460659 0.39073113 0.40673664
[25] 0.42261826
```

The above examples show simple expressions. You can use more elaborate expressions that involve vectors and scalars.

Vectors and Vectors

Adding Vectors

You can create vectors using other vectors. For example, you can create the vector P1 by adding the elements in vectors S3 and S4:

```
> P1 = S3 + S4
```

Typing P1 at the prompt yields:

```
 [1] 1.017452 1.449113 1.784387 2.069756 2.323224 2.554018 2.767621 2.967600
 [9] 3.156434 3.335926 3.507434 3.672013 3.830502 3.983579 4.131802 4.275637
[17] 4.415477 4.551658 4.684467 4.814156 4.940944 5.065022 5.186563 5.305716
[25] 5.422618
```

Multiplying Vectors

You can also create the vector P2 as the product of vectors S3 and S4:

```
> P2 = S3 * S4
```

Typing P2 at the prompt yields:

```
 [1] 0.01745241 0.04935534 0.09064854 0.13951295 0.19488617 0.25604140
 [7] 0.32243598 0.39364097 0.46930340 0.54912375 0.63284184 0.72022722
[13] 0.81107256 0.90518885 1.00240185 1.10254942 1.20547942 1.31104807
[19] 1.41911868 1.52956058 1.64224826 1.75706067 1.87388066 1.99259447
[25] 2.11309131
```

Functions of Vectors

You can also create the vector P3 as the sum of the functions of vectors S3 and S4:

```
> P3 = S3^2 + sqrt(S4)
```

Typing P3 at the prompt yields:

```
[1]  1.132108  2.186814  3.228771  4.264115  5.295222  6.323309  7.349098
 [8]  8.373059  9.395518 10.416711 11.436817 12.455973 13.474290 14.491856
[15] 15.508743 16.525012 17.540714 18.555893 19.570586 20.584825 21.598638
[22] 22.612051 23.625085 24.637759 25.650091
```

Vector Multiplication

The examples above show R performing operations on corresponding elements in the vectors involved. To multiply vectors as such you need to use the %*% operator to tell R that you are doing a vector multiplication:

```
> P4 = S3 %*% S4
```

Typing P4 yields:

```
         [,1]
[1,] 22.50076
```

Outer Products

The last vector operator is the outer product %o% and function outer() which has the following definition:

outer(x, y, FUN="*", ...)

Where parameters **x** and **y** are numeric vectors that can have different sizes. The parameter **FUN** specifies the function to apply in calculating the resulting matrix. The number of rows and columns for that matrix is equal to length(x) and length(y), respectively. The resulting matrix (call it M) has the following indexing and value assignment scheme:

M[c(arrayindex.x, arrayindex.y)] = FUN(x[arrayindex.x], y[ayyarindex.y], …)

The operations X %o% Y is equivalent to calling outer(X, Y). Here is an example of using the %o% operator. Execute the following commands to initialize two vectors, v1, and v2, of unequal lengths, and then calculate and display the outer product for these vectors:

```
> v1=1:5
> v2=seq(.1,1,0.1)
> v1 %o% v2
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9     1
[2,]  0.2  0.4  0.6  0.8  1.0  1.2  1.4  1.6  1.8     2
[3,]  0.3  0.6  0.9  1.2  1.5  1.8  2.1  2.4  2.7     3
[4,]  0.4  0.8  1.2  1.6  2.0  2.4  2.8  3.2  3.6     4
[5,]  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5     5
```

Vector v1 has 5 elements and vector v2 has 10 elements. The resulting outer product of these vectors is a matrix that has 5 rows and 10 columns. Calling function outer() with just the names of the vectors v1 and v2 yields the same matrix:

```
> outer(v1, v2)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9     1
[2,]  0.2  0.4  0.6  0.8  1.0  1.2  1.4  1.6  1.8     2
[3,]  0.3  0.6  0.9  1.2  1.5  1.8  2.1  2.4  2.7     3
[4,]  0.4  0.8  1.2  1.6  2.0  2.4  2.8  3.2  3.6     4
[5,]  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5     5
```

The function outer() is very useful in calculating data used to plot 3D surfaces and contours. In such case, the argument for parameter FUN represents the equation used to calculate the surface given values for x and y.

Basic Statistical Functions

R offers a set of functions to support basic statistical calculations. The next table lists these functions. The Example column assumes that the vectors xd and yd are defined as:

```
xd = 1:10
yd = c(1,9,4,2,4,7,8,3,5,6)
```

Table 3. List of basic statistical functions.

Function	***Purpose***	***Example***
sum(x)	Sum of values in vector x.	sum(xd) returns 55
prod(x)	Product of values in vector x.	prod(xd) yields 3628800
mean(x)	Mean of values in vector x.	mean(xd) gives 5.5
sd(x)	Standard deviation of values in vector x.	sd(xd) returns 3.027650
median(x)	Median of values in vector x.	median(xd) gives 5.5
var(x)	Variance of values in vector x	var(xd) yields 9.166667
var(x,y)	Covariance of the values in vectors x and y	var(xd,yd) returns 1.833333
cov(x,y)	Covariance of the values in	cov(xd,yd) returns 1.833333

Function	*Purpose*	*Example*
	vectors x and y	
cor(x,y)	Correlation between the values in vectors x and y.	cor(xd,yd) returns 0.2327814

It is worthwhile mentioning that the functions listed in the above tables also work with matrices and vector/matrix expressions.

Special Vector Operations

R offers a collection of functions that works on vectors. The function sort(x) returns a vector whose values are sorted in ascending order. The function rev(x) returns a vector whose values are arranged in a reverse order (regardless of whether the source vector's values are sorted or not). The function order(x) returns the rank indices of vector x that show the rank of each element in that vector.

Sorting Vectors

Here is an example for sorting and reversing ordered arrays. First create the vector x with sample data:

```
> x = c(55, 67, 41, 34, 29, 65, 45)
```

Next, sort the elements in vector x using function sort():

```
> sort(x)
[1] 29 34 41 45 55 65 67
```

Reversing Vector

Reverse sort the data in vector x by using the functions rev() and sort():

```
> rev(sort(x))
[1] 67 65 55 45 41 34 29
```

Perform a descending order sort using the data in vector x:

```
> sort(x, decreasing = TRUE)
[1] 67 65 55 45 41 34 29
```

Getting the Rank of Vector Elements

The function order(x) displays the rank indices for the vector x that indicate the rank of each vector element. You can use the results of function order() to sort the values of a vector. The following commands illustrate how function order() works:

```
> order(x)
[1] 5 4 3 7 1 6 2
> x[order(x)]
[1] 29 34 41 45 55 65 67
```

The function order(x) displays the rank indices for vector x. The command x[order[x]) displays the sorted elements of vector x and has the same effect as typing sort(x).

Finding the Largest and Smallest Values in a Vector

The functions which.max() and which.min() return the index of the largest and smallest element in a vector, respectively. Applying these functions to the vector x you get:

```
> x = c(55, 67, 41, 34, 29, 65, 45)
> which.max(x)
[1] 67
> which.min(x)
[1] 29
```

Matrices

Matrices in General

R supports matrices that you can create from the command line. The basic components and information for matrices in R are:

- A sequence of data--basically a vector
- The number of rows in the matrix
- The number of columns in the matrix
- The matrix-fill orientation which is either by row or by column. The default orientation is by columns.

Creating a Matrix

The R interpreter takes the values in the data sequence and places them in the specified matrix rows and columns, following the specified (or default) matrix-fill orientation. The general syntax is:

matrix(data, nrow = 1, ncol = 1, byrow = FALSE).

If you omit the parameter **nrow** or **ncol**, The function calculates the value of the omitted parameter by using the number of data elements divided by the value of the rows or columns specified. Using both parameters ensures that you obtain the matrix with the specified dimensions.

Here is an example for creating a simple matrix M1:

```
> M1 = matrix(c(1,2,3,4), nrow = 2, ncol = 2, byrow = TRUE)
```

When you type M1 at the prompt, R displays the following data for the matrix:

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

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

Consider the matrix M1b that fills its data by column:

```
> M1b = matrix(c(1,2,3,4), nrow = 2, ncol = 2, byrow = FALSE)
```

When you type M1b at the prompt, you get the following output:

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

The matrices M1 and M1b contain the same values, albeit arranged differently among the matrix elements.

Sizing Up a Matrix

To obtain the dimensions of a matrix, R offers the dim() function. This function returns a vector of two values--the number of rows and the number of columns. To determine the dimensions of the matrix M1, type the following command:

```
> dim(M1)
[1] 2 2
```

The above command tells you that matrix M1 has 2 rows and 2 columns.

R also allows you to separately query the number of rows and columns of a matrix using the nrow() and ncol() functions, respectively. Applying these functions to matrix M1, type in the following commands:

```
> nrow(M1)
[1] 2
> ncol(M1)
[1] 2
```

Building Matrices with Numeric Sequences

You can also use the seq() function to build matrices. Here is an example for creating the matrix M2 using the sequence of 1 to 25, filled column-wise:

```
 > M2 = matrix(seq(1,25), nrow = 5, ncol = 5)
```

Typing M2 at the prompt yields the following output:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

If the number of data elements is less than the number of matrix elements, the R interpreter reuses as many times needed the sequence of supplied data. If the number of matrix rows or columns is not an integer multiple (or sub-multiple) of the number of data elements, you get a warning message from the R interpreter. The interpreter still goes ahead and fills the matrix using the available data elements.

For example:

```
> M2b = matrix(seq(1,5), nrow = 5, ncol = 5)
```

Yields the following when you type M2b:

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

However:

```
> M2c = matrix(seq(1,7), nrow = 5, ncol = 5)
```

Yields the following warning message:

```
Warning message:
In matrix(seq(1, 7), nrow = 5, ncol = 5) :
  data length [7] is not a sub-multiple or multiple of the number of
rows [5]
```

When you type M2c you get the following output:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6    4    2    7
[2,]    2    7    5    3    1
[3,]    3    1    6    4    2
[4,]    4    2    7    5    3
[5,]    5    3    1    6    4
```

Creating Diagonal Matrices

To create a diagonal matrix with non-zero elements on the diagonal, R offers the diag() function, which has the following syntax:

diag(value_at_diagonal, num_rows, num_columns)

To create a diagonal matrix Md with five rows and five columns, you type:

```
> Md = diag(1, 5, 5)
```

When you type Md at the prompt, you get the following output:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

If you replace the first argument of function diag() with, say 11. Typing diag(11, 5, 5) yields the following output:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    11    0    0    0    0
[2,]    0    11    0    0    0
[3,]    0    0    11    0    0
[4,]    0    0    0    11   0
[5,]    0    0    0    0    11
```

Combining Matrices

Just like with vectors, you can create matrices from smaller ones. R allows you to create a matrix by merging two matrices either by rows or by columns. The functions rbind() and cbind() binds matrices by rows and by columns, respectively.

To illustrate the use of function cbind(), consider merging the matrices Mat1 and Mat2 into Mat12. The following commands define the matrices Mat1 and Mat2:

```
> Mat1 = matrix(1:9, nrow = 3, ncol = 3)
> Mat2 = matrix(11:22, nrow = 3, ncol = 4)
```

The two merged matrices must have the same number of rows for function cbind() to work.

Typing Mat1 you get:

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

And typing Mat2 you get:

```
     [,1] [,2] [,3] [,4]
[1,]   11   14   17   20
[2,]   12   15   18   21
[3,]   13   16   19   22
```

Now to merge the two matrices into Mat12, you type the following command:

```
> Mat12 = cbind(Mat1, Mat2)
```

When you type Mat12 you get:

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    4    7   11   14   17   20
[2,]    2    5    8   12   15   18   21
[3,]    3    6    9   13   16   19   22
```

To illustrate the use of function rbind(), consider merging the matrices Mat3 and Mat4 into Mat34. The following commands define the matrices Mat3 and Mat4:

```
> Mat3 = matrix(1:9, nrow = 3, ncol = 3)
> Mat4 = matrix(11:22, nrow = 4, ncol = 3)
```

The two merged matrices must have the same number of columns for function rbind() to work.

Typing Mat3 you get:

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

And typing Mat4 you get:

```
     [,1] [,2] [,3]
[1,]   11   15   19
[2,]   12   16   20
[3,]   13   17   21
[4,]   14   18   22
```

Now to merge the two matrices into Mat34, you type the following command:

```
> Mat34 = rbind(Mat3, Mat4)
```

When you type Mat34 you get:

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
[4,]   11   15   19
[5,]   12   16   20
[6,]   13   17   21
[7,]   14   18   22
```

Accessing Matrix Elements

R extends the features of accessing vectors to matrices. You can access individual matrix elements, access a sub-matrix within a matrix, or even exclude a sub-matrix within a matrix. Consider the matrix Mat34 which contains the following data:

```
     [,1] [,2] [,3]
[1,]    1    4    7
```

```
[2,]    2    5    8
[3,]    3    6    9
[4,]   11   15   19
[5,]   12   16   20
[6,]   13   17   21
[7,]   14   18   22
```

If you type Mat34[5, 2] you get:

```
[1] 16
```

To access the sub-matrix made up of rows 4, 5 and 6, and columns 2 and 3, you type:

```
> Mat34[4:6,2:3]
     [,1] [,2]
[1,]   15   19
[2,]   16   20
[3,]   17   21
```

Accessing Matrix Rows and Columns

You can access specific rows or columns in a matrix. To access a specific row you use the format mat[row,] leaving out the column index altogether. Likewise, to access a specific column you use the format mat[,col], leaving out the row index. To access a range of rows and columns you use the format mat[row.range,] and mat[,col.range], respectively. You can create these row ranges using the form from:to or the combine function c() to select indices that are not necessarily contiguous.

For example, to access the second row of Mat34, you type the following:

```
> Mat34[2,]
[1] 2 5 8
```

To access the second column in matrix Mat34, you type the following:

```
> Mat34[,2]
[1]  4  5  6 15 16 17 18
```

To access rows 2, 3, and 4 in matrix Mat34, you type the following:

```
> Mat34[2:4,]
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
[3,]   11   15   19
```

Similarly, to access columns 2 and 3 in matrix Mat34, you type the following:

```
> Mat34[,2:3]
     [,1] [,2]
[1,]    4    7
[2,]    5    8
[3,]    6    9
```

```
[4,]   15   19
[5,]   16   20
[6,]   17   21
[7,]   18   22
```

Naming Matrices Elements, Rows, and Columns

You can associate names with the elements of a matrix. R gives you three options:

1. The function names() permits you to tag each matrix element with a name.
2. The function rownames() allows you to tag the rows of a matrix.
3. The function colnames() permits you to tag the columns of a matrix.

You can use the name of a matrix element to access the corresponding matrix component. Using function names() allows you to access any matrix element by its associated name. Using function colnames() permits you to access any matrix column by name. Using function rownames() gives you access to any matrix row by name.

Here is an example for creating a matrix and then tagging its columns using the single letters X, Y, Z, and T:

```
> nm = matrix(1:12, nrow=3, ncol=4)
> nm
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> colnames(nm) = c("X", "Y", "Z", "T")
> nm
     X Y Z  T
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12
```

You can access the second column of matrix nm, by name, when you execute the following command:

```
> nm[,"Y"]
[1] 4 5 6
```

The above command uses the name of the second column "Y" to access the values in that column.

Matrix Operations

R supports operations between matrices and scalars, vectors, and other matrices.

Matrices and Scalars

Matrix operations with scalars are easy and straightforward. Such operations have the scalar values interact individually with each matrix element. You can add, subtract, divide, and multiply scalars and matrices.

Matrices and Vectors

R allows you to perform mathematical operations between vectors and matrices. R requires that the number of vector elements be equal to or less than the number of rows in the matrix. The vector-matrix operations occur on an element-to-element basis.

Here is an example that adds a vector to a matrix. Type in the following commands to create the matrix and then the vector:

```
> m=matrix(1:20,nrow=5,ncol=4)
> x=1:5
```

Now type the name of the matrix m:

```
> m
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

Type in the name of the vector x:

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

Now perform the vector to matrix addition by executing the following command to store the result in matrix m2:

```
> m2=m+x
```

To inspect the values in matrix m2, type in the matrix name at the command line:

```
> m2
     [,1] [,2] [,3] [,4]
[1,]    2    7   12   17
[2,]    4    9   14   19
[3,]    6   11   16   21
[4,]    8   13   18   23
[5,]   10   15   20   25
```

The values in matrix m2 are the sum of the values in matrix m and the vector x. The operations involve corresponding elements. For example:

m2[1,1] = m[1,1] + x[1]

m2[2,1] = m[2,1] + x[2]

…

m2[5,4] = m[5,4] + x[5]

In general you have:

m2[i, j] = m[i, j] + x[i]

If the vector x had less than 5 elements, the operation would wrap the vector elements. For example, if x had 4 values, then the scheme to calculate the elements of matrix m2 would be as follows:

m2[1,1] = m[1,1] + x[1]

m2[2,1] = m[2,1] + x[2]

m2[3,1] = m[3,1] + x[3]

m2[4,1] = m[4,1] + x[4]

m2[5,1] = m[5,1] + x[1]

m2[1,2] = m[1,2] + x[2]

m2[2,2] = m[2,2] + x[3]

…

Notice that for row 5, the operation uses x[1] since there is no x[5] defined. In the case of row 1 and column 2, the operation uses x[2], and so on. Here is what R produces if x has 4 elements:

```
> m=matrix(1:20,nrow=5,ncol=4)
> x=1:4
> m2=m+x
> m2
     [,1] [,2] [,3] [,4]
[1,]    2    8   14   20
[2,]    4   10   16   18
[3,]    6   12   14   20
[4,]    8   10   16   22
[5,]    6   12   18   24
```

Matrices and Matrices

R supports elements-by-element operations for addition, subtractions, multiplication, and division. Let's try the elements-by-element multiplication. If you type the following command, you get the subsequent output:

```
> M1 * M1b
     [,1] [,2]
[1,]    1    6
[2,]    6   16
```

Recalling that M1 and M1b contain the following matrices, respectively:

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

And

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

If you examine the operand and resulting matrices you see that the multiplication is done for corresponding matrix elements. In other words:

R(i,j) = Mat1(i,j) * Mat2(i,j) for all i = 1 to number of rows, and j = 1 to number of columns

R supports the same approach for the other basic operations. The rule to follow is that the operand matrices must have compatible dimensions. That is, the number of rows for both operand matrices must be equal, and the same goes for the number of columns. The resulting matrix will also have the same number of rows and columns as the operand matrices.

