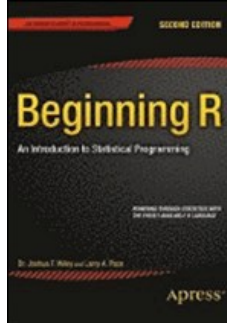


# Chapters *To Go*



## Beginning R: An Introduction to Statistical Programming, Second Edition

by Joshua F. Wiley and Larry A. Pace  
Apress. (c) 2015. Copying Prohibited.

---

Reprinted for jjhemsle jjhemsle, Syracuse University Library

none@books24x7.com

Reprinted with permission as a subscription benefit of **Skillport**,  
<http://skillport.books24x7.com/>

---

All rights reserved. Reproduction and/or distribution in whole or in part in electronic, paper or other forms without written permission is prohibited.



# Chapter 1: Getting Started

## Overview

There are compelling reasons to use R. Enthusiastic users, programmers, and contributors support R and its development. A dedicated core team of R experts maintains the language. R is accurate, produces excellent graphics, has a variety of built-in functions, and is both a functional language and an object-oriented one. There are (literally) thousands of contributed packages available to R users for specialized data analyses.

Developing from a novice into a more competent user of R may take as little as three months by only using R on a part-time basis (disclaimer:  $n = 1$ ). Realistically, depending on background, your development may take days, weeks, months, or even a few years, depending on how often you use R and how quickly you can learn its many intricacies. R users often develop into R programmers who write R functions, and R programmers sometimes want to develop into R contributors, who write packages that help others with their data analysis needs. You can stop anywhere on that journey you like, but if you finish this book and follow good advice, you will be a competent R user who is ready to develop into a serious R programmer if you want to do it. We wish you the best of luck!

## 1.1 What is R, Anyway?

R is an open-source implementation of the S language created and developed at Bell Labs. S is also the basis of the commercial statistics program S-PLUS, but R has eclipsed S-PLUS in popularity. If you do not already have R on your system, the quickest way to get it is to visit the CRAN (Comprehensive R Network Archive) website and download and install the precompiled binary distribution for your operating system. R works on Windows, Mac OS, and Linux systems. If you use Linux, you may already have R with your Linux distribution. Open your terminal and type `$ R --version`. If you do not already have R, the CRAN website is located at the following URL:

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

Download and install the R binaries for your operating system, accepting all the defaults. At this writing, the current version of R is 3.2.0, and in this book, you will see screenshots of R working in both Windows 7 and Windows 8.1. Your authors run on 64-bit operating systems, so you will see that information displayed in the screen captures in this book. Because not everything R does in Unix-based systems can be done in Windows, I often switch to Ubuntu to do those things, but we will discuss only the Windows applications here, and leave you to experiment with Ubuntu or other flavors of Unix. One author runs Ubuntu on the Amazon Cloud, but that is way beyond our current needs.

Go ahead and download Rstudio (current version as of this writing is 0.98.1103) now too, again, accepting all defaults from the following URL:

<http://www.rstudio.com/products/rstudio/download/>

Rstudio is a very forgiving environment for the novice user, and code written in here will work just as well in R itself.

Launch Rstudio and examine the resulting interface. Make sure that you can identify the following parts of the R interface shown in [Figure 1-1](#): the menu bar, the script editing area, the R console, and the R command prompt, which is `>`.