In addition to the element-by-element operations, R allows you to multiply two matrices using the %*% operator. In this case the number of columns of the first matrix operand must be equal to the number of rows of the second matrix operand. The resulting matrix will have the same number of rows as the first matrix operand and the same number of columns as the second matrix operand. Using matrices M1 and M1b, type the following command:

```
> M1c = M1 * M1b
```

And then type M1c. The result is:

```
     [,1] [,2]
[1,]    5   11
[2,]   11   25
```

The element at (1,1) = 1 * 1 + 2 * 2 = 1 + 4 = 5

The element at (2,1) = 1 * 3 + 2 * 4 = 3 +8 = 11

The element at (1,2) = 1 * 3 + 2 * 4 = 3 + 8 = 11

The element at (2,2) = 3 * 3 + 4 * 4 = 9 + 16 = 25

Matrix Transpose

The function t(x) returns the transpose of a matrix x. For example, if you type in:

```
>t(M1)
```

You get:

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

This is a matrix that is equal to the M1b matrix.

Matrix Determinant

The det() function returns the determinant of a matrix. To calculate and then display the determinant of matrix M1, enter the following commands:

```
> detM1 = det(M1)
> detM1
[1] -2
```

Now recall the elements of matrix M2:

```
> M2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Next calculate and display the determinant of matrix M2:

```
> detM2 = det(M2)
> detM2
[1] 0
```

The matrix M2 has a determinant of 0, since the elements of each column are a multiple of the first column.

Solving Systems of Linear Equations

R provides the function solve() to solve a system of linear equations. The first argument for function solve() is the matrix of coefficients. The second argument is the solution vector. Here is an example.

Enter the following commands to create the matrix of coefficients A and solution vector b:

```
> A = matrix(c(1.1,1,1,1,1.1,1,1,1,1.1), nrow = 3, ncol = 3, byrow = TRUE)
> b = c(6.1, 6.2, 6.3)
```

Next, view the elements of matrix A and vector b:

```
> A
     [,1] [,2] [,3]
[1,]  1.1  1.0  1.0
[2,]  1.0  1.1  1.0
[3,]  1.0  1.0  1.1

> b
```

```
[1] 6.1 6.2 6.3
```

Now solve the system of linear equations:

```
> x = solve(A,b)
> x
[1] 1 2 3
```

This is the correct solution.

Inverting Matrices

R allows you to use the function solve() with only the matrix of coefficients argument to return the inverse of that matrix. For example:

```
>A.inv = solve(A)
>A.inv

          [,1]      [,2]      [,3]
[1,]  6.774194 -3.225806 -3.225806
[2,] -3.225806  6.774194 -3.225806
[3,] -3.225806 -3.225806  6.774194
```

And then execute the next command to yield an identity matrix (taking in considerations that non-diagonal elements will have very small values instead of zeros):

```
>A.inv %*% A
              [,1]         [,2]          [,3]
[1,]  1.000000e+00 4.440892e-16  5.128276e-16
[2,] -8.881784e-16 1.000000e+00 -6.860831e-16
[3,]  0.000000e+00 0.000000e+00  1.000000e+00
```

You can adjust the above output to look nicer using the round() function, which I introduce later in this tutorial.

Get the Means of Matrix Rows and Columns

R offers the functions rowMeans() and colMeans() to obtain the row means and column means, respectively of a matrix.

To illustrates the rowMeans() and colMeans() functions, first recall matrix X1:

```
> X1
     [,1] [,2] [,3] [,4]
[1,]    1    7   25    6
[2,]    1    1   29   15
[3,]    1   11   56    8
[4,]    1   11   31    8
[5,]    1    7   52    6
```

Now obtain the row and column means:

```
> rowMeans(X1)
[1]  9.75 11.50 19.00 12.75 16.50
```

```
> colMeans(X1)
[1]  1.0  7.4 38.6  8.6
```

To calculate the row sums, you perform the following operation:

```
> ncol(X1) * rowMeans(X1)
[1] 39 46 76 51 66
```

Notice that you use the ncol() function (and NOT the nrow() function) to multiply it with the rowMeans() in order to get the sum of rows.

Likewise, to calculate the column sums, you perform the following operation:

```
> nrow(X1) * colMeans(X1)
[1]   5  37 193  43
```

R offers the functions rowsum() and colsumn(). They don't just simply get the sums. These functions need "grouped" data to work.

Performing Multiple Linear Regression

Using the above matrix operations it is easy to perform multiple linear regression. Consider the data in Table 4.

Table 4. Sample data for multiple regression calculations.

Observation #	X	Y	Z	T (dependent variable)
1	7	25	6	60
2	1	29	15	52
3	11	56	8	20
4	11	31	8	47
5	7	52	6	33

The above table translates into the data matrix X (the first column is ones, needed to calculate the intercept):

```
1       7     25      6
1       1     29     15
1      11     56      8
1      11     31      8
1       7     52      6
```

And the vector Y:

```
60
52
20
47
33
```

To create the matrix X enter the following command that stores elements by column:

```
> X=matrix(c(rep(1,5),7,1,11,11,7,25,29,56,31,52,6,15,8,8,6),nrow=5,ncol=4)
```

To create the vector Y enter the following command:

```
> Y = c(60,52,20,47,33)
```

You can also use the function matrix() to create the column vector Y as a matrix with 5 rows and one column:

```
> Y = matrix(c(60,52,20,47,33), nrow=5, ncol=1)
```

Enter the following commands to create the matrices that will hold the statistical summations:

```
> SumXX = t(X) %*% X
> SumXY = t(X) %*% Y
```

Now solve for the regression coefficients:

```
> MRC = solve(SumXX, SumXY)
> MRC
           [,1]
[1,] 103.447317
[2,]  -1.284097
[3,]  -1.036928
[4,]  -1.339488
```

Where 103.447317 is the intercept, -1.284097 is the slope for variable X, -1.036928 is the slope for variable Y, and -1.339488 is the slope for variable Z. In other words, the multiple regression model is:

$T = 103.447317 - 1.284097\ X - 1.036928\ Y - 1.339488\ Z.$

You can get the regression coefficients using the following single command and bypass intermediate variables:

```
> MRC = solve(t(X) %*% X, t(X) %*% Y)
```

Multi-Dimensional Arrays

R allows you to create arrays with dimensions that exceed those of matrices. The array() command performs this task by specifying the data used to fill in the super-array and the number and size of the dimensions.

For example, to create a three dimensional array with 5 by 3 by 2 elements, filled with the sequence of 1 to 5, you can type:

```
> MDA=array(c(1,2,3,4,5),dim=c(5,3,2))
```

When you type MDA, you get:

```
, , 1

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

, , 2

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

You can even create an array with four dimensions. For example to create an array with 5 by 3 by 2 by 2 elements, filled with the sequence of 1 to 5, you can type:

```
> MDA2=array(c(1,2,3,4,5),dim=c(5,3,2,2))
```

When you type MDA2 you get:

```
> MDA2
, , 1, 1

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

, , 2, 1

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

, , 1, 2

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

```
, , 2, 2

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

The rules for accessing single matrix elements or sub-matrices that you learned earlier also apply to multi-dimensional arrays.

Logical Variables

R supports logical (or Boolean) values, logical variables (single variables, arrays, and matrices), and logical operators. The values TRUE and FALSE represent the Boolean true and false, respectively. When converting logical values to numbers, R treats TRUE and FALSE as 1 and 0, respectively. In order to offer shorthand for TRUE and FALSE, R provides the predefined global variables T and F. These variables are initialized as TRUE and FALSE, respectively.

Logical Operators

Table 4 shows the logical operators in R.

Table 5. Logical operators in R.

Operator	*What it does?*	*Example*
==	Is equal?	1 == 0 is FALSE
!=	Is not equal?	1 == 0 is TRUE
>	Is greater?	1 > 0 is TRUE
>=	Is greater or equal to?	1 >= 0 is TRUE
<	Is less than	1 < 0 is FALSE
<=	Is less than or equal to?	1 <= 0 is FALSE
& or &&	Logical AND	(1 > 0) & (3 == 3) is TRUE
| or ||	Logical OR	(1 > 0) | (2 < 1) is TRUE
!	Logical NOT	!(1 < 0) is TRUE
xor(x, y)	Exclusive OR	xor((1 < 0), (2 > 0)) is TRUE

You can assign a single logical value to a variable by using a logical expression that uses explicit values and scalar variables. When the logical expressions involve vectors or matrices, the result is a vector or matrix of logical values.

Working with Logical Values

Here is an example for the simple case of logical value assignment:

```
> sv1 = 5
> sv2 = 10
> bv = sv1 > sv2
> bv
[1] FALSE
```

Here is an example of using a vector in a logical expression:

```
> Y = c(60,52,20,47,33)
> bvv = Y > 50
> bvv
[1]  TRUE  TRUE FALSE FALSE FALSE
```

The logical expression Y > 50 is applied to each element in vector Y and therefore produces a vector of logical values.

Using Logical Operations to Filter Out Missing Data

Logical vectors are useful in filtering out missing data. R assigns NA, which stands for Not Available, for missing values.

Let's create a vector Vwmd that contains missing data:

```
> Vwmd = c(4, 9, NA, 16, NA, 25, NA)
> Vwmd
[1]  4  9 NA 16 NA 25 NA
```

Now, we create a Boolean vector that scans variable Vwmd and translates valid data into TRUE:

> bvv = Vwmd != "NA"

> bvv

[1] TRUE TRUE   NA TRUE   NA TRUE   NA

Now, suppose you want to calculate the square root for the elements in vector Vwmd such that you filter out those elements with NA. Execute the following command:

```
> sqrt(Vwmd[!is.na(bvv)])
[1] 2 3 4 5
```

The above command uses the logical NOT operator with the Boolean function is.na() to test each element of the logical vector bvv. The logical expression !is.na(bvv) returns a vector of TRUE values for elements of bvv that are NOT NA. This logical expression then filters out those elements in vector Vwmd that have valid values, calculate their square roots, and displays the vector of filtered values:

The following command performs the same task, but without needing the Boolean vector bvv:

```
> sqrt(Vwmd[!is.na(Vwmd!="NA")])
[1] 2 3 4 5
```

Strings

R supports strings, although that support is minimal and to the point.

String Assignment

You can assign a string to a variable, such as:

```
> name1 = "John Smithfield"
```

String Length

To obtain the number of characters in a variable, R offers the nchar() function. To determine the number of characters in the variable name1, you type the following command:

```
> nchar(name1)
[1] 15
```

What happens if you use the length() function with a string? Here is the answer, applied to the variable name1:

```
> length(name1)
[1] 1
```

R considers the entire string as a single element!

String Conversion

R offers the functions as.character(x) and as.numeric(str) to convert between strings and numbers. The following commands tests these two functions:

```
> astr=as.character(12.3)
> astr
[1] "12.3"
> nchar(astr)
[1] 4
> as.numeric(astr)
[1] 12.3
```

You can also use the conversion functions as.real(), as.complex() and as.integer() to convert strings to double-precision numbers, complex numbers, and to integers, respectively.

String Concatenation

The function paste(string1, string1, …, sep=" ") allows you to paste several literal strings and/or the contents of string variables. Here is an example:

```
> paste("The", "rain", "in", "Spain")
[1] "The rain in Spain"
```

The above call to function paste() builds a big string from smaller strings. By default, the function inserts a single space between each two concatenated substrings. You can eliminate this internal padding when you set the parameter sep to be an empty string:

```
> paste("The", "rain", "in", "Spain", sep="")
[1] "TherainінSpain"
```

You can assign a comma to parameter sep and create a string that separates its constituent substrings with commas:

```
> paste("The", "rain", "in", "Spain", sep=",")
[1] "The,rain,in,Spain"
```

Of course you can mix data types, as long as you can convert the non-strings to strings using function as.character():

```
> operand1 = 355
> operand2 = 113
> paste(as.character(operand1), "/", as.character(operand2), "=",
as.character(round(operand1 / operand2, 3)))
[1] "355 / 113 = 3.142"
```

Substring Extraction and Assignment

R offers the substr(str, firstChar, lastChar) function to support extracting a substring AND assigning characters to a substring. The function works a bit like the Visual Basic function MID().

To test the substr() function, first assign the string "1234567890" to the variable ruler, and then extract substring using character indices that are inside and outside the valid range of 1 to 10:

```
> ruler="1234567890"
> substr(ruler, 1, 3)
[1] "123"
> substr(ruler, 3, 5)
[1] "345"
> substr(ruler, 11, 15)
[1] ""
> substr(ruler, 5, 15)
[1] "567890"
```

Notice that out-of-range values are internally rounded to the actual limits of the string. If the range is completely outside the actual character count, the substr() function returns an empty string.

Now create the variable strarr to be an array of 3 strings, each string being "1234567890":

```
> strarr = rep("1234567890", 3)
> strarr
 [1] "1234567890" "1234567890" "1234567890"
```

Replace characters "345" of strarr[1] with "abc", and then type srtarr[1] to view the updated contents of that string:

```
> substr(strarr[1],3,5)<-"abc"
> strarr[1]
[1] "12abc67890"
```

Replace characters "890" of strarr[2] with "abcdef", and then type srtarr[2] to view the updated contents of that string:

```
> substr(strarr[2],8,10)<-"abcdef"
> strarr[2]
[1] "1234567abc"
```

Notice that the assignment DOES NOT expand the string. Instead, the function uses the first three letters "abc" and ignores the additional characters "def".

Replace characters "345" of strarr[3] with "abcdef", and then type srtarr[3] to view the updated contents of that string:

```
> substr(strarr[3],3,5)<-"abcdef"
> strarr[3]
 [1] "12abc67890"
```

Again, the function uses the first three letters "abc" and ignores the additional characters "def", because the call to the function specified overwriting characters 3 to 5.

Splitting Strings

The function strsplit(x, split) allows you to split a string using the split character. Here is an example.

```
> tokens = "1,2,3,4,5"
> strsplit(tokens,",")
[[1]]
[1] "1" "2" "3" "4" "5"
```

The variable tokens stores the numbers 1 to 5, delimited with commas. The strsplit() function splits these numbers based on the comma as delimiter.

Regular Pattern Search and Substitution

The function grep(pattern, x) searches for matches to pattern within x and returns the indices of x where the pattern appears.

Here is an example for using grep()

```
> txt = c("Sunday","Monday","Tuesday","Wednesday")
> i = grep("nday", txt)
> i
[1] 2 3
> txt[i]
```

```
[1] "Sunday"     "Monday"
```

The function gsub(pattern, replacement, x) performs replacement of matches determined by regular expression matching. The function sub() works the same but only replaces the first occurrence.

Here is an example for using function gsub():

```
> gsub("([ab])", "\\1_\\1_", "abc and ABC")
[1] "a_a_b_b_c a_a_nd ABC"
```

The above command scans for the letters a or b in the string "abc and ABC" and duplicates each matching character in the output. The function also includes an underscore character after each duplicated character. The replacement reference "\\1" refers to the matching character and is used in the replacement. The underscore that appears after the reference "\\1" is a literal one.

Type the next two commands as variants of the one above to see the effect of eliminating one or two underscores:

```
> gsub("([ab])", "\\1\\1_", "abc and ABC")
[1] "aa_bb_c aa_nd ABC"
> gsub("([ab])", "\\1\\1", "abc and ABC")
[1] "aabbc aand ABC"
```

Character Case Change

R offers the functions tolower(s) and toupper(s) to convert the character case of a string to lowercase and uppercase, respectively.

Apply the function tolower() to the variable name1, store the resulting string in variable name1.lc, and then type the name of that variable to view its contents:

```
> name1.lc=tolower(name1)
> name1.lc
[1] "john smithfield"
```

Apply the function toupper() to the variable name1, store the resulting string in variable name1.uc, and then type the name of that variable to view its contents:

```
> name1.uc=toupper(name1)
> name1.uc
[1] "JOHN SMITHFIELD"
```

Using the Predefined Character Array letters

R predefines the array **letters** with valid indices in the range of 1 to 26. Each index in that range returns a lowercase character. To obtain uppercase characters you must convert the lowercase character to uppercase using function toupper().

To illustrate using the predefined array letters, type the following commands to display the range of letters from the 10th to the 20th letters:

```
> letters[10:20]
 [1] "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
```

You can use the array letters with the function toupper() to generate random pseudo-words.

Lists

Lists are collection of values that can be of different data types. Think of a list as a structure in C, C++, and Visual Basic.

Creating a List

Here is an example for a list stored in variable L1:

```
> L1 = list(int.val=12, real.val=3.14, str.val = "John", vec.val = 1:10)
```

The list L1 contains the following elements:

- The tag int.val with the integer value of 12.
- The tag real.val with the floating point number of 3.14.
- The tag str.val with the string "John".
- The tag vec.val with the vector 1, 2, 3, …, 9, 10.

The list L1 contains heterogeneous elements.

Accessing List Members

When you type L1 at the command prompt, R displays:

```
$int.val
[1] 12

$real.val
[1] 3.14

$str.val
[1] "John"

$vec.val
 [1]  1  2  3  4  5  6  7  8  9 10
```

The output includes the name of each list tag and its value. The names of the list tags are preceded by a dollar sign.