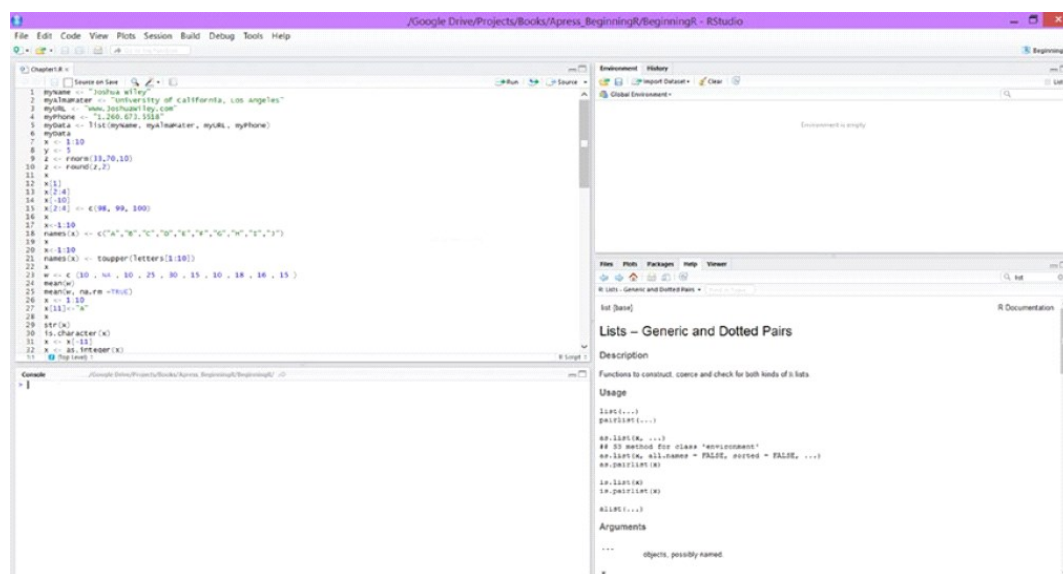


Figure 1-1: The R console running in Rstudio

Before we continue our first R session, let's have a brief discussion of how R works. R is a high-level vectorized computer language and statistical computing environment. You can write your own R code, use R code written by others, and use R packages you write and those written by you or by others. You can use R in batch mode, terminal mode, in the R graphical user interface (RGui), or in Rstudio, which is what we will do in this book. As you learn more about R and how to use it effectively, you will find that you can integrate R with other languages such

as Python or C++, and even with other statistical programs such as SPSS.

In some computer languages, for instance, C++, you have to declare a data type before you assign a value to a new variable, but that is not true in R. In R, you simply assign a value to the object, and you can change the value or the data type by assigning a new one. There are two basic assignment operators in R. The first is `<-`, a left-pointing assignment operator produced by a less than sign followed by a "minus" sign, which is really a hyphen. You can also use an equals sign `=` for assignments in R. I prefer the `<-` assignment operator, and will use it throughout this book.

You must use the `=` sign to assign the parameters in R functions, as you will learn. R is not sensitive to white space the way some languages are, and the readability of R code is benefited from extra spacing and indentation, although these are not mandatory. R is, however, case-sensitive, so to R, the variables `x` and `X` are two different things. There are some reserved names in R, which I will tell you about in Chapter 5.

The best way to learn R is to use R, and there are many books, web-based tutorials, R blog sites, and videos to help you with virtually any question you might have. We will begin with the basics in this book but will quickly progress to the point that you are ready to become a purposeful R programmer, as mentioned earlier.

Let us complete a five-minute session in R, and then delve into more detail about what we did, and what R was doing behind the scenes. The most basic use of R is as a command-line interpreted language. You type a command or statement after the R prompt and then press `<Enter>`, and R attempts to implement the command. If R can do what you are asking, it will do it and return the result in the R console. If R cannot do what you are asking, it will return an error message. Sometimes R will do something but give you warnings, which are messages concerning what you have done and what the impact might be, but that are sometimes warnings that what you did was not what you probably wanted to do. Always remember that R, like any other computer language, cannot think for you.

## 1.2 A First R Session

Okay, let's get started. In the R console, type `<Ctrl> + L` to clear the console in order to have a little more working room. Then type the following, pressing the `<Enter>` key at the end of each command you type. When you get to the personal information, substitute your own data for mine:

```
> 1 + 1
[1] 2
> 1 ^ 1
[1] 1
> 1 * 1
[1] 1
> 1 - 1
[1] 0
> 1 : 10
[1] 1 2 3 4 5 6 7 8 9 10
> (1 : 10) ^ 2
[1] 1 4 9 16 25 36 49 64 81 100
> myName <- "Joshua Wiley"
> myAlmaMater <- "University of California, Los Angeles"
> myURL <- "www.JoshuaWiley.com"
> myPhone <- "1.260.673.5518"
> myData <- list(myName, myAlmaMater, myURL, myPhone)
> myData
[[1]]
[1] "Joshua Wiley"

[[2]]
[1] "University of California, Los Angeles"

[[3]]
[1] "www.JoshuaWiley.com"

[[4]]
[1] "1.260.673.5518"
```