You can access a specific list element by its positional index. For example, to access the second list element in L1 you type:

```
> L1[2]
```

The output is the name and value in the second list element:

```
$real.val
[1] 3.14
```

You can also use the tag of a list element to index the value in that element. For example, to access the list element that has the tag "str.val" you enter:

```
> L1["str.val"]
```

The output is:

```
$str.val
[1] "John"
```

You can access the value of a list element by using the list$tag syntax. To access the value for the tag str.val in variable L1, you type:

```
> L1$str.val
[1] "John"
```

If a list tag has at least one space as part of its name, then you need to enclose that tag in a pair of double quotes, following the syntax list$"tag name".

You can also access the value associated with an element by using nested square brackets and enclosing the tag in a pair of double quotes. For example, to access the value associated with the tag str.val in variable L1, you type:

```
> L1[["str.val"]]
[1] "John"
```

You can access multiple list elements using an index vector. For example to access the third and fourth elements of variable L1 you type:

```
> L1[3:4]
$str.val
[1] "John"

$vec.val
 [1]  1  2  3  4  5  6  7  8  9 10
```

Accessing List Members Using Abbreviated Tags

R allows you to access list tags by typing enough of a tag name to access the value associated with that tag. Here is an example. Type the following command to create a list:

```
> demo.list = list(one=1, two=2, three=3, four=4)
```

Execute the following set of commands to access various tags in list demo.list using full and abbreviated tag names:

```
> demo.list$one
[1] 1
> demo.list$three
[1] 3
> demo.list$o
[1] 1
> demo.list$f
[1] 4
> demo.list$t
NULL
> demo.list$th
[1] 3
```

Notice that when you type demo.list$o the runtime system succeeds in recognizing that the single letter o refers to tag one, and responds by displaying the value 1. The same is true when you type demo.list$f. R displays 4 which is the value with tag four. Notice that when you type demo.list$t, the single letter t is not enough to distinguish between the tags two and three. R responds by displaying NULL. When you enter demo.list$th, R identified the letters th with the tag three and displays 3.

More List Information

The function length() returns the number of elements in a list. Typing the following commands yields the number of elements in variable L1:

```
> length(L1)
[1] 4
```

You can also access the names of the list tags using the names() function. To access the tags in variable L1 you type:

```
> names(L1)
[1] "int.val"  "real.val" "str.val"  "vec.val"
```

Data Frames

About Data Frames

Now that you are introduced to lists in R, it is worth pointing out that data frames in R are like arrays of lists. Each data frame column can have a distinct data type.

A data frame is a matrix of possible heterogeneous data and with additional information like names for each matrix column (i.e. column variable). See the section *File Input Options*.

Assembling Data Frames from Vectors

In R you typically get a data frame from reading data from a file. R allows you to assemble a data frame from vectors. These vectors may have a different data types, but should have the same number of elements. Here is an example that uses three numeric vectors to create a data frame. Enter the next set of commands to create the vectors x, y, and z, and then employ these vectors to assemble the data frame df:

```
> x=1:10
> y=11:20
> z=21:30
> df=data.frame(x,y,z)
> df
    x  y  z
1   1 11 21
2   2 12 22
3   3 13 23
4   4 14 24
5   5 15 25
6   6 16 26
7   7 17 27
8   8 18 28
9   9 19 29
10 10 20 30
```

Now let's create a data frame that has three numeric vectors and a string vector. Enter the following commands to create the numeric vectors x, y, and z, the string vector s, and then use these vectors to assemble the data frame df2:

```
> x=1:10
> y=11:20
> z=21:30
> s = c("55", "45", "65", "35s", "41", "67", "71B", "11C", "12C", "15C")
> df2 = data.frame(x,y,z,s)
> df2
    x  y  z   s
1   1 11 21  55
2   2 12 22  45
3   3 13 23  65
4   4 14 24  35s
5   5 15 25  41
6   6 16 26  67
7   7 17 27  71B
8   8 18 28 11C
9   9 19 29 12C
10 10 20 30 15C
```

Going back to the three numeric vectors x, y, and z. What happens if we use these vectors to implicitly create a matrix and then convert that matrix into a data frame? Let's use two approaches. The first approach uses the data.frame() function as before. Here is the command that creates the data frame df3 from the three vectors by way of a matrix:

```
> df3=data.frame(matrix(c(x,y,z),nrow=10,ncol=3))
> df3
   X1 X2 X3
1   1 11 21
2   2 12 22
3   3 13 23
4   4 14 24
5   5 15 25
6   6 16 26
7   7 17 27
8   8 18 28
9   9 19 29
10 10 20 30
```

The first command uses the function matrix() to create a matrix from the vectors x, y, and z, The call to function matrix() specifies that the matrix has 10 rows and 3 columns. The function data.frame() converts the matrix into a data frame. The data frame specifies the names of the columns as X1, X2, and X3.

The second approach uses the as.data.frame() conversion function. Here is the command that creates the data frame df4 using the matrix assembled from the vectors x, y, and z:

```
> df4=as.data.frame(matrix(c(x,y,z),nrow=10,ncol=3))
> df4
   V1 V2 V3
1   1 11 21
2   2 12 22
3   3 13 23
4   4 14 24
5   5 15 25
6   6 16 26
7   7 17 27
8   8 18 28
9   9 19 29
10 10 20 30
```

The commands used to create data frames df3 and df4 are very similar (except for the frame-creating functions). Notice that when you inspect the values in data frame df4, you see that the column names are V1, V2, and V3. This naming convention is the same one used by other R functions when reading header-less data from a file.

Assembling Data Frames with Missing Values

The last example for data frame is one that shows missing values. Remember that R uses NA to represent missing values. Enter the following commands to create a new version of vectors x, y, and z which have missing values:

```
> x=c(1:4, NA, NA, 7:10)
> y=c(11:15, rep(NA,3), 19, 20)
> z=c(rep(NA,4),25:30)
> df5=data.frame(x,y,z)
> df5
```

```
    x  y  z
1   1 11 NA
2   2 12 NA
3   3 13 NA
4   4 14 NA
5  NA 15 25
6  NA NA 26
7   7 NA 27
8   8 NA 28
9   9 19 29
10 10 20 30
```

The vectors x, y, and z have missing values at arbitrary indices. The call to function data.frame() assembles the data frame df5 which have several missing values. In fact only the last two rows of data frame df5 have full sets of values!

Editing Variables and Objects

R offers the function objects() to list all of the different variables and objects that currently live in the workspace. Here is an example of what function objects() give as a snapshots of variables in my own system at one given moment:

```
> objects()
 [1] "f"           "L1"          "mat"         "old.wd"      "opar"
 [6] "plot.loglog" "plot.logy"   "reg.plot"    "x"           "xp"
[11] "y"           "y2"          "yp"
```

The output of function objects() gives you an idea of which objects are no longer needed and can be removed using the rm() function.

You can view or edit a variable using the edit(x) or the de(x) function.

Using the edit() Function

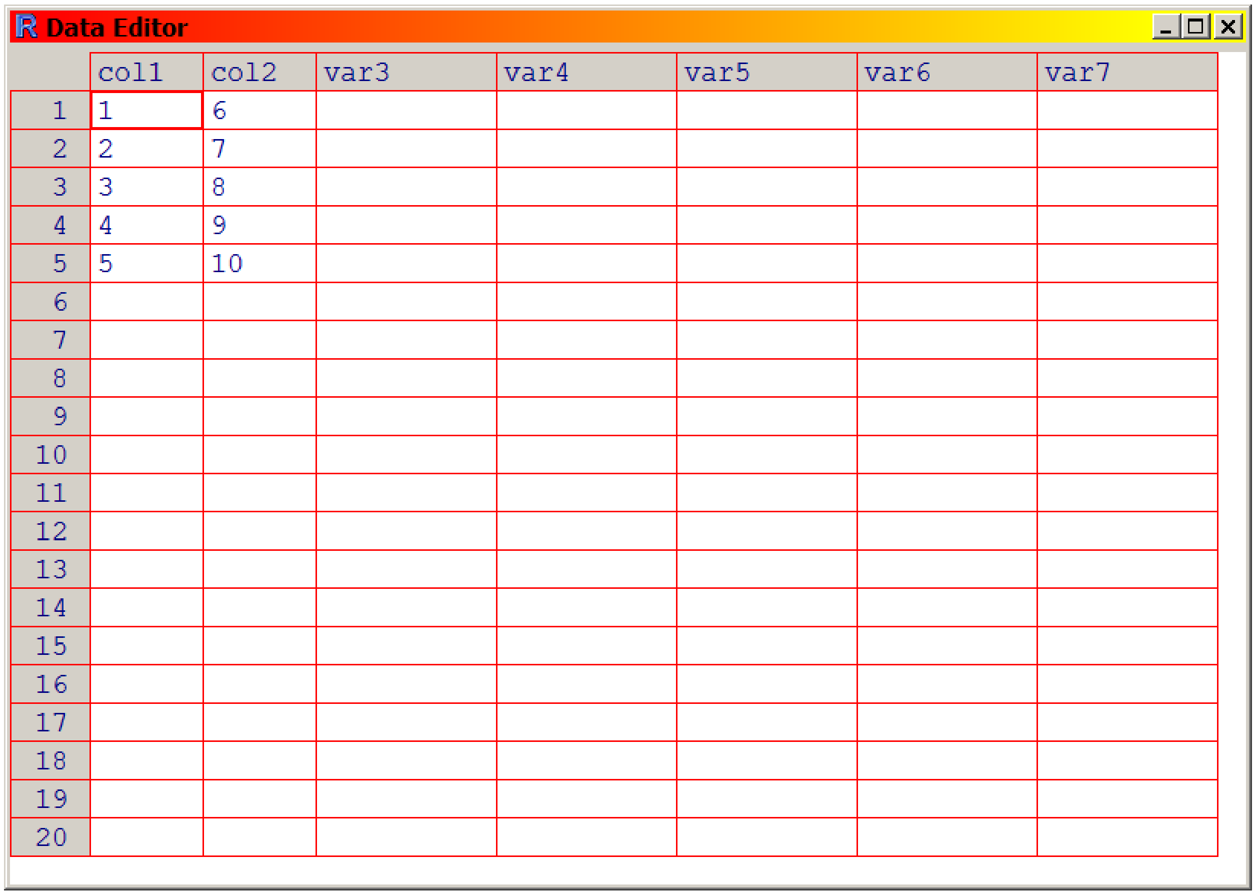
The edit() function uses one of two editor windows depending on the type of object you are editing. The function uses the Data Editor window when editing a matrix. This window displays a grid that faintly resembles a simplified version of an Excel spreadsheet. When editing non-matrix objects, the edit() function uses the R Editor window which looks more like the window of a very simple text editor. The Find and Replace menu commands offer typical dialog box with no extra fanfare.

For example to edit the matrix in variable mat you type the following command:

```
> edit(mat)
```

Causing the function to display the Data Editor window in Figure 1.

Figure 1. Editing a matrix with the Data Editor.



	col1	col2	var3	var4	var5	var6	var7
1	1	6					
2	2	7					
3	3	8					
4	4	9					
5	5	10					
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							

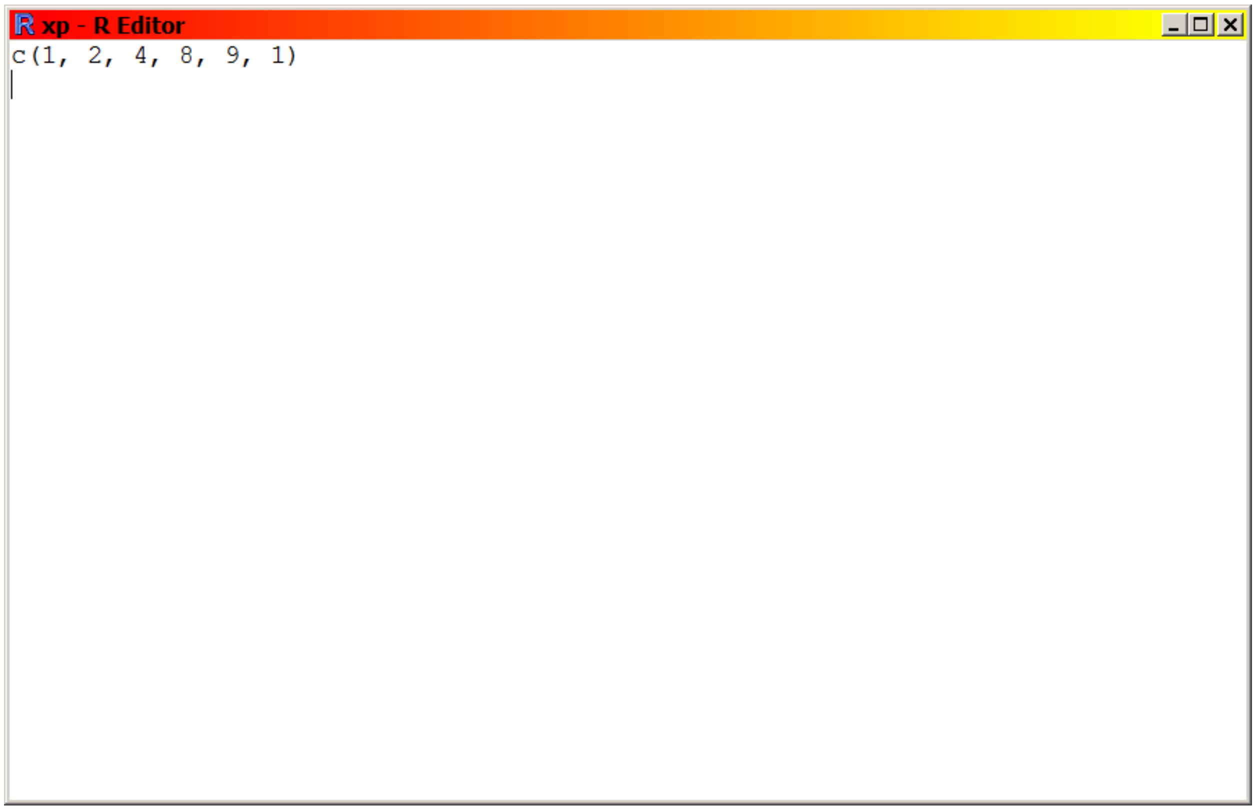
You can view the data and/or edit the existing values, add more data (to new rows and/or new columns), or delete existing values. When you close the Data Editor window, the edit() function lists the current values in variable mat:

```
     col1 col2
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

When you edit a small vector, like the one in variable xp, the function edit() opens the R Editor window with the following data after you execute the next command:

```
> edit(xp)
```

Figure 2. Editing a vector with the R Editor.



```
c(1, 2, 4, 8, 9, 1)
```

The R Editor window in Figure 2 shows the definition for variable xp the way I created it. You can view and edit values by adding, changing, or removing values that appear in the editor window. When you close that window, the function edit() displays the current values in variable xp:

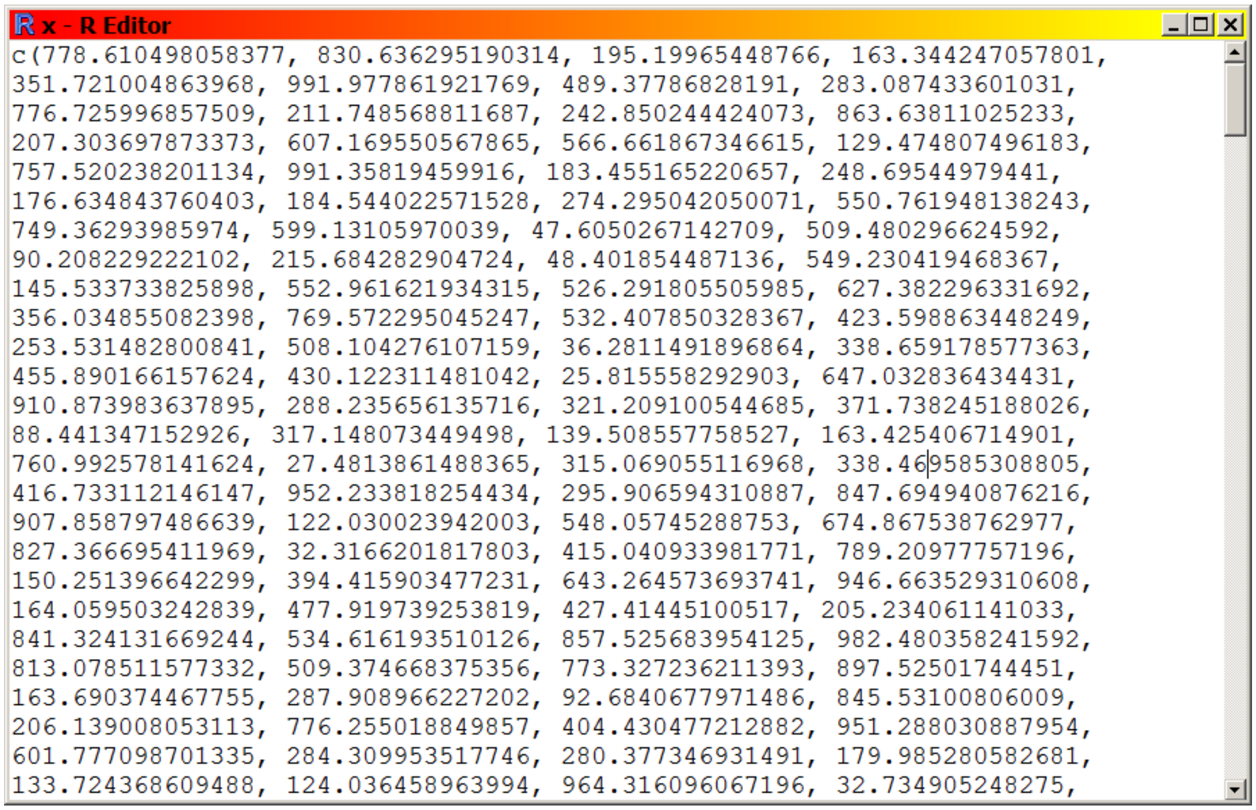
```
[1] 1 2 4 8 9 1
```

In case of editing a vector with numerous values, the R Editor window may well be full and shows the vertical scroll bar to allow you to visit all the vector's values. To illustrate this aspect, edit the variable x, to view its *current* values, by executing the next command:

```
> edit(x)
```

The above command displays the R Editor window in Figure 3 with the following data:

Figure 3. Editing a large vector with the R Editor.



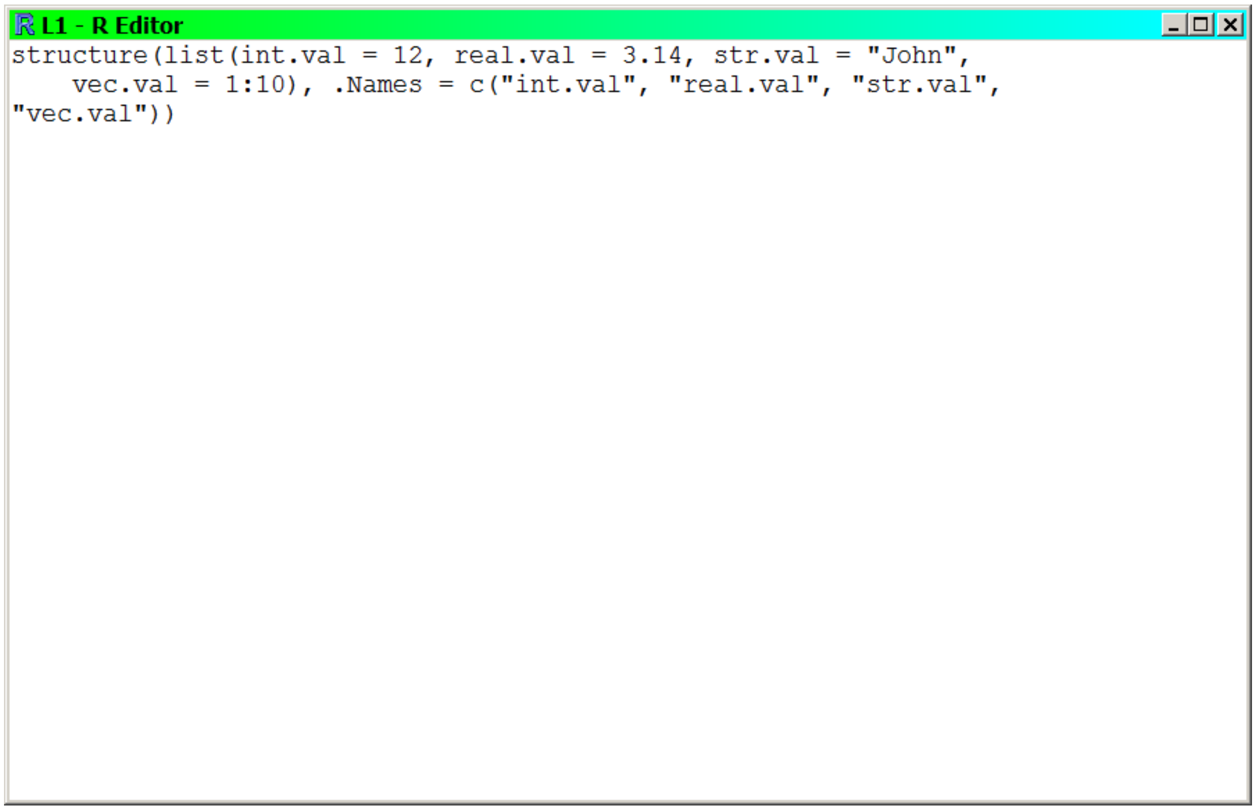
```
c(778.610498058377, 830.636295190314, 195.19965448766, 163.344247057801,
351.721004863968, 991.977861921769, 489.37786828191, 283.087433601031,
776.725996857509, 211.748568811687, 242.850244424073, 863.63811025233,
207.303697873373, 607.169550567865, 566.661867346615, 129.474807496183,
757.520238201134, 991.35819459916, 183.45516522065 7, 248.69544979441,
176.634843760403, 184.544022571528, 274.295042050071, 550.761948138243,
749.36293985974, 599.13105970039, 47.6050267142709, 509.480296624592,
90.208229222102, 215.684282904724, 48.401854487136, 549.230419468367,
145.533733825898, 552.961621934315, 526.291805505985, 627.382296331692,
356.034855082398, 769.572295045247, 532.407850328367, 423.598863448249,
253.531482800841, 508.104276107159, 36.2811491896864, 338.659178577363,
455.890166157624, 430.122311481042, 25.815558292903, 647.032836434431,
910.873983637895, 288.235656135716, 321.209100544685, 371.738245188026,
88.441347152926, 317.148073449498, 139.508557758527, 163.425406714901,
760.992578141624, 27.4813861488365, 315.069055116968, 338.469585308805,
416.73311214614 7, 952.233818254434, 295.906594310887, 847.694940876216,
907.858797486639, 122.030023942003, 548.05745288753, 674.867538762977,
827.366695411969, 32.3166201817803, 415.040933981771, 789.20977757196,
150.251396642299, 394.415903477231, 643.264573693741, 946.663529310608,
164.059503242839, 477.919739253819, 427.41445100517, 205.234061141033,
841.32413166924 4, 534.616193510126, 857.525683954125, 982.480358241592,
813.078511577332, 509.374668375356, 773.327236211393, 897.52501744451,
163.690374467755, 287.908966227202, 92.6840677971486, 845.53100806009,
206.139008053113, 776.255018849857, 404.430477212882, 951.288030887954,
601.777098701335, 284.309953517746, 280.377346931491, 179.985280582681,
133.724368609488, 124.036458963994, 964.316096067196, 32.734905248275,
```

You can examine the values in the R Editor window and scroll up and down to access more values. When you close the R Editor window, the function edit() displays all of the values in variable x. I will not list these values here because the vector x has 1000 numbers! Keep in mind that since I frequently use variable x in many examples, it will have vastly different values. I created the values for variable x shown above as a set of 1000 random numbers ranging between 1 and 1000, using the following command:

```
> x = runif(1000, 1, 1000)
```

When you edit an object like variable L1 which contains a list, the edit() function displays the R Editor window in Figure 4 with the following contents:

Figure 4. Editing a list with the R Editor.



```
structure(list(int.val = 12, real.val = 3.14, str.val = "John",
    vec.val = 1:10), .Names = c("int.val", "real.val", "str.val",
"vec.val"))
```

The R Editor window shows you the details of the list elements (tags and values). You can edit the list and when you close the R Editor window, the function edit() displays the contents of variable L1:

```
$int.val
[1] 12

$real.val
[1] 3.14

$str.val
[1] "John"

$vec.val
 [1]  1  2  3  4  5  6  7  8  9 10
```

Working with the de() Function

In addition to the edit() function, R offers the de(x) function (a short name for function data.entry(x)) which brings up the Data Editor window and displays the value(s) associated with the variable x. This function has some interesting features:

- The function de() works with variables that are scalars, vectors, matrices, and lists.

- First, the Data Editor window appears as long as the type of variable you are editing is valid for function de().
- You can edit any value(s) in the Data Editor. When you close the Data Editor window, the function de() returns the updated contents of the variable you editor. HOWEVER, the function does NOT update the variable that you passed as an argument to function de()! This means that the function de() is good for viewing data OR FOR MAKING MODIFIED COPIES of existing variables. The extent of modification depends on your needs and circumstances. Without assigning the returned value of function de(), that function is good only for viewing data!

A good way to use de() is to create a modified copy of an existing variable and then store the result of the function in another variable. For example:

```
> y = de(x)
```

Allows you to view variable x, edit its values, and then store the modifications in variable y. To edit, store the modified copy, and view the results in one swoop, enclose the entire command in parentheses:

```
> (y = de(x))
```

Console Output Options

Controlling Number of Digits

The powerful function options() allows you to specify the number of digits to display. The general syntax is:

options(digits=number)

To display 5 digits type in:

```
> options(digits=5)
```

And then test that command by performing the following simple calculation:

```
> sqrt(17)
[1] 4.1231
```

Now set the number of digits to 10 and recalculate the square root of 17:

```
> options(digits=10)
> sqrt(17)
[1] 4.123105626
```

You can use the format() function to control the format of a number (as a string). The general syntax for the format() function is:

format(x, digits, nsmall, width).

Where **x** is the value to be formatted, **digits** is the number of digits to include, **nsmall** is the number of decimal places to use, and **width** is the total character string should be.

The round(x, decimals) function rounds the value of the first argument to the specified number of decimals. Here is an example where we round the square root of 2 to 3 decimals:

```
> round(sqrt(2), 3)
[1] 1.414
```

We can also round a previous matrix operation. Recall that the matrix multiplication of A.inv and A yields:

```
> A.inv %*% A
              [,1]         [,2]          [,3]
[1,]  1.000000e+00 4.440892e-16  5.128276e-16
[2,] -8.881784e-16 1.000000e+00 -6.860831e-16
[3,]  0.000000e+00 0.000000e+00  1.000000e+00
```

Now apply the round() function to the above operation with, say 2 decimals, and you get:

```
> round(A.inv %*% A, 2)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

The above results looks more like what you'd expect, since the function rounded the small off-diagonal values to zero.

Displaying Simple Output

Simplest Way!

The examples in this tutorial typically ask you to assign a value to a variable and then type the name of that variable to examine its contents. R allows you to do both in one swoop, simply by enclosing the command in parentheses. For example:

```
>(sqrt2 = sqrt(2))
```

Calculates the square root of 2, stores the result in the variable sqrt2, and displays the square root of 2:

```
[1] 1.414214
```

That is, the R interpreter performs the calculations, stores the result in variable sqrt2, and then displays the contents of that variable. This is the trick I mentioned much earlier in the tutorial. If you missed it then, I am presenting it again here.

You can use the functions show() to show an object. For example, if you type show(MRC) you get the following output:

```
              [,1]
[1,] 103.447316593
[2,]  -1.284096504
[3,]  -1.036927622
[4,]  -1.339487937
```

The function show() is a formal way to display the value of an object. You can get the same results by simply typing the name of the targeted object. So what is function show() really good for? The answer is it is good for use INSIDE user-defined functions! Unlike the command line, placing the name of a variable on a single line does not make a user-defined function display the name of that variable. It is function show() that forces a user-defined function to display data.

Displaying Commented Output

The function cat() allows you to concatenate different results. The general syntax for the cat() function is:

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
```

Where parameter **file** specifies the output file. By default the output goes to the standard output device--the console window. The parameter **sep** is the character that separates each term. Here an example:

```
>cat(355, "/", 113, "=", 355/113, "\n")
[1] 355 / 113 = 3.14159292035398
```

Compare with the above output with the next one:

```
> cat(355, "/", 113, "=", 355/113, "\n", sep="")
[1] 355/113=3.14159292035398
```

The second example specifies that the separator is a null string. Using function cat() with a specific filename allows you to write selected results and values to that file.

Using the Functions sprintf(), gettextf(), and gettext()

R provides the function sprintf(frmt,…) as a wrapper to the C function sprintf(). The parameter **frmt** is the string that represents the format for the output. This function allows you to create a string containing formatted output. The function obeys the same rules as the sprintf() in C. Please consult a the help in R, or a C book, or search the Internet for the formatting rules used by the function sprintf().

Here is an example of using the function sprintf() to display random numbers. Type the following function in a script window. The test function generates the random numbers, store them in variables, and then display them as a table using function sprintf():

```
test.sprintf = function()
{
  rn=runif(10,0,1)
  cat(sprintf(" i    X[i]       Running Mean[1:i]\n"))
  cat("------------------------------\n")
  for (i in 1:length(rn))
    cat(sprintf("%2i %10.8f %10.4f\n", i, rn[i], mean(rn[1:i])))
}
```

Select the function's code by pressing the CTR+A keys and then run the code by pressing the CTRL+R keys to make the R interpreter learn (or memorize, if you like) the function. Then execute the following command at the command line prompt:

```
> test.sprintf()
```

The above command runs the test function which displays results that look like the ones below (remember that when you run the test function you will get different values each time):

```
i    X[i]       Running Mean[1:i]
------------------------------
 1 0.49489709     0.4949
 2 0.73992969     0.6174
 3 0.88525848     0.7067
 4 0.19044839     0.5776
 5 0.65160659     0.5924
 6 0.41233288     0.5624
 7 0.08864863     0.4947
 8 0.23712170     0.4625
 9 0.01636101     0.4130
10 0.70972073     0.4426
```

I chose to run the above example using a function because formatting output within functions makes more sense. If you try to execute the same commands outside a function the output will not be contiguous, since the commands will immediately show results, as shown below (the commands appear in red to make easier to read and distinguish from the output):

```
>   rn=runif(10,0,1)
>    cat(sprintf(" i    X[i]       Running Mean[1:i]\n"))
 i    X[i]       Running Mean[1:i]
>   cat("------------------------------\n")
------------------------------
>   for (i in 1:length(rn))
+     cat(sprintf("%2i %10.8f %10.4f\n", i, rn[i], mean(rn[1:i])))
1 0.49489709     0.4949
 2 0.73992969     0.6174
 3 0.88525848     0.7067
 4 0.19044839     0.5776
 5 0.65160659     0.5924
 6 0.41233288     0.5624
 7 0.08864863     0.4947
 8 0.23712170     0.4625
 9 0.01636101     0.4130
10 0.70972073     0.4426
```

R also offers the function gettext(frmt,…) that works in a very similar manner to function sprintf(). In addition, R offers the function gettext() that returns an array of strings where each string is based on the value of a variable of element of an array/matrix in the arguments for the function gettext(). Here is an example that clarifies how function gettext() works. Type in the following commands to declare variables i1, x1, and y1, and the supply the names of these variables are arguments to function gettext():

```
> i1=123
> x1=355/113
> y1=c(1,2,3)
> gettext(i1, x1, y1)
[1] "123"              "3.14159292035398" "1"                "2"
[5] "3"
```

The above call to function gettext() refers to the variables i1 and x1 as well as the vector y1. The result is an array of strings based on the values in variables i1, x1, and each element in variable y1.

In the case function gettext() encounters a matrix, it returns an array of strings based on the matrix elements visited by rows. Here is an example. Type in the following commands to create matrix m1 and then supply that matrix as an argument for function gettext():

```
> m1=matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, ncol=3)
> m1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> gettext(m1)
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9"
```

Mathematical Functions

R comes with a generous set of mathematical functions. The following table summarizes a selection of these functions. The *Example* column in the next table assumes that the following vectors are already defined:

```
x1 = c(41, 55, 67, 65, 29, 19)
w = c(3, 1, 2, 1, 2, 2)
m1 = matrix(c(1, 5, 2, 8, 7, 4, -1, 3, 9), nrow = 3, ncol = 3)
```

Table 6. List of mathematical and vector functions.

Function	***Purpose***	***Example***
acos	Arccosine function	acos(0.5) gives 1.047198
acosh(x)	Inverse hyperbolic cosine	acosh(2) yields 1.316958
asin	Arcsine function	asin(0.5) returns 0.5235988
asinh(x)	Inverse hyperbolic sine	asinh(2) yields 1.443635
atan, atan2	Arctangent function	atan(1) yields 0.7853982
atanh(x)	Inverse hyperbolic tangent	atanh(0.1) gives 0.1003353
BesselI(n,x)	Modified Bessel function of the first kind	besselI(1,1) returns 0.5651591
BesselJ(n,x)	Bessel function of the first kind	besselJ(1,1) gives 0.4400506
BesselK(n,x)	Modified Bessel function of the first kind	besselK(1,1) returns 0.4400506
BesselY(n,x)	Bessel function of the first kind	besselY(1,1) returns -0.7812128
beta(a,b)	Beta function	beta(2.1, 1.5) gives 0.2495071

cor(x)	Correlation matrix of matrix x	round(cor(m1),2) yields
		`     [,1]  [,2]  [,3]`
		`[1,] 1.00  0.04  0.13`
		`[2,] 0.04  1.00 -0.99`
		`[3,] 0.13 -0.99  1.00`

cos	Cosine function	cos(pi/2) yields 6.123032e-17
cosh(x)	Hyperbolic cosine	cosh(1) yields 1.543081
cumax(x)	Yields the cumulative maximum of the values in x	cumax(x1) returns 41 55 67 67 67 67
cumin(x)	Yields the cumulative minimum of the values in x	cumin(x1) returns 41 41 41 41 29 19
cumprod(x)	Calculates the cumulative product of the values in x	cumprod(x1) yields 41 2255 151085 9820525 284795225 5411109275
cumsum(x)	Calculates the cumulative sum of the values in x	cumsum(x1) gives 41 96 163 228 257 276
diff(x)	The lagged and iterated differences of vector x	diff(x1) yields 14 12 -2 -36 -10
digamma(x)	Di-Gamma function	digamma(2) returns 0.4227843
exp	Exponential	exp(1) gives 2.718282
factorial(x)	Factorial	factorial(5) yields 120
gamma(x)	Gamma function	gamma(2.1) returns 1.046486
lbeta(a,b)	Log Beta function	lbeta(5.3,3.3) gives -5.131163
lfactorial(x)	Log of factorial	lfactorial(5) returns 4.787492
lgamma(x)	Log of Gamma function	lgamma(101) gives 363.7394
log	Natural logarithm	log(2) yields 0.6931472
log(x, baseN)	Calculates the logarithm of x with base baseN	log(10, 2) returns 3.321928
log10	Common logarithm	log10(2) yields 0.30103
max(x)	Maximum of the elements of x	max(x1) returns 67
mean(x)	Calculates the mean of the values in x	mean(x1) returns 46

Function	***Purpose***	***Example***
median(x)	Calculates the median of the values in x	median(x1) yields 48
min(x)	Minimum of the elements of x	min(x1) returns 19
prod(x)	Returns the product of the values in x	prod(x1) gives 5411109275
psigamma(x, deriv = 0)	Psi-Gamma function	psigamma(2.3) yields 0.6000399
quantile(x,probs=)	Yields the sample quantiles corresponding to the given probabilities	quantile(x1,0.95) returns 95% 66.5
range(x)	Returns the minimum and maximum of x	range(x1) yields 18 67
rank(x)	Ranks of the values in x	rank(x1) gives 3 4 6 5 2 1

round(x, n)	Round the values in x to n decimals	round(cor(m1),2) yields
		`     [,1]  [,2]  [,3]`
		`[1,] 1.00  0.04  0.13`
		`[2,] 0.04  1.00 -0.99`
		`[3,] 0.13 -0.99  1.00`

sd(x)	Standard deviation of x	sd(x1) returns 19.62651
sin	Sine function	sin(pi/2) yields 1
sinh(x)	Hyperbolic sine	sinh(1) returns 1.175201
sum(x)	Sum of the values in x	sum(x1) returns 276
tan	Tangent function	tan(pi/4) yields 1
tanh(x)	Hyperbolic tangent	tanh(1) returns 0.7615942
trigamma(x)	Tri-Gamma function	trigamma(2) yields 0.6449341
var(x)	Variance of the values in x, calculated based on n-1 points	var(x1) yields 385.2
weighted.mean(x, w)	Returns the mean of x with weights w	weighted.mean(x1, w) returns 43``

The next table shows the functions that generate random numbers that are distributed according to various probability distribution functions. Since the examples generate random numbers they will vary between each trial.

Table 7. List of random number generating functions.

Function	***Purpose***	***Example***
rbeta(n, shape1, shape2	Generates n values that follow the Beta distribution with a given shape1 and shape2 values.	rbeta(3, 1, 2) generates 0.53107415 0.03205541 0.14892362
rbinom(n, size, prob)	Generates n values that follow the Binomial distribution with a given size and probability values.	rbinom(3, 10, 0.9) returns 10 9 8
rcauchy(n, location=0, scale=1)	Generates n values that follow the Cauchy distribution with a given location and scale values.	rcauchy(3) yields -2.2035069 0.1428067 -1.9328056
rchisq(n, df)	Generates n values that follow	rchisq(3, 10) gives

Function	***Purpose***	***Example***
	the Pearson Chi-square distribution with a given df value.	10.353459 9.538816 10.271604
rexp(n, rate=1)	Generates n values that are exponentially distributed with a given rate.	rexp(3) gives 1.3181335 1.4190392 0.0752405
rf(n, df1, df2)	Generates n values that follow the Fisher-Snedecor distribution with a given df1 and df2 degrees of freedom.	rf(3, 10, 30) yields 1.329699 1.146452 1.215574
rgamma(n, shape, scale=1)	Generates n values that follow the Gamma distribution with a given shape and scale values.	rgamma(3, 2, 2) yields 1.6487988 0.2508304 1.0610589
rgeom(n, prob)	Generates n values that follow the Geometric distribution with a given probability value.	rgeom(3, 0.1) gives 6 4 5
rhyper(nn, m, n, k)	Generates nn values that follow the Hypergeometric distribution with a given m, n, and k values.	rhyper(3, 5, 10, 8) returns 3 2 3
rlnorm(n, meanlog=0, sdlog=1)	Generates n values that are log-normally distributed, having a log mean and log standard deviation.	rlnorm(3) returns 0.6466235 2.4315825 0.3725019
rlogis(n, location=0, scale=1	Generates n values that follow the logistic distribution with a given location and scale values.	rlogis(3) gives 0.5038529 2.0590600 1.2137074
rnbinom(n, size, prob)	Generates n values that follow the negative Binomial distribution with a given size and probability values.	rnbinom(3, 10, 0.1) yields 68 79 82
rnorm(n, mean=0, sd=1)	Generates n values that are normally distributed, having a mean and sd standard deviation	rnorm(3) generates 1.1926073 -0.6970034 0.6974508
rpois(n, lambda)	Generates n values that follow the Poisson distribution with a given lambda value.	rpois(3, 10) returns 9 14 16
rt(n, df)	Generates n values that follow the student-t distribution with a given df degrees of freedom.	rt(3, 30) returns 1.2526477 0.8859305 -2.1098595
runif(n, min=0, max=1)	Returns n values that are uniformly distributed in the range min and max.	runif(3) yields 0.2833825 0.7570550 0.4676397
rweibull(n, shape, scale=1)	Generates n values that follow the Weibull distribution with a given shape and scale values.	rweibull(3, 2, 2) returns 1.0116638 2.2431239 0.7159903

Each of the functions shown above, has three siblings that you can access by replacing the first letter r with the letters d, p, or q in order to obtain the names of the corresponding probability density function, the cumulative probability density function, and the value of the quartile function, respectively.

For example the function rnorm(n, mean, sd) has the sibling functions dnorm(x, mean, sd), pnorm(x, mean, sd), and qnorm(p, mean, sd) that return the functions for the probability density, the cumulative probability density, and the value of the quartile, respectively. Note that the parameter p is a probability that has a value between 0 and 1. The parameter x is single value, vector, or matrix of values.

Handling Errors

The Functions try() and geterrmessage()

R offers the function try() to trap runtime errors that can halt the R interpreter. The declaration for the try() function is:

```
try(expression, silent=FALSE)
```

The parameter **expression** represents any expression and function calls that may raise an error. The Boolean parameter **silent** tells the function whether or not to display an error message. When you set this parameter to TRUE, the try() suppresses (most) error messages, but not warnings. You can retrieve the last error message using function geterrmessage(). This function has no parameters.

Here is a simple example for using the functions try() and geterrmessage(). Execute the following command to perform an offending operation:

```
> if (exists("x")) rm(x)
> y=try(sum(x), silent=TRUE)
```

The first command removes the variable x from the workspace if that variable exists. The second command uses the try() function to catch the possible error in the expression sum(x). The function try() does what you expect to and since the parameter silent is TRUE, the function try() displays no error messages. To view the error message of the last operation, execute the following command:

```
> geterrmessage()
[1] "Error in try(sum(x), silent = TRUE) : object 'x' not found\n"
```

And voila! You get the last error message.

Testing for a Runtime Error

To test for a runtime error, carry out the following steps:

1. Generate a runtime error that is clearly different from the one you a trying to detect. Trap this error using the try() function and assign TRUE to the parameter silent. This step ensures that you overwrite the last error message which might have been generated by an operation just like the one you want to test in step 3.
2. Store the error message of the error you just created in a variable.
3. Trap the operation you want to test the runtime error for using again the try() function. Again, assign TRUE to the silent parameter.
4. Compare the error message you stored in step 2 with the one reported by the function geterrmessage(). If the two messages are different then step 3 has generated a new error.

Here is an example that illustrates the above steps. Execute the following commands to trap the error in the expression sum(dummy):

```
> try(sort(dummy), silent=TRUE)
> lem=geterrmessage()
> try(sum(dummy), silent=TRUE)
> if (lem != geterrmessage()) cat("Error occured\n", geterrmessage(), sep="")
Error occured
Error in try(sum(dummy), silent = TRUE) : object 'dummy' not found
```

The first command generates an error that is different from the one we want to trap. This command calls function try() and traps the expression sort(dummy) which is different from the targeted expression sum(dummy). The second command stores the error message in the variable lem (short for last error message). The third command calls the function try() to trap the targeted expression sum(dummy). The last command determines if the messages in variable lem and the one returned by function geterrmessage() are different, and if so to display the error message. You could have chosen a different kind of action to take, depending on the kind expression and context in which you are executing that expression.

Suppressing Warnings

The parameter less function supressWarnings() suppresses warnings at runtime. So between this function and the function try() you can execute functions such that you suppress warnings and handle errors quietly.

Here is an example for using the function suppressWarnings(). Execute the following commands. The first one causes the interpreter to display a warning while the second one uses the function suppressWarnings() to muffle the same warning:

```
> asin(2)
[1] NaN
Warning message:
In asin(2) : NaNs produced
> suppressWarnings(asin(2))
```

```
[1] NaN
```

Suppressing Both Error and Warning Messages

Armed with the functions try() and suppressWarnings() you can suppress both warning and error messages using the following general form:

result = suppressWarnings(try(expression, silent=TRUE))

By making the call to function try() the argument for function suppressWarnings(), you can quietly evaluate the expression and store the result (whatever the result is in the case of error) in a variable.

Console Input

The readline Function

The R language provides you with the readline(prompt="") function that allows you to prompt for and input a single line of text. You may need to parse the input into smaller substrings. You may also need to convert that input, or its parsed substrings, into numeric values or any other type the input strings are convertible to.

As an example, let's use the readline() function to enter a temperature value and store it in variable x. Your input appears underlined:

```
> x = readline("Enter temperature: ")
Enter temperature: 123
> x
[1] "123"
```

The last output reminds us that the value 123 is stored as the string "123" in variable x. To get a numerical value from the keyboard, you must use the as.numeric() function to convert the string into a number:

```
> x = as.numeric(readline("Enter temperature: "))
Enter temperature: 123
> x
[1] 123
```

The last output confirms that the variable x now stores the numeric value of 123.

The scan Function

R offers the scan() function to enter an array of data from a file. By default the scan function (with no arguments) will attempt to read an array of numbers from the standard input device (typically the keyboard). The function prompts you with a sequence number for your next input.

You enter each number and then press the Enter key. When you are done, just press the Enter key and the scan() function will finish storing your input in the variable you specify.

The simplest way to use the function scan() is to use all the default parameters and not to specify any argument. This makes the scan() function prompt you to enter an array of floating point numbers.

Here is an example. Store the numbers 1 to 5 in the variable x using the scan() function. When you are done, just press Enter when prompted for the 6th value. Then type x to view the contents of the variable. Your input appears as underscored digits and <Enter> refers to pressing the Enter key:

```
> x=scan()
1: 1<Enter>
2: 2<Enter>
3: 3<Enter>
4: 4<Enter>
5: 5<Enter>
6: <Enter>
Read 5 items
> x
[1] 1 2 3 4 5
```

The scan() function has the parameter **what** that tells the function what kind of data to expect. The default is the data type double. The supported types are logical, integer, numeric, complex, character, raw, and list.

Let's redo the last example. This time we specify that the what parameter is assigned the integer(0) argument. Your input appears as underscored digits and <Enter> refers to pressing the Enter key:

```
> x=scan(what=integer(0))
1: 1<Enter>
2: 2<Enter>
3: 3<Enter>
4: 4<Enter>
5: 5<Enter>
6: <Enter>
Read 5 items
> x
[1] 1 2 3 4 5
```

If you enter a non-integer value, the scan() function will stop and display an error message.

You can tell the function scan() how many values you want to input by using the parameter **nmax**. Assign the number of input values to parameter nmax when you call function scan(). Here is an example that tells function scan() to enter 5 values. Your input appears as underscored digits and <Enter> refers to pressing the Enter key:

```
> x=scan(nmax=5)
1: 1<Enter>
2: 2<Enter>
3: 3<Enter>
4: 4<Enter>
5: 5<Enter>
Read 5 items
> x
[1] 1 2 3 4 5
```

Of Files and Directories

Before you learn about file input and output, you should learn about some operations that query and manipulate directories and files.

The directory separators in R are either \\ or /. For example, the reference to the Windows directory in drive C is either C:\\Windows or C:/Windows.

The Working Directory

The functions getwd() and setwd(dir) return and set the working directory, respectively. For example, when I type getwd() on my system I get:

```
> getwd()
[1] "C:/Users/Namir/Documents"
```

I can alter the working directory using the setwd() function. For example, I can set the working directory to be C:\Users\Namir\Documents\R using the following command:

```
> old.wd = getwd()
> setwd("C:/Users/Namir/Documents/R")
[1] "C:/Users/Namir/Documents/R"
```

I can restore the old working directory using the following commands:

```
> setwd(old.wd)
> getwd()
[1] "C:/Users/Namir/Documents"
```

Setting Your Own Working Directory

Now is a good time for you to use the function setwd() to set your own working directory. You will use that directory to store functions that are presented later in this tutorial.

Determine the full path for your own working directory and execute the following command:

```
setwd("Your own path here")
```

For example, to set the working directory for my own R files, I type the following command:

```
> setwd("C:/Users/Namir/Documents/R")
```

If you do this task you save yourself a lot of hassle. When you load functions from R files, in this tutorial, you will not need to specify each time the full path for the directory containing these R files. That's a lot of error-prone typing and frustration that you can avoid!

Listing Files in a Directory

R offers the list.files() and dir() functions to list files in a directory. These functions work in a very similar manner and have an identical list of parameters, so I will focus on the shorter function dir(). The declaration of the function dir() is:

```
dir(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE,
           ignore.case = FALSE)
```

The parameter **path** specifies the directory path to examine. The default value refers to the current path. The parameter **pattern** specifies a file pattern that the function uses to select specific files. The Boolean parameter **all.files** is a flag that when set to TRUE will force all files to appear in the output. Otherwise, the output lists normal files only. The parameter **full.names** is a Boolean flag that when set to TRUE will include the path with each filename listed. Otherwise, the output contains the names of the files only. The Boolean parameter **recursive** is a flag that tells the function to look recursively for files in subdirectories. The parameter **ignore.case** is a Boolean switch when set to TRUE will ignore case sensitivity in the specified pattern. Otherwise, the function will use the specified pattern in a case sensitive manner.

To locate any .txt files in the root directory of drive C, you type the following command (the output is specific to my computer):

```
> dir("C:/", pattern="txt")
 [1] "default.txt"   "MyData1.txt"   "myData2.txt"   "myData3.txt"
"myData3B.txt"
 [6] "myData3C.txt" "myData3D.txt" "myData4.txt"   "myData4B.txt"
"myData4C.txt"
[11] "xla.txt"
```

Creating a Directory

R allows you to create a new directory using the function dir.create (path, showWarnings = TRUE). The parameter **path** specifies the path of the new directory. The Boolean parameter **showWarnings** is a flag that determines whether or not the function displays a warning message if the direction creation hits a snag!

Testing If a File Exists

The file.exists(filename) function determines whether or not the specified file exists. If the file exists, the function yields TRUE. Otherwise, the function returns FALSE. To ensure the function file.exists() works correctly, make sure that you include the correct full or partial path of the targeted file.

To test if file C:\myData1.txt exist, type in the following command:

```
> file.exists("C:/myData1.txt")
[1] TRUE
```

The function returns TRUE to indicate that the file C:\myData1.txt does exist.

Creating a File

The function file.create(filename, showWarnings = TRUE) creates a new file specified by the fully qualified parameter **filename**. If there is an error in creating the file and the Boolean parameter **showWarnings** is TRUE (which is the default value), then the function displays a warning message.

Let's create file C:\test.txt and then test its existence with the function file.exists():

```
> file.create("C:/test.txt")
[1] TRUE
> file.exists("C:/test.txt")
[1] TRUE
```

The function file.create() has successfully created the file C:\test.txt.

Renaming a File

The function file.rename(from, to) renames the file specified by parameter **from** into the filename specified by parameter **to**.

Let's rename the file C:\test.txt into file C:\empty.txt by using the function file.rename(), then use function file.exists() twice—first to make sure that file C:\test.txt no longer exists, and second to make sure that file C:\empty.txt does exist:

```
> file.rename("C:/test.txt", "C:/empty.txt")
[1] TRUE
> file.exists("C:/test.txt")
[1] FALSE
> file.exists("C:/empty.txt")
[1] TRUE
```

The question that comes to mind is "Can function file.rename() move a file by specifying a different directory in the argument of parameter **to**?" The answer is yes it can! To test this feature create the new directory C:\tempo and then move file empty.txt from C:\ to the new directory. Execute the following commands to perform the tasks at hand:

```
> dir.create("C:/tempo")
> file.rename("C:/empty.txt", "C:/tempo/empty.txt")
[1] TRUE
```

The function file.rename() returns TRUE to indicate that the operation was a success. You can use the function file.exists() to double check the success of that operation.

Removing a File

The file.remove(filename) removes the targeted filename. To illustrate using this function, remove the file C:\tempo\empty.txt by executing the following command:

```
> file.remove("C:/tempo/empty.txt")
[1] TRUE
```

The function returns TRUE to confirm the deletion of file empty.txt from the directory C:\tempo. You can use the function file.exists() to double check the success of that operation.

You can also use the function unlink(file) to remove a file or directory. Unlike the file.remove() function, the unlink() function does not echo a value. Here is an example that creates a file and then deletes it. Include additional commands to call function file.exists() to test for the file's existence before and after calling function unlink():

```
> file.create("C:/tempo/testfile.txt")
[1] TRUE
> file.exists("C:/tempo/testfile.txt")
[1] TRUE
> unlink("C:/tempo/testfile.txt")
> file.exists("C:/tempo/testfile.txt")
[1] FALSE
```

Appending to a File

The file.append(file1, file2) appends the file specified by parameter **file2** to the one specified by parameter **file1**.

To illustrate the function file.appemd(), first create File1.txt in the directory C:\tempo and write the following single text line in that file:

```
Line 1 from File 1
(empty line)
```

Make sure that the file has a second empty line. Then create File2.txt in the directory C:\tempo and write the following two lines in that file:

```
Line 1 from File 2
Line 2 from File 2
(empty line)
```

Now append File2.txt to File1.txt using the following command:

```
> file.append("C:/tempo/File1.txt", "C:/tempo/File2.txt")
```

```
[1] TRUE
```

Now open file File1.txt with a text editor. You should see the following lines:

```
Line 1 from File 1
Line 1 from File 2
Line 2 from File 2
```

Copying a File

The function file.copy(from, to, overwrite=FALSE) allows you to duplicate a file. When the parameter **overwrite** is TRUE, the function overwrites the target file. Otherwise, the target file remains intact and the function file.copy() returns FALSE.

Copy file File1.txt into File2.txt using the file.copy() function and set the parameter overwrite to TRUE:

```
> file.copy("C:/tempo/File1.txt", "C:/tempo/File2.txt", overwrite=TRUE)
[1] TRUE
```

When you open file C:\tempo\File2.txt with a text editor you should see the following lines:

```
Line 1 from file 1
Line 1 from File 2
Line 2 from File 2
```

The above lines match the contents of file C:\tempo.File1.txt.

Getting File Information

The file.info(filename) function returns the following information related to the argument for parameter **filename**:

- The size of the file
- Whether or not the file is actually a directory
- The file permission mode as an octal number
- The last modification time
- The creation time
- The last access time
- Whether or not the file is an executable

To illustrate the use of function file.info(), type the following command to obtain the information on file C:\tempo\File1.txt:

```
> file.info("C:/tempo/File1.txt")
                   size isdir mode               mtime               ctime
C:/tempo/File1.txt   58 FALSE  666 2009-09-27 01:10:52 2009-09-27 01:00:12
                                 atime exe
C:/tempo/File1.txt 2009-09-27 01:00:12  no
```

Getting the Base Filename

R supplies the function basename(file) to extract the base filename from a fully qualified filename. The specified file MUST exist. Otherwise the function displays an error message. To illustrate how the function basename() works, type the following command:

```
> basename("C:/regdata/lr1.txt")
[1] "lr1.txt"
```

The argument for the function basename() is the fully qualified filename C:\regdata\lr1.txt (which the example assumes to exist). The function returns the base filename of lr1.txt.

File Input Options

The Scan Function

In the last section, you got an introduction to the scan() function. This function has numerous parameters. Here is the syntax for the function scan() showing the most relevant parameters:

scan(file = "", what = double(0), n = -1, sep = "", skip = 0, quiet = FALSE)

The parameter **file** specifies the input file. The default argument of null refers to the standard input device—typically the keyboard. The parameter **what** specifies the kind of data to enter. . The supported types are logical, integer, numeric, complex, character, raw, and list. The parameter **n** specifies the number of values to obtain from the source file. The default value of -1 means that the number of input values is specified by how many values are in the input file. The parameter **sep** specifies the data delimiter character. By default, there is no delimiter used. When you enter data from an Excel comma-separated values, for example, you need to set the parameter sep to be ",". The parameter **skip** represents the number of lines to skip before you start reading the data. The default value is 0. When you read from a file that has one or more heading lines, then you need to set the skip parameters accordingly. The parameter **quiet** tells the function scan() whether or not it should report the number of input values. The default value is FALSE.

The read Function Family

The R functions library offers a set of similar functions that allow you to read data tables from files. The most relevant file input function is read.table(). The general syntax for this function is:

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown")
```

The following table comments of selected parameters of the read.table() function.

Table 8. The parameters of function read.table().

Parameter	*Comments*	*Default Value*
file	The input file. Must be fully qualified with a valid path.	No default value.
header	A header flag indicating that the first line contains the headers.	FALSE meaning the input file has no headers. TRUE signals that the file has headers in the first line.
sep	The field separator character. Values on each line of the file are separated by this character.	Empty string to signal that white space acts as a separator between values in the input file.
quote	The quoting characters.	Single and double quotes.
dec	The decimal point character.	The dot character.
row.names	A vector of row names.	No default.
col.names	A vector of column names.	No explicit default. However, the function read.table() uses `"V"` followed by the column number as a scheme to construct default column names.
skip	Number of lines to skip before reading data.	Zero is the default causing the input to start with the first line.
nrows	The number of rows to read.	-1 to read as many rows as present in the input file.

You can learn more about the other parameters by typing help(read.table) at the command prompt.

R presents the following input functions that are variants of the function read.table():

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, comment.char="", ...)

read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, comment.char="", ...)

read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, comment.char="", ...)

read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, comment.char="", ...)
```

The function read.csv() reads command-delimited .CSV files created by Excel. The function read.csv2() reads semicolon-delimited .CSV files created by Excel. The function read.delim() reads tab-delimited text files. The function read.delim2() reads tab-delimited text files with numbers having the comma as the decimal characters. All four functions assume, by default, that the input files have headers in the first line. Each read.xxx() function returns a data frame.

Consider the following data which are stored in file C:\MyData1.txt:

```
Xv Yv Zv Tv
7 25 6 60
1 29 15 52
11 56 8 20
11 31 8 47
7 52 6 33
(empty line)
```

Use the function read.table() to read the file C:\MyData1.txt and store the data frame in the variable myData:

```
>myData= read.table("c:/myData1.txt", header = TRUE)
> myData
  Xv Yv Zv Tv
1  7 25  6 60
2  1 29 15 52
3 11 56  8 20
4 11 31  8 47
5  7 52  6 33
(empty line)
```

R allows you to access the columns of the data using the $ access operator and the column's name. The general syntax of dataframe.name$column.name accesses the column named column.name in the data frame dataframe.name. So for example, to access the columns for variables Xv and Tv you type the next two commands:

```
> myData$Xv
[1]  7  1 11 11  7
> myData$Tv
[1] 60 52 20 47 33
```

The function attach(dataframe.name) allows you to directly access the column variables in a data frame (assuming there is no name conflict with an existing global variable). Think of the function attach() as one that *attaches columns variables to the workspace*, making these column variables accessible as regular workspace variables that you explicitly create. For example, if you attach data frame myData, you can then directly access the data in variables Xv and Tv:

```
> attach(myData)
> Xv
[1]  7  1 11 11  7
> Tv
[1] 60 52 20 47 33
```

When you are done with the data frame, use the detach(dataframe.name) function to detach the column variables of the targeted data frame. This closure is a highly recommended step that saves on memory resources and possible conflict with the variables attached to other data frames.

You can also access the columns of a data frame variable by index. For example, to access the first column of data frame myData, you type:

```
> myData[1]
  Xv
1  7
2  1
3 11
4 11
5  7
```

You can also use the name of the column as an index. To access the fourth column which stores the data for variable Tv, you can use that variable's name as an index:

```
> myData["Tv"]
  Tv
1 60
2 52
3 20
4 47
5 33
```

Using the attach() function remains the easiest. However, if the data frame contains variables whose names match those of existing ones in the environment, then using the data frame name with an index is the needed remedy.

In the above results, the output includes the name of the column variable and its data. R is smart enough to distinguish between the column name and the data. That's why you can apply numeric functions to the column variables as though they were ordinary vectors or matrices. For example you can calculate the sum for the Tv column:

```
> sum(Tv)
[1] 212
```

You can also calculate the sum for the values stored in the entire data frame myData:

```
> sum(myData)
[1] 485
```

R treats the column variables as vectors and the data frames as a matrix, while keeping in mind that the data frame has named columns.

The Fixed Width Format File Input

R allows you to read data from fixed width format files using function read.fwf(). These are files where the data appears following a rather strict spacing format. The general syntax for the read.fwf() function is:

```
read.fwf(file, widths, header = FALSE, sep = "\t",
         skip = 0, row.names, col.names, n = -1,
         buffersize = 2000, ...)
```

The parameter **file** specifies the input file. The parameter **widths** is an integer vector that specifies the widths of the fixed-width fields on each line. The parameter can also be a list of integer vectors giving widths for multiline records. The Boolean parameter **header** signals whether the first line of the file contains the names of the variables. When this parameter is TRUE in a call to function read.fwf(), the variable names must be delimited by the character specified by the next parameter **sep**. The parameter **sep** represents the separator character used internally. The parameter **skip** specifies the number of lines in the files to skip. The parameter **row.names** and **col.names** work line with function read.table(). The parameter **n** specifies the maximum number of lines to read. The default value for this parameter imposes no restriction on the number of lines to read. The parameter **buffersize** specifies the number of lines to read at a time.

Let's test the function read.fwf() with the same data that is formatted a bit differently. The first version is the data in file C:\myData4.txt:

```
007025006060
001029015052
011056008020
011031008047
007052006033
(empty line)
```

Note that the above data needs an additional trailing blank line for the function resad.fwf() to work.

The above data has four columns and each column occupies three digits. The format replaces spaces with leading zeros. To read the data from file C:\myData4.txt you need to specify the argument for the parameter widths to be the vector c(3,3,3,3). After you read the table in variable myData2, type that variable's name to view its contents:

```
> myData2 = read.fwf("C:/myData4.txt", c(3,3,3,3))
> myData2
  V1 V2 V3 V4
1  7 25  6 60
2  1 29 15 52
3 11 56  8 20
4 11 31  8 47
5  7 52  6 33
```

The command allows the function read.fwf() to successfully read the columns and assign default names for the column variables.

Now consider the file C:\myData4B.txt which contains the following version of the data:

```
  7 25  6 60
  1 29 15 52
 11 56  8 20
 11 31  8 47
  7 52  6 33
(empty line)
```

The data in file C:\myData4B.txt has four columns of data with each value occupying three characters. The values have spaces between them instead of leading zeros. To read the data from file C:\myData4B.txt you need to specify the argument for the parameter widths to be the vector c(3,3,3,3), just like in the first case. After you read the table in variable myData2, type that variable's name to view its contents:

```
> myData2 = read.fwf("C:/myData4B.txt", c(3,3,3,3))
> myData2
  V1 V2 V3 V4
1  7 25  6 60
2  1 29 15 52
3 11 56  8 20
4 11 31  8 47
5  7 52  6 33
```

Now consider the file C:\myData4C.txt which contains the following version of the data:

```
Xv Yv Zv Tv
  7 25  6 60
  1 29 15 52
 11 56  8 20
 11 31  8 47
  7 52  6 33
(empty line)
```

The above data have column headers. Notice that the first character of the column header MUST be a non-space character to allow function read.fwf() to work properly.

To read the data from file C:\myData4C.txt you need to specify the argument for the parameter widths to be the vector c(3,3,3,3), set the argument for the header to TRUE, and set the argument for the sep to "". After you read the table in variable myData2, type that variable's name to view its contents:

```
> myData2 = read.fwf("C:/myData4C.txt", c(3,3,3,3), header=TRUE, sep="")
> myData2
  Xv Yv Zv Tv
1  7 25  6 60
2  1 29 15 52
3 11 56  8 20
```

```
4 11 31  8 47
5  7 52  6 33
```

The function read.fwf() was able to successfully read the header and data rows.

The load Function

The load(file, workspace) function loads data sets written with the function save() which are discussed in the next section.

The section for the function save() contains the example that shows how to use the function load().

File Output Options

The Sinking Ship!

R supports an interesting kind of output-- one where you save your session (or parts thereof) to an output file. This feature allows you to later examine (and even edit) the output that occurred in a session. The sink(filename) function allows you to save the output from that point on to a file. You can turn off this feature by calling function sink() without any arguments. Of course you can toggle this feature at will and as needed to save relevant parts of your session. The sink() function has an **append** parameter that allows you to append text from one or more sessions. The append feature allows you to copy different parts of a session to the same file.

The save Function

The function save(…, file) allows you to save one or more variables to a binary .RData file.

To illustrate how the save() function works, first recall variables X and Y in the environment:

```
> X
     [,1] [,2] [,3]
[1,]    7   25    6
[2,]    1   29   15
[3,]   11   56    8
[4,]   11   31    8
[5,]    7   52    6
> Y
     [,1]
[1,]   60
[2,]   52
[3,]   20
[4,]   47
[5,]   33
```

Execute the following command to save these variables to the file C:\MyData2.RData:

```
> save(X, Y, file="C:/MyData2.RData")
```

Now remove the variables X and Y from the workspace:

```
> rm(X,Y)
```

Next, attempt to view the contents of variable X:

```
> X
Error: object 'X' not found
```

The error message is expected since you just removed variables X and Y from the workspace. Next, load the binary file C:\MyData2.RData and specify the current workspace (as .GlobalEnv) where the input variables will be restored to:

```
> load("C:/MyData2.RData", .GlobalEnv)
```

Notice that the load() function does not specify the names of the variables, since the input .RData file knows about the names of the variables and their data. Now check to see if you get any data when you type the names of variables X and Y:

```
> X
     [,1] [,2] [,3]
[1,]    7   25    6
[2,]    1   29   15
[3,]   11   56    8
[4,]   11   31    8
[5,]    7   52    6
> Y
     [,1]
[1,]   60
[2,]   52
[3,]   20
[4,]   47
[5,]   33
```

And success! You read the variables back in the workspace environment.

R also provides the save.image() function to save all of the variables in the current workspace:

```
save.image(file = ".RData", version = NULL, ascii = FALSE,
           compress = !ascii, safe = TRUE)
```

Since the function save.image() saves all of the variables in the current workspace, you need not specify the name for of any variable. The load() function will restore all of the variables stored by function save.image().

The write Function

The write() function allows you to write a matrix to a text file. The declaration of the write() function is:

```
write(x, file = "data",
      ncolumns = if(is.character(x)) 1 else 5,
      append = FALSE, sep = " ")
```

The parameter **x** represents the output matrix. The parameter **file** specifies the name of the output file. The parameter **ncolumns** indicates the number of output columns. The Boolean parameter **append** is an append flag. The parameter **sep** specifies the separator between the output values.

To illustrate using the write() function, recall the values in matrix X:

```
> X
     [,1] [,2] [,3]
[1,]    7   25    6
[2,]    1   29   15
[3,]   11   56    8
[4,]   11   31    8
[5,]    7   52    6
```

Now write the data of matrix X to file C:\MyData2.txt:

```
> write(X, "c:/MyData2.txt", nrow(X))
```

When you open the file C:\MyData2.txt with a text editor you view the following data:

```
7 1 11 11 7
25 29 56 31 52
6 15 8 8 6
```

Notice that the write command stores the matrix in a transposed state! You can read back the above data using the read.table() function:

```
> myData2= read.table("c:/myData2.txt", header = FALSE)
> myData2
  V1 V2 V3 V4 V5
1  7  1 11 11  7
2 25 29 56 31 52
3  6 15  8  8  6
> myData2=t(myData2)
> myData2
   [,1] [,2] [,3]
V1    7   25    6
V2    1   29   15
V3   11   56    8
V4   11   31    8
V5    7   52    6
```

The last two commands restore the data in its original orientation by taking the transpose of the input matrix.

The above commands store the input from the file to the data frame myData2. When you type the name of that data frame you see the default column names (since the call to read.table() specified that the argument to parameter header is FALSE) and the data.

The write.table Function

The write.table() function allows you to write a a matrix or data frame to a text file. The declaration of the write.table() function is:

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))
```

The following table comments of selected parameters of the write.table() function.

Table 9. The parameters of function write.table().

Parameter	*Comments*	*Default Value*
x	A matrix or data frame.	No default value.
file	The output filename. Must be fully qualified with a valid path.	No default value.
append	An append flag.	FALSE so that the function overwrites data in any existing file.
quote	A logical value or a numeric vector. When `TRUE`, any character or factor columns are enclosed by double quotes. If a numeric vector, its elements are taken as the indices of those columns to enclose in quotes. In both cases, row and column names are quoted if they are written. When `FALSE`, the function inserts no quotes.	TRUE.
sep	Is the field separator character. Values on each line of the file are separated by this character.	Single space character.
eol	The string to include at the end of each line.	The \n newline character.
na	The string that represents a missing value.	The string "NA".
dec	The decimal point character	The dot character.
row.names	A Boolean value that signals whether the row names of argument `x` are written along with `x`, or a character vector of row names to be included.	TRUE.
col.names	A Boolean value that signals whether the column names of argument `x` are written along with `x`, or a character vector of column names to be included.	TRUE.

In addition, R offers the functions write.cvs() and write.cvs2():

```
write.cvs(x, file = "", append = FALSE, quote = TRUE, sep = ",",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))
write.cvs2(x, file = "", append = FALSE, quote = TRUE, sep = ";",
            eol = "\n", na = "NA", dec = ",", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))
```

The write.cvs() function uses the comma to delimit values in a row. The function write.cvw() uses the semicolon to delimit values in a row and uses the comma as a decimal character.

To illustrate using function write.table(), first consider the following data which are stored in file C:\MyData1.txt:

```
Xv Yv Zv Tv
7 25 6 60
1 29 15 52
11 56 8 20
11 31 8 47
7 52 6 33
(empty line)
```

Use the function read.table() to read the file C:\MyData1.txt and store the data frame in the variable myData:

```
>myData= read.table("c:/myData1.txt", header = TRUE)
> myData
  Xv Yv Zv Tv
1  7 25  6 60
2  1 29 15 52
3 11 56  8 20
4 11 31  8 47
5  7 52  6 33
```

Now use function write.table() to write the data frame in variable myData to file C:\myData3.txt:

```
>write.table(myData,"c:/myData3.txt")
```

When you view file C:\myData3.txt with a text editor you see the following text lines:

```
"Xv" "Yv" "Zv" "Tv"
"1" 7 25 6 60
"2" 1 29 15 52
"3" 11 56 8 20
"4" 11 31 8 47
"5" 7 52 6 33
```

Now write the data in variable myData to file C:\myData3B.txt. This time set the argument for row.names to FALSE:

```
>write.table(myData,"c:/myData3B.txt", row.names=FALSE)
```

When you view file C:\myData3B.txt with a text editor you see the following text lines:

```
"Xv" "Yv" "Zv" "Tv"
7 25 6 60
1 29 15 52
11 56 8 20
11 31 8 47
7 52 6 33
```

Write the data in variable myData to file C:\myData3C.txt, but this time set both arguments for row.names and for col.names to FALSE:

```
> write.table(myData,"c:/myData3C.txt", row.names=FALSE, col.names=FALSE)
```

When you view file C:\myData3C.txt with a text editor you see the following text lines:

```
7 25 6 60
1 29 15 52
11 56 8 20
11 31 8 47
7 52 6 33
```

The above output looks like the contents of the text file written using the write() function.

Finally, write the data in variable myData to file C:\myData3D.txt, but this time set both arguments for parameters row.names and quote to FALSE:

```
> write.table(myData,"c:/myData3D.txt", row.names=FALSE, quote = FALSE)
```

When you view file C:\myData3D.txt with a text editor you see the following text lines:

```
Xv Yv Zv Tv
7 25 6 60
1 29 15 52
11 56 8 20
11 31 8 47
7 52 6 33
```

The above output looks identical to the contents of the file myData1.txt.

Functions

No respectable programming language will avoid supporting user-defined functions. The R language is no exception. You can declare user-defined functions to extend the operations of R by adding new functions that meet your needs.