We began with the use of R as a basic calculator. We can create sequences of integers by using the colon operator. Using the exponentiation operator  $(1:10)^2$  gives us the squares of all the numbers in the vector 1 to 10. Observe that when you type a command and press the `<Enter>` key, R will return the result on the next line, prefaced by an index, such as `[1]`. You can assign values to variables without declaring the variable type, as we discussed, so you can just type `myName <- "Joshua Wiley"` to give the variable a value.

This might have seemed a strange way to start, but it shows you some of the things you can enter into your R workspace simply by assigning them. Character strings must be enclosed in quotation marks, and you can use either single or double quotes. Numbers can be assigned as they were with the `myPhone` variable. With the name and address, we created a `list`, which is one of the basic data structures in R. Unlike vectors, lists can contain multiple data types. We also see square brackets `[` and `]`, which are R's way to index the elements of a data object, in this case our list. We can also create vectors, matrices, and data frames in R. Let's see how to save a vector of the numbers from 1 to 10. We will call the vector `x`. We will also create a "constant" called `y`:

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

See that R starts its listing of both `x` and `y` with an index `[ 1 ]`. This is because R does not recognize a scalar value. To R, even a single number is a vector. The object `y` is a vector with one element. The `[ 1 ]` in front of `x` means that the first element of the vector appears at the beginning of the line. Let's make another vector, `z`, containing a sequence of 33 randomly generated numbers from a normal distribution with a mean of 70 and a standard deviation of 10. Because the numbers are random, your `z` vector will not be the same as mine, though if we wanted to, we could set the seed number in R so that we would both get the same vector:

```
> z <- rnorm( 33 , 70 , 10 )
> z <- round( z , 2 )
> z
[ 1 ] 81.56 70.85 77.48 64.02 68.94 80.24 60.84
70.93 75.21 75.05 52.17 52.29
[ 13 ] 70.20 79.29 84.75 64.88 73.74 71.19 61.01
63.43 55.74 71.54 69.71 82.52
[ 25 ] 73.40 75.39 79.28 80.36 65.79 73.15 75.41
69.56 85.87
```

When R must wrap to a new line in the console to print additional output, it shows the index of the first element of each line.

To see a list of all the objects in your R session, type the command `ls()`:

```
> ls()
[1] "myAlmaMater" "myData" "myName" "myPhone" "myURL" "x" "y" "z"
```

To see the current working directory, type the command `getwd()`. You can change the working directory by typing `setwd()`, but I usually find it easier to use the `File > Change dir...` and navigate to the directory you want to become the new working directory. As you can see from the code listing here, the authors prefer working in the cloud. This allows us to gain access to our files from any Internet-connected computer, tablet, or smartphone. Similarly, our R session is saved to the cloud, allowing access from any of several computers at home or office computers.

```
> getwd()
[1] "C:/Users/Joshua Wiley/Google Drive/Projects/Books/Apress_BeginningR/BeginningR"
```

In addition to `ls()`, another helpful function is `dir()`, which will give you a list of the files in your current working directory.

To quit your R session, simply type `q()` at the command prompt, or if you like to use the mouse, select `File > Exit` or simply close Rstudio by clicking on the X in the upper right corner. In any of these cases, you will be prompted to save your R workspace.

Go ahead and quit the current R session, and save your workspace when prompted. We will come back to the same session in a few minutes. What was going on in the background while we played with R was that R was recording everything you typed in the console and everything it wrote back to the console. This is saved in an R history file. When you save your R session in an `RData` file, it contains this particular workspace. When you find that file and open it, your previous workspace will be restored. This will keep you from having to reenter your variables, data, and functions.

Before we go back to our R session, let's see how to use R for some mathematical operators and functions (see [Table 1-1](#)). These operators are *vectorized*, so they will apply to either single numbers or vectors with more than one number, as we will discuss in more detail later in this chapter. According to the R documentation, these are "unary and binary generic functions" that operate on numeric and complex vectors, or vectors that can be coerced to numbers. For example, logical vectors of `TRUE` and `FALSE` are coerced to integer vectors, with `TRUE = 1` and `FALSE = 0`.

**Table 1-1: R's mathematical operators and functions**

Operator/Function	R Expression	Code Example
Addition	+	2 + 2
Subtraction	-	3 - 2
Multiplication	*	2 * 5
Division	/	4 / 2
Exponent	^	3 ^ 2
Square Root	sqrt( )	sqrt(81)
Natural Logarithm	log( )	log(10)
Modulus	% %	x % % y
Absolute Value	abs( )	abs(-3)

Table 1-2 shows R's comparison operators. Each of these evaluates to a logical result of `TRUE` or `FALSE`. We can abbreviate `TRUE` and `FALSE` as `T` and `F`, so it would be unwise to name a variable `T` or `F`, although R will let you do that. Note that the equality operator `==` is different from the `=` used as an assignment operator. As with the mathematical operators and the logical operators (see Chapter 4), these are also vectorized.

Table 1-2: Comparison operators in R

Operator	R Expression	Code Example
Equality	<code>==</code>	<code>x == 3</code>
Inequality	<code>!=</code>	<code>x != 4</code>
Greater than	<code>&gt;</code>	<code>5 &gt; 3</code>
Less than	<code>&lt;</code>	<code>3 &lt; 5</code>
Greater than or equal to	<code>&gt;=</code>	<code>3 &gt;= 1</code>
Less than or equal to	<code>&lt;=</code>	<code>3 &lt;= 3</code>

R has six "atomic" vector types (meaning that they cannot be broken down any further), including `logical`, `integer`, `real`, `complex`, `string` (or character), and `raw`. Vectors must contain only one type of data, but lists can contain any combination of data types. A `data frame` is a special kind of list and the most common data object for statistical analysis. Like any list, a data frame can contain both numerical and character information. Some character information can be used for `factors`. Working with factors can be a bit tricky because they are "like" vectors to some extent, but they are not exactly vectors.

My friends who are programmers who dabble in statistics think factors are evil, while statisticians like me who dabble in programming love the fact that character strings can be used as factors in R, because such factors communicate group membership directly rather than indirectly. It makes more sense to have a column in a data frame labeled `sex` with two entries, `male` and `female`, than it does to have a column labeled `sex` with 0s and 1s in the data frame. If you like using 1s and 0s for factors, then use a scheme such as labeling the column `female` and entering a 1 for a woman and 0 for a man. That way the 1 conveys meaning, as does the 0. Note that some statistical software programs such as SPSS do not uniformly support the use of strings as factors, whereas others, for example, Minitab, do.

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

1.3 Your Second R Session

Reopen your saved R session by navigating to the saved workspace and launching it in R. We will put R through some more paces now that you have a better understanding of its data types and its operators, functions, and "constants." If you did not save the session previously, you can just start over and type in the missing information again. You will not need the list with your name and data, but you will need the `x`, `y`, and `z` variables we created earlier.

As you have learned, R treats a single number as a vector of length 1. If you create a vector of two or more objects, the vector must contain only a single data type. If you try to make a vector with multiple data types, R will coerce the vector into a single type.

1.3.1 Working with Indexes

R's indexing is quite flexible. We can use it to add elements to a vector, to substitute new values for old ones, and to delete elements of the vector. We can also subset a vector by using a range of indexes. As an example, let's return to our `x` vector and make some adjustments:

```
> x
[ 1 ] 1 2 3 4 5 6 7 8 9 10
> x [ 1 ]
[ 1 ] 1
> x [ 2 : 4 ]
[ 1 ] 2 3 4
> x [ - 1 0 ]
[ 1 ] 1 2 3 4 5 6 7 8 9
> x
[ 1 ] 1 2 3 4 5 6 7 8 9 10
> x [ 2 : 4 ] <- c ( 9 8 , 99 , 1 0 0 )
> x
[ 1 ] 1 9 8 9 9 1 0 0 5 6 7 8 9 1 0
```