Working With Functions

There are two general syntaxes for user-defined function. The first syntax is:

name **<- *function* (***parameters***)**

{

statements

return **(***result***)**

}

The second syntax is:

name <- ***function (****parameters****)***

{

statements

expression | *assignment statement*

}

The second form does not use an explicit return statement. Instead, the runtime system returns the value of the *last expression executed* in the function's body as the function's result. The last expression executed may not always be the last statement in the function. In this tutorial I will use the return statement, since it is clearer and more consistent with other common programming languages. In addition, using the return statement makes more sense for functions that have multiple exit points.

Here is an example of a practical function. The function mlr() performs a multiple regression with the independent variable matrix X and the dependent variable column vector Y. The function augments the matrix X by appending a column of ones to the left of matrix X. The function then calculates the regression coefficients using the solve() and t() functions along with the %*% matrix multiplication operator:

```
mlr <- function(X,Y)
{
  nr=nrow(X)
  vect.ones = matrix(rep(1,nr), nrow=nr, ncol=1)
  X1 = cbind(vect.ones, X)
  mlr.coeff = solve(t(X1) %*% X1, t(X1) %*% Y)
  return (mlr.coeff)
}
```

A variation of the above is to have the matrix X store all of the variables. The new version of the function then specifies the index of the column containing the dependent variable. The function handles separating the dependent variable from the independent ones, appending the columns of ones to the independent variables matrix, and then performing the regression calculations using matrix operations.

```
mlr2 <- function(X,Y.index)
{
  Y = X[,Y.index]
  X1 = X[,-Y.index]
  nr=nrow(X1)
  vect.ones = matrix(rep(1,nr), nrow=nr, ncol=1)
  X2 = cbind(vect.ones, X1)
  mrc.coeff = solve(t(X2) %*% X2, t(X2) %*% Y)
  return (mrc.coeff)
}
```

The user-defined functions in R pass the arguments by value. This means that the arguments that are variables do not alter the value of these variables outside the scope of the function, because the functions work with a copy of the data in the arguments.

Loading Functions from Script Files

R allows you to store and load scripts. A script comprises of lines of code that represent R statements that you may well type at the command line, or user-defined functions (which you can also type as the command lines). To save an R script to a file, use the R IDE's File | Save As menu command.

To load a script, use the source(filename) function. This function loads the script from the specified filename. The filename must include the full or partial path that helps the R interpreter find the targeted file if that file is located outside the current working directory.

For example to load the function mlr2(), I first save it to file mlr2.r and then use the following call to source():

```
> source("mlr2.r")
```

When the R interpreter loads a script, which contains a user-defined function, it remembers (or memorizes, if you prefer) that definition. You can then use the user-defined function you just loaded like any other R function, during the current session.

Loading R Statements from Script Files

R does not limit using script files to store and load functions. You can also store R statements in .r script files. When you load these files using the source() function, R loads and executes these files. If the statements you load plot graphs then the R interpreter displays these graphs. To view data calculated in the R statements you load you need to use functions like show() or cat().

Here is an example of code that you may want to store in the file test.r:

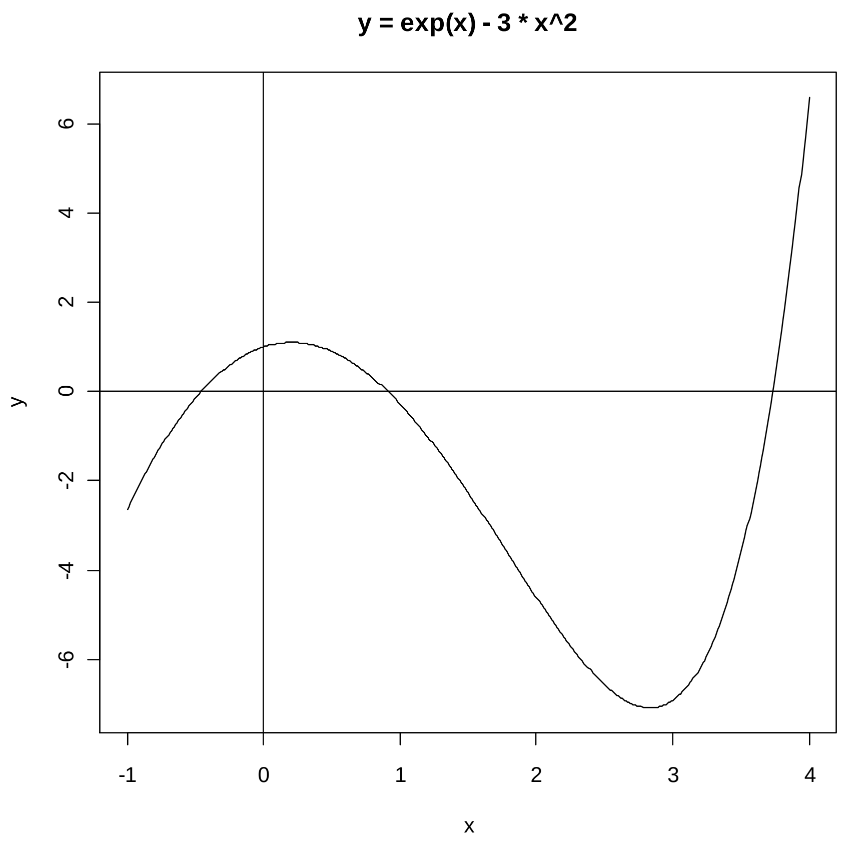
```
x=seq(-1,4,0.01)
y = exp(x)-3*x^2
show(x)
show(y)
plot(x, y, type="l", main="y = exp(x) - 3 * x^2")
abline(h=0)
abline(v=0)
```

After you store the above code in the file test.r, type the following command:

```
> source("test.r")
```

The R interpreter loads that file and executes its statements. The script displays the graph in Figure 5 and also displays the values in vectors x and y, using the function show(). To learn more about the plot() and abline() functions, please consult the *Namir's 102 Plotting Tutorial*.

Figure 5. A sample graph.



Loading Functions and R Statements from Script Files

R allows you to store and load functions and R statements in script files. When you call the function source() to load such files, the R interpreter reads the lines that define the functions, learns these functions, and also reads and executes the non-functions statements. Thus you can write script files that load user-defined functions and then tests or demonstrate them.

Here is an example of a script file that combines a user-defined function and R statements:

```
test = function(a, b, c)
{
  return (a + b + c)
}

val1 = 1
val2 = 2
val3 = 3
sum.val = test(val1, val2, val3)
```

```
show(sum.val)
```

Save the above R lines of code in file test2.r and then load it by executing the following statement:

```
> source("test2.r")
[1] 6
```

The call to function source() loads the script in file test2.r which performs the following tasks:

- Learns the declaration of function test().
- Initializes the variables val1, val2, and val3.
- Adds these values by calling function test() and stores the result in the variable sum.val.
- Displays the sum in the variable sum.val.

Chaining Script Files

R allows you to chain script files. This feature means that when you load a script file using the function source() the loaded script file itself may contain a call to source() to load a second script file, and so on! If planed carefully, chaining scripts can offer a powerful and clever way to assemble code in R.

Here is an example that draws from the last one. Store the following lines in file test3a.r:

```
test = function(a, b, c)
{
  return (a + b + c)
}

val1 = 1
val2 = 2
val3 = 3
source("test3b.r")
```

The above script performs the following tasks:

- Declares function test()
- Initializes the variables val1, val2, and val3.
- Loads the script file test3b.r.

The code for the script file test3b.r appears next:

```
sum.val = test(val1, val2, val3)
show(sum.val)
```

The file test3b.r has two statements—one that calls function test() to add the values in variables val1, val2, and val3, and the other statement display the sum (stored in variable sum.val). If you execute the following command you get the sum of the values in variables val1, val2, and val3:

```
> source("test3a.r")
[1] 6
```

The above example shows the last line in file test3a.r contains the call to function source() to chain load another script file. Such a call to function source() can appear anywhere in a script file, among executable statements, and not just at the end of a script file. To illustrate this feature, I can rewrite the code for files test3a.r and test3b.r as follows. Here is the new version of code for file test3a.r:

```
test = function(a, b, c)
{
  return (a + b + c)
}

source("test3b.r")

sum.val = test(val1, val2, val3)
show(sum.val)
```

And here is the new version of code for file test3b.r:

```
val1 = 1
val2 = 2
val3 = 3
```

In file test3a.r, the call to function source() appears in the middle, but outside the declaration of function test(). In other words, the call to function source() appears in the area of executable R statements.

If you store the above two code segments and execute the following command:

```
> source("test3a.r")
[1] 6
```

You get the same answer as before.

You can also use decision making constructs to determine which script files to chain load and which ones to leave out. You can also dynamically assemble the name of existing script files. The possibilities are endless! There are some limitations however. For example, you cannot place a call to function source() inside the declaration of a user-defined function. Such a call to function source() throws off the R interpreter which seems to make a switch from "learn" mode and into "execute" mode. Such a switch may either generate an error message or, worse yet, yield unpredictable results.

Viewing a Function's Code

To view the code for a function that is accessible in the workplace, just type the name of the function at the command prompt, leaving out the parentheses. The runtime system displays the statements that make up that function.

Comments in the Source Code

You can insert comments in R statements. A comment starts with the # character. The R interpreter ignores the text that appears after the # character. R supports one kind of comment style that runs to the end of the line. A comment can appear on a separate line or after an executable statement. Here is a version of function mlr2() that has comments:

```
mlr2 <- function(X,Y.index)
{
  # select the dependent variable
  Y = X[,Y.index]
  # select the independent variable
  X1 = X[,-Y.index]
  nr=nrow(X1) # get the number of rows
  # create a vector of ones
  vect.ones = matrix(rep(1,nr), nrow=nr, ncol=1)
  # column-bind the vector of ones with matrix X1
  X2 = cbind(vect.ones, X1)
  # calculate the regression coefficients
  mrc.coeff = solve(t(X2) %*% X2, t(X2) %*% Y)
  return (mrc.coeff) # return the regression coefficients vector
}
```

Specifying Default Parameter Values

You have seen that several popular R functions share numerous parameters, most of which have default values assigned to them. Such default values are suitable for typical cases and absolve you from making function calls with a full list of arguments. R allows the user-defined functions to have and use default parameters.

R supports a versatile feature with default parameters that comes in handy when you have multiple default parameters. When you call the function that has multiple default parameters, you can chose to assign a value to any default parameter by stating its name followed by an equal sign and the assigned value. You can do this with one or more default parameters and still use the default values for the other default parameters that you do not explicitly assign an argument to. The next example illustrates this feature.

Here is an example of a function that uses default parameter values. The function calculates the root of another mathematical function using Newton's iterative algorithm:

```
Newton.root <- function(fx, x, toler = 1e-6, max.iters = 50, trace = FALSE)
{
  # initialize iteration counter
  iter = 0
  diff = 10 * toler
  while (abs(diff) > toler) {
    # increment iteration counter
    iter = iter + 1
    # exit if the number of iterations exceed allowable limit
    if (iter > max.iters)
      break
    # calculate increment used to approximate the 1st derivative
    h = 0.01 * (abs(x) + 1)
    fx0 = fx(x)
    # calculate guess refinement
    diff = fx0 * h /(fx(x+h) - fx0)
    # refine the guess for the root
    x = x - diff
    if (trace)
      cat("At iteration", iter, "x =", x, "\n")
  }

  # display warning message if iteration limits are exceeded
  if (iter > max.iters)
    cat("Exceeded iteration limits\n")

  return (c(x, iter))
}
```

The function has the following parameters:

- The parameter **fx** represents the function you want to find the root for.
- The parameter **x** specifies the initial guess for the root.
- The parameter **toler** represents the tolerance of the refined guess. This parameter has the default value of 1e-6.
- The parameter **max.iters** specifies the maximum number of iterations allowed. This parameter has the default value of 100.
- The parameter **trace** that is a Boolean flag which specifies whether or not the function displays a "progress" report during its iteration. This feature is handy especially with difficult cases to solve.

The function returns a vector representing the refined root and the number of iterations used to calculate that refined root. The function uses a while loop and if statements, which are explained in the next section. If you already know how to program, these statements should be familiar.

Save the above function in file Newton.root.r. The targeted function is:

```
fx1 <- function(x)
{
  return (exp(x) - 3*x^2)
}
```

The above function represents:

$$f(x) = e^x - 3x^2$$

Save the targeted function in file fx1.r. Now load the scripts for both functions and test the function Newton.root() using the multiple commands with the following argument sets:

- The arguments fx1 and the initial guess for the root of 4.

- The arguments fx1, the initial guess for the root of -1, and the tolerance of 1e-10.
- The arguments fx1, the initial guess for the root of 1, the tolerance of 1e-20, and maximum number of iterations of 10.
- The arguments fx1, the initial guess for the root of 2.9 (which is located near a minimum value of the function f(x) = $e^x - 3x^2$), and turning on the trace mode by assigning TRUE to the trace parameter.

```
> source("Newton.root.r")
> source("fx1.r")
> Newton.root(fx1, 4)
[1] 3.733079 6.000000
> Newton.root(fx1, -1, 1e-10)
[1] -0.4589623  7.0000000
> Newton.root(fx1, 1, 1e-12, 10)
[1] 0.9100076 8.0000000
> Newton.root(fx1, 2.9, trace=TRUE)
At iteration 1 x = 9.843416
At iteration 2 x = 8.908474
At iteration 3 x = 7.981417
At iteration 4 x = 7.073799
At iteration 5 x = 6.20557
At iteration 6 x = 5.407875
At iteration 7 x = 4.723314
At iteration 8 x = 4.200916
At iteration 9 x = 3.881016
At iteration 10 x = 3.755781
At iteration 11 x = 3.734481
At iteration 12 x = 3.733140
At iteration 13 x = 3.733082
At iteration 14 x = 3.733079
At iteration 15 x = 3.733079
[1]  3.733079 15.000000
```

The function Newton.root() succeeds in finding good approximations to the three roots of the targeted function, each time within a reasonable number of iterations. The first call to function Newton.root() specifies arguments for only the first two parameters. Thus, the function uses the default arguments for the parameters toler, max.iters, and trace.

The second call to function Newton.root() specifies arguments for the first three parameters. Thus, the function uses the default arguments for the parameters max.iters and trace.

The third call to function Newton.root() specifies arguments for the first four parameters and uses the default parameters value for trace.

The fourth call to the function assigns values to the first, second, and fifth parameters. The function uses the default values for the third and fourth parameters. Notice that in the fourth function call, the third argument (trace = TRUE) appears as an assignment to the fifth parameter, selected by name. We did not have to worry about the order of the arguments and how they correspond to the order of the declared parameters. In this example, I chose the initial guess for the root to be 2.9 which is a value near a minima of the function f(x) = $e^x - 3x^2$. This makes the trace mode interesting as we watch the iterations struggle a bit before it starts to zoom in on a root.

Using Abbreviated Names for Default Parameters

R allows you to abbreviate the name of default parameters by typing enough letters to identify these parameters. Here is an example. Consider the following short function:

```
test <- function(one.var=1, two.var=2, three.var =3 , four.var = 4)
{
  return (one.var + two.var + three.var + four.var)
}
```

You can type in the above function at the command line prompt. To illustrate using the abbreviated default parameters names, execute the following command that calls function test():

```
> test(o=10, tw=20, th=30, f=40)
[1] 100
```

The above command specifies a value for each default parameter using their abbreviated names. The single letters o and f are enough to match the parameters one.var and four.var, respectively. In the case of parameters two.var and three.var, the call to function test() uses two letters tw and th to select these default parameters.

What happens if you call function test() and supply the argument of 100 to default parameter two.var by simply typing the first letter of that parameter? Execute the following command to examine the result:

```
> test(t=100)
Error in test(t = 100) : argument 1 matches multiple formal arguments
```

The runtime interpreter complains that the letter t has multiple matches, making it unable to determine whether the argument 100 is meant for parameter two.var or three.var.

☞ Abbreviating default parameter names is a feature that saves on typing at the cost of clarity.

The lack of clarity becomes evident when you read your own code in the future. It is even worse when *someone else* reads your code. If you have a great urge to use abbreviated default parameter names, at least type in about half of the parameter's name, or at least enough for the struggling code reader to identify the parameter.

Recursive Functions

R allows you to write recursive functions. Such functions call themselves and often offer solutions that are more elegant and shorter than non-recursive implementations. A recursive function calls itself for most arguments and has specific arguments for which it returns a value. This action ends the recursion of the function.

The typical example for a recursive function is one that calculates the factorial. Here is the code for function factorial():

```
factorial <- function(n, echo = FALSE)
{
  if (n < 2)
    return (1)
  else {
    if (echo) cat("About to make recursive call for ", n-1, "!\n", sep="")
    return (n * factorial(n-1, echo))
  }
}
```

The function factorial has two parameters—**n** and **echo**. The parameter **n** represents the integer for which we want to calculate the factorial for. The Boolean parameter **echo** is a flag that tells the function whether or not to display messages about the recursive calls. The function uses an if statement to test if the argument for parameter n is less than 2. When that condition is true, the function returns 1 and exits. This action ends the function's recursion. By contrast, when the condition of the if statement is false, the function executes the else clause statements. The first statement in the else clause is a nested if statement that examines the Boolean value of parameter echo. If that parameter is TRUE, the function calls the cat() function to trace the recursive call. The second statement in the else clause makes the recursive call to function factorial() and returns the expression n * factorial(n-1, echo).

Save the function factorial() in file factorial.r, then load it, and use it twice--once with the arguments of 5 and TRUE, and the other time with just the argument of 5 (the function assigns the default parameter value of FALSE to the parameter echo):

```
> source("factorial.r")
> factorial(5, TRUE)
About to make recursive call for 4!
About to make recursive call for 3!
About to make recursive call for 2!
About to make recursive call for 1!
[1] 120
> factorial(5)
[1] 120
```

Both function calls factorial(5,TRUE) and factorial(5) return the correct factorial value of 120. The first call also displays messages that trace the recursive calls made in the else clause found in function factorial().

Nested Functions

R allows you to declare nested functions. In fact, a nested function can have its own nested functions. These nested functions obey the following visibility rule—a nested function is visible only to its immediate parent function.

Nested functions are also found in the programming language Pascal. Many computer scientists criticize the nested function feature as one that causes much runtime overhead.

Here is an example that shows nested functions up to two levels:

```
test <- function(a, b)
{
  # declare a nested function
  power2 <- function(x)
  {

    # declare a nested function
    recip = function(y)
    {
      return (1/y)
    }

    if (x < 1) x = recip(x)
    return (x*x)
   }

   # declare a nested function
   power.half = function(x)
   {
     return (sqrt(x))
   }

  return (power.half(power2(a) + power2(b)))
}
```

The function test(a,b) *basically* returns the square root of the sum of a squared plus b squared. The function test() declares the nested functions power2(x) and power.half(x) to return the square of x and the square root of x, respectively. The nested function power2() declares its own nested function recip(x) to return the reciprocal of x. If you modify the code and try to change the last return statement to something like the following, you can an error at run time, because the function recip(x) is not visible to the function test():

```
return (recip(power.half(power2(a) + power2(b))))
```

If you save the above function, load it into the workspace, and then execute the following command, you get the value 5:

```
> test(3, 4)
 [1] 5
```

Writing to Global Variables

R allows a function to write to global variables. This feature may break the rule of functions having their own variables. Perhaps this feature solves some special difficult programming cases, but I consider it a rogue feature.

To write to a global variable, use the special <<- assignment operator. The variable receiving a value through the <<- operator is a global one. The runtime system creates such a global variable, if needed.

Here is a simple example. Type in the following commands that make sure the variable tv is removed from the global environment, declares a function that writes to variable tv, calls that function, and then inspects the value in variables tv:

```
> if(exists("tv")) rm(tv)
> exists("tv")
[1] FALSE
> test=function(x) { tv <<- x; return (TRUE) }
> test(1)
[1] TRUE
> tv
[1] 1
```

Quietly Closing the R Application

R allows you to exit an R application by typing q() at the command prompt. You can exit from a user-defined function by calling the function q() with the parameter "yes". Such an exit may be convenient after having R perform a long series of calculations. You can have a *master* function invoke other server functions to perform all kinds of tedious and log calculations, save the results to one or more output files, and then finally exit the R application when done. Such a master function might look like:

```
master.fx = function(parameter1, parameter3, …)
{
  server.fx1()
  server.fx2()
  …
  q("yes")
}
```

Recap of Functions

After working with functions let’s recap their features:

- Naming functions follows the same rule as naming variables.
- Functions may have zero or more parameters. Functions with no parameters rely on input data from particular global variables, specific data files, or console input.
- Parameters are not typed.
- Parameters pass data to the function by value.

- The parameters of a function and its local variables are not accessible outside that function. Functions create their own environment. Therefore, calling function objects() inside a function returns a different set of objects than calling objects() from the command line!
- Parameters of a function can be the names of other functions.
- Parameters of a function can have default values.
- Parameters with default values can appear as arguments in a function call in any order (after the list of arguments for non-default parameters appear). Consequently, the call must state the name of the parameter and its argument using the format parameter.name=argument.
- A function can access global variables, unless the function has parameters or local variables that share the same exact name as global variables. In this case, the parameters and local variables overshadow the global variables with the same name.
- A function can write a value to a global variable using the special <<- assignment to global variable operator.
- Functions can be recursive.
- Functions can have nested functions. Even these nested functions can have nested functions of their own. A nested function is only visible to the immediate parent function.
- You can define a function from the command line, load it from a script editor window, or load it from a file using the source() function.
- A function returns one object, which can be a single value, an array, a matrix, and so on. Using an array or a list is convenient way to return multiple values. Using arrays allows you to return results of the same type. Using lists allows you to return results that have different data types.
- The function uses the **return** statement to explicitly return a value. If there is no **return** statement in a function, the runtime system returns the value of the *last statement executed* in the function, right before the function exits.

Program Flow

Decision-making and looping are valuable programming constructs that work best when used as part of the statements in a function. That is why the examples for the various decision-making constructs and loops appear in user-defined functions. Also, it is worth noting that the decision-making constructs and loops are heavily influence by C and C++.

Decision-Making Constructs

The if Statement

The if statement is the simplest decision-making construct. This statement tests a condition and executes one or more statements when that condition is true. The syntax for the if statement is:

if (condition)

statement | *statement block*

✍A statement block has one or more statements enclosed in a pair of open and closed braces.

Here is an example of a function that prompts you to enter the coefficients of a quadratic equation:

$A\ x^2 + B\ x + C = 0$

The function calculates the discriminant value as $B^2 - 4AC$ and uses an if statement to determine whether that discriminant is zero or a positive number. If it is, the function calculates the two roots and returns their values. Rather than leaving you clueless when the discriminate is negative (which leads to imaginary roots), the function uses a second if statement to determine if the discriminant is negative. If it is, the function displays an error message and returns the string "No real solution".

Here is the listing of function quad1.r:

```
quad1 <- function()
{
  cat("Enter coefficients for A x^2 + B x + C\n\n")
  a = as.numeric(readline("Enter value for A: "))
  b = as.numeric(readline("Enter value for B: "))
  c = as.numeric(readline("Enter value for C: "))

  disc = b*b - 4 * a * c

  if (disc >= 0) {
    root1 = (b + sqrt(disc)) / 2 / a
    root2 = (b - sqrt(disc)) / 2 / a
    return (c(root1, root2))
  }

  if (disc < 0) {
    cat("Solution involves imaginary values\n\n")
    return ("No real solution")
  }
}
```

Save the above text in file quad1.r and remember the path used to save it.

Here is a sample session for solving the roots of $x^2 - 5x + 6x = 0$. The first statement loads the function's source code from a script file. Please note that the user's input appears underlined:

```
> source("quad1.r")
> (r=quad1())
Enter coefficients for A x^2 + B x + C
Enter value for A: 1
Enter value for B: -5
Enter value for C: 6
[1] -2 -3
```

Here is a sample session for solving the roots of $x^2 + x + x = 0$, which has imaginary roots:

```
> (r=quad1())
Enter coefficients for A x^2 + B x + C

Enter value for A: 1
Enter value for B: 1
Enter value for C: 1
Solution involves imaginary values

[1] "No real solution"
```

The if-else Statement

The if-else statement tests a condition and allows your program to take alternate courses of action based on whether or not the condition is true.

The syntax for the if-else statement is:

```
if (condition)
        statement | statement block
else
        statement | statement block
```

As an example, here is function quad2() as a new version of function quad1(). The new function replaces the second if statement in quad1(), with an else clause. The program executes the statements in the else clause when the tested condition is false.

```
quad2 <- function()
{
  cat("Enter coefficients for A x^2 + B x + C\n\n")
  a = as.numeric(readline("Enter value for A: "))
  b = as.numeric(readline("Enter value for B: "))
  c = as.numeric(readline("Enter value for C: "))

  disc = b*b - 4 * a * c

  if (disc >= 0) {
    root1 = (b + sqrt(disc)) / 2 / a
    root2 = (b - sqrt(disc)) / 2 / a
    return (c(root1, root2))
  }
  else {
    cat("Solution involves imaginary values\n\n")
    return ("No real solution")
  }
}
```

The functions quad1() and quad2() interact with the user in an identical manner.

The ifelse Function

R offers the ifelse() function to replace simple cases of if-else statements. The general syntax for the ifelse() function is:

ifelse(condition, value when true, value when false)

Here is an example for using the ifelse() function:

```
> ifelse(runif(1) >= 0.5, "It is 0.5 or higher", "It is less than 0.5")
[1] "It is 0.5 or higher"
```

The if-elseif Statement

The if-elseif statement tests a condition and allows your program to examine multiple conditions and to take alternate courses of action based on which tested condition is true.

The syntax for the if-else statement is:

```
if (condition1)
    statement | statement block
else if (condition2)
    statement | statement block
else if (condition3)
    statement | statement block
…
else
    statement | statement block
```

When the if-elseif statement tests N conditions, it can offer N+1 or N (when no else clause appears in the if statement) alternative courses of action. The program executes the clause for which the first condition is true. The program executes the statements in the else clause, which is the catch-all clause, only when all of the tested conditions are false. Using the catch-all else clause is optional, although it is highly recommended as a good programming practice. Eliminating the else clause is a loud statement about the programmer's overconfidence in his programming skills.

Here is a third version of the quadratic-solving function. This version uses the if-elseif statement to solve the cases when the discriminant is positive, zero (leading to identical roots), and negative (leading to imaginary roots). The function yields real or imaginary roots:

```
quad3 <- function()
{
  cat("Enter coefficients for A x^2 + B x + C\n\n")
  a = as.numeric(readline("Enter value for A: "))
  b = as.numeric(readline("Enter value for B: "))
  c = as.numeric(readline("Enter value for C: "))

  disc = b*b - 4 * a * c
  two.a = 2 * a

  if (disc > 0) {
    root1 = (b + sqrt(disc)) / two.a
    root2 = (b - sqrt(disc)) / two.a
    return (c(root1, root2))
  }
  else if (disc < 0) {
    real.part = b / two.a
    imag.part = sqrt(-disc) / two.a
    return (paste(real.part, " +/- ", imag.part, "i\n", sep=""))
  }
  else {
    root1 = b / two.a
    return (c(root1, root1))
  }
}
```

Here is a sample session for solving the roots of $x^2 + x + x = 0$, which has imaginary roots:

```
> source("quad3.r")
> (r=quad3())
Enter coefficients for A x^2 + B x + C

Enter value for A: 1
Enter value for B: 1
Enter value for C: 1
0.5 +/- 0.8660254i
NULL
```

Since R handles imaginary numbers, we can modify the code for function quad3() and let R handle square roots of negative values. If we rewrite the else if clause as:

```
  else if (disc < 0) {
    real.part = b / two.a
    imag.part = sqrt(disc) / two.a
    return (c(real.part + imag.part, real.part - imag.part))
  }
```

You get an error message when the function runs, because R considers the attempt to take the square root of a negative real number an erroneous act! The trick is to guide R's hands into yielding a complex result by converting the value in variable disc into a complex number. This conversion tells R that the argument for the square root function is now a complex number and therefore the result should also be a complex number too. Here is the modified code:

```
  else if (disc < 0) {
    real.part = b / two.a
    imag.part = sqrt(as.complex(disc)) / two.a
    return (c(real.part + imag.part, real.part - imag.part))
  }
```

Replacing the above code snippet in the function quad3() will trigger R's complex math engine and generate the expected results. The modified version of function quad3() interacts with the user the same way as the original version.

The switch Function

R offers a switch() function, and not a switch statement like many programmers might expect. The general syntax for the switch() function is:

switch(index selector, expression1, expression2, ...)

switch(string selector, assignment1,assignment2, ...)

The selector can be an index used to select an expression by its position in the list of expressions. The selector can also be a string that matches the variable in one of the assignments. If the selectors do not have a match in the list of expressions or assignment, the switch() function returns NULL.

Here are a few examples of using the switch() function with an integer selector:

```
> switch(1, "One", "Two", "Three", "Four")
[1] "One"
> switch(3, "One", "Two", "Three", "Four")
[1] "Three"
> switch(4, "One", "Two", "Three", "Four")
[1] "Four"
```

In each call to function switch(), the first argument is an index that returns an expression in the list that follows. When the first argument is 1, the function returns the first expression which is the string "One". When the first argument is 2, the function returns the second expression which is the string "Two". When the first argument is 4, the function returns the fourth expression which is the string "Four".

Here are a few examples of using the function switch() with a string selector:

```
> switch("sum", sum=sum(x), mean=mean(x), sd=sd(x))
[1] 5050
> switch("sd", sum=sum(x), mean=mean(x), sd=sd(x))
[1] 29.01149
> switch("mean", sum=sum(x), mean=mean(x), sd=sd(x))
```

```
[1] 50.5
```

In each call to function switch(), the first argument is an string that returns an expression in the list of assignments that follows. When the first argument is "sum", the function returns the value of expression sum(x) since its pseudo assignment goes to variable sum, which matches the first argument. When the first argument is "sd", the function returns the value of expression sd(x) since its pseudo assignment goes to variable sd, which matches the first argument. When the first argument is "mean", the function returns the value of expression mean(x) since its pseudo assignment goes to variable mean, which matches the first argument.

You can change the assignment variables in a switch statement so that they don't match the expression returned by the switch statements. Here are two examples:

```
> switch("meanOfX", sumOfX=sum(x), meanOfX=mean(x), sdOfX=sd(x))
[1] 50.5
> switch("sdOfX", sumOfX=sum(x), meanOfX=mean(x), sdOfX=sd(x))
[1] 29.01149
```

The new example changes the assignment variables from sum, mean, sd into sumOfX, meanOfX, and sdOfX, respectively. For the switch() function to properly work, it must now select one of the strings "sumOfX", "meanOfX", or "sdOfX".

Loops

The for Loop

The for loop repeats a set of statements for a specified number of times. The general syntax for the for loop is:

***for** (var **in** range)*

statement | statement block

The for loop uses a loop control variable and a range of values. You can create a range of values by simply using the from:to syntax. You can also use the seq() function to create a sequence of numbers that skips values by 2 or more. You can even use the c() function to create a custom array of indices that directs the iterations over arbitrary index values. This approach allows you to create loops that iterate over arbitrary indices—something that is very hard to mimic in other popular languages like Visual Basic, C++, C, and C#, to name a few languages.

Here is a simple example of a function that adds integers from 1 to a user-specified value:

```
sumInts <- function(n)
{
  sum = 0
  for (i in 1:n) {
    sum = sum + i
  }
  return (sum)
}
```

The loop itself increments the loop control variable i (in the range of 1 to n) and add the value of i to the variable sum. The last statement in the function returns the value stored in variable sum.

Save the above function in script sumInts.r, load it, and then test it to calculate the sum of 1 to 1000:

```
> source("sumInts.r")
> (s=sumInts(1000))
[1] 500500
```

The while Loop

The while loop iterates as long as the loop's condition is true. The general syntax for the while loop is:

while *(condition)*

statement | *statement block*

To illustrate the while loop, here is a new version of the integer-adding function that uses the while loop:

```
sumInts2 <- function(n)
{
  sum = n
  while (n > 0) {
    n = n - 1
    sum = sum + n
  }
  return (sum)
}
```

The while loop iterates as long as the value in parameter n is positive. The loop itself decrements the value in variable n and add the new value to the variable sum. The last statement in the function returns the value stored in variable sum.

Save the above function in script sumInts2.r, load it, and then test it to calculate the sum of 1 to 1000

```
> source("sumInts2.r")
> (s=sumInts2(1000))
[1] 500500
```

Exiting from Loops

The R language offers the **break** statement to break out of a loop. Here is an example of a function that uses a while loop with a break statement:

```
sumInts3 <- function(n)
{
  sum = n
  while (1 > 0) {
    n = n - 1
    if (n == 0) break
    sum = sum + n
  }
  return (sum)
}
```

The while loop has a condition that is always true! To avoid being caught in an infinite iteration, the loop uses an if statement which tests whether the value of n is zero. If it is zero, the runtime system executes the break statement which takes the program flow out of the while loop. The loop itself decrements the value in variable n and add the new value to the variable sum. The last statement in the function returns the value stored in variable sum.

Resuming the Next Loop Iteration

The **next** statement allows a loop to skip the rest of the current iteration and to resume at the start of the next one. Here is an example of a function that adds the reciprocals of numbers squared, defined in a range. The range may cover negative and positive numbers. The for loop used in the summation process has an if statement that examines if the loop control variable is zero (a case that arises when the range of numbers extends between negative and positive values). When this condition is true, the code executes the next statement to skip the rest of the current iteration (and avoid dividing by zero) and to resume at the start of the next iteration. The function uses a call to function cat() so you can trace the loop's iteration:

```
sumReciprocals <- function(fromVal, toVal)
{
  sum = 0
  for (i in fromVal:toVal) {
    if (i == 0) next
    sum = sum + 1/i^2
    cat("At i = ", i, " sum = ", sum, "\n")
  }
  return (sum)
}
```

Save the above function in file sumReciprocals.r. When you type the following commands:

```
> source("sumReciprocals.r")
> sumReciprocals(-3, 3)
```

You get the following output:

```
At i =  -3  sum =  0.1111111
At i =  -2  sum =  0.3611111
At i =  -1  sum =  1.361111
At i =  1  sum =  2.361111
At i =  2  sum =  2.611111
At i =  3  sum =  2.722222
[1] 2.722222
```

Notice that the call to function sumReciprocals() specifies the range of integers from -3 to 3—a range that includes zero. The output displays the calls to function cat() for the values between -3 to 3, EXCEPT for the value of zero! Skipping zero avoids the runtime error of dividing by zero.

The repeat Loop

R offers the open **repeat** loop that iterates unconditionally. To use this loop, you need to have at least one if statement with a break statement in order to exit the loop. Since the repeat loop relies on using the break statement to stop iterating (unless you have a mighty good reason to keep on looping forever), I decided to present the loop at the end of the section.

The general syntax for the repeat loop is:

```
repeat {
    …
    if (condition) break
    …
}
```

Here is a version of the integer-adding function that uses the repeat loop:

```
sumInts5 <- function(n)
{
  sum = n
  repeat {
    n = n - 1
    if (n == 0) break
    sum = sum + n
  }
  return (sum)
}
```

The above code shows that the repeat loop has an if statement that tests if the variable n is zero. If that condition is true, the R interpreter executes the break statement which directs the program flow to the first statement after the end of the repeat loop. The loop itself decrements the value in variable n and add the new value to the variable sum. The last statement in the function returns the value stored in variable sum.

Save the above function in file sumInts5.r, load it in the workspace, and then type the following command:

```
> source("sumInts5.r")
> sumInts5(100)
```

```
[1] 5050
```