Note that if you simply ask for subsets, the `x` vector is not changed, but if you reassign the subset or modified vector, the changes are saved. Observe that the negative index removes the selected element or elements from the vector but only changes the vector if you reassign the new vector to `x`. We can, if we choose, give names to the elements of a vector, as this example shows:

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

```
> x
[ 1 ] 1 2 3 4 5 6 7 8 9 10
> names(x) <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J")
> x
A B C D E F G H I J
1 2 3 4 5 6 7 8 9 10
```

This showcases the difference between thinking as a user versus thinking as a programmer! R has a variety of built-in functions that automate even the simplest kind of operations. You just saw me waste our time by typing in the letters A through J. R already knows the alphabet, and all you have to do is tell R you want the first 10 letters. The more you know about R, the easier it is to work with, because it keeps you from having to do a great deal of repetition in your programming. Take a look at what happens when we ask R for the letters of the alphabet and use the power of built-in character manipulation functions to make something a reproducible snippet of code. Everyone starts as an R user and (ideally) becomes an R programmer, as discussed in the introduction:

```
> x
[ 1 ] 1 2 3 4 5 6 7 8 9 10
> names(x) <- toupper(letters[1:10])
> x
A B C D E F G H I J
1 2 3 4 5 6 7 8 9 10
```

The `toupper` function coerces the letters to uppercase, and the `letters[1:10]` subset gives us A through J. Always think like a programmer rather than a user. If you wonder if something is possible, someone else has probably thought the same thing. Over two million people are using R right now, and many of those people write R functions and code that automates the things that we use on such a regular basis that we usually don't even have to wonder whether but simply need to ask where they are and how to use them. You can find many examples of efficient R code on the web, and the discussions on StackExchange are very helpful.

If you are trying to figure something out that you don't know how to do, don't waste much time experimenting. Use a web search engine, and you are very likely to find that someone else has already found the solution, and has posted a helpful example you can use or modify for your own problem. The R manual is also helpful, but only if you already have a strong programming background. Otherwise, it reads pretty much like a technical manual on your new toaster written in a foreign language.

It is better to develop good habits in the beginning than it is to develop bad habits and then having to break them first before you can learn good ones. This is what Dr. Lynda McCalman calls a BFO. That means a blinding flash of the obvious. I have had many of those in my experience with R.

### 1.3.2 Representing Missing Data in R

Now let's see how R handles missing data. Create a simple vector using the `c()` function (some people say it means *combine*, while others say it means *concatenate*). I prefer *combine* because there is also a `cat()` function for concatenating output. For now, just type in the following and observe the results. The built-in function for the mean returns NA because of the missing data value. The `na.rm = TRUE` argument does not remove the missing value but simply omits it from the calculations. Not every built-in function includes the `na.rm` option, but it is something you can program into your own functions if you like. We will discuss functional programming in Chapter 5, in which I will show you how to create your own custom function to handle missing data. We will add a missing value by entering NA as an element of our vector. NA is a legitimate logical character, so R will allow you to add it to a numeric vector:

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

Observe that the mean is calculated when you omit the missing value, but unless you were to use the command `w <- w[-2]`, the vector will not change.

### 1.3.3 Vectors and Vectorization in R

Remember vectors must contain data elements of the same type. To demonstrate this, let us make a vector of 10 numbers, and then add a character element to the vector. R coerces the data to a character vector because we added a character object to it. I used the index `[11]` to add the character element to the vector. But the vector now contains characters and you cannot do math on it. You can use a negative index, `[-11]`, to remove the character and the R function `as.integer()` to coerce the vector back to integers.

To determine the structure of a data object in R, you can use the `str()` function. You can also check to see if our modified vector is integer again, which it is:

```
> x <- 1:10
> x[11] <- "A"
> x
[ 1 ] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
" A "
```



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

Add  $y$  to  $x$  as follows. See that R recycles  $y$  for each value of  $x$ , so that the addition operation results in a new vector. No explicit looping was required:

```
> x + y
[1] 6 7 8 9 10 11 12 13 14 15
```

The way vectorization works when you use operations with two vectors of unequal length is that the shorter vector is *recycled*. If the larger vector's length is a multiple of the length of the shorter vector, this will produce the expected result. When the length of the longer vector is not an exact multiple of the shorter vector's length, the shorter vector is recycled until R reaches the end of the longer vector. This can produce unusual results. For example, divide  $z$  by  $x$ . Remember that  $z$  has 33 elements and  $x$  has 10:

```
> z
[1] 81.56 70.85 77.48 64.02 68.94 80.24 60.84
70.93 75.21 75.05 52.17 52.29 [13] 70.20 79.29
84.75 64.88 73.74 71.19 61.01 63.43 55.74 71.54
69.71 82.52 [25] 73.40 75.39 79.28 80.36 65.79
73.15 75.41 69.56 85.87
> x
[1] 1 2 3 4 5 6 7 8 9 10
> round(z/x, 2)
[1] 81.56 35.42 25.83 16.00 13.79 13.37 8.69
8.87 8.36 7.50 52.17 26.14 [13] 23.40 19.82 16.95
10.81 10.53 8.90 6.78 6.34 55.74 35.77 23.24
20.63 [25] 14.68 12.56 11.33 10.04 7.31 7.32
Warning message:
In z/x : longer object length is not a multiple of shorter object length
```

R recycled the  $x$  vector three times, and then divided the last three elements of  $z$  by 1, 2, and 3, respectively. Although R gave us a warning, it still performed the requested operation.

### 1.3.4 A Brief Introduction to Matrices

Matrices are vectors with dimensions. We can build matrices from vectors by using the `cbind()` or `rbind()` functions. Matrices have rows and columns, so we have two indexes for each cell of the matrix. Let's discuss matrices briefly before we create our first matrix and do some matrix manipulations with it.

A *matrix* is an  $m \times n$  (row by column) rectangle of numbers. When  $n = m$ , the matrix is said to be "square." Square matrices can be *symmetric* or *asymmetric*. The *diagonal* of a square matrix is the set of elements going from the upper left corner to the lower right corner of the matrix. If the off-diagonal elements of a square matrix are the same above and below the diagonal, as in a correlation matrix, the square matrix is *symmetric*.

A *vector* (or *array*) is a 1-by- $n$  or an  $n$ -by-1 matrix, but not so in R, as you will soon see. In statistics, we most often work with symmetric square matrices such as correlation and variance-covariance matrices. An entire matrix is represented by a boldface letter, such as  $A$ :

$$A_{m,n} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

Matrix manipulations are quite easy in R. If you have studied matrix algebra, the following examples will make more sense to you, but if you have not, you can learn enough from these examples and your own self-study to get up to speed quickly should your work require matrices.

Some of the most common matrix manipulations are *transposition*, *addition* and *subtraction*, and *multiplication*. Matrix multiplication is the most important operation for statistics. We can also find the *determinant* of a square matrix, and the *inverse* of a square matrix with a nonzero determinant.

You may have noticed that I did not mention division. In matrix algebra, we write the following, where  $B^{-1}$  is the inverse of  $B$ . This is the matrix algebraic analog of division (if you talk to a mathematician, s/he would tell you this is how regular 'division' works as well. My best advice, much like giving a mouse a cookie, is don't):

(1, 1)  $AB=C$   
 $A=B^{-1}C$

We define the inverse of a square matrix as follows. Given two square matrices,  $A$  and  $B$ , if  $AB = I$ , the identity matrix with 1s on the diagonals and 0s on the off-diagonals, then  $B$  is the right-inverse of  $A$ , and can be represented as  $A^{-1}$ . With this background behind us, let's go ahead and use some of R's matrix operators. A difficulty in the real world is that some matrices cannot be inverted. For example, a so-called singular matrix has no inverse. Let's start with a simple correlation matrix:

$A = \begin{bmatrix} 1.00 & 0.14 & .. & 0.35 \\ 0.14 & 1.00 & .. & 0.09 \\ 0.35 & 0.09 & .. & 1.00 \end{bmatrix}$

In R, we can create the matrix first as a vector, and then give the vector the dimensions  $3 \times 3$ , thus turning it into a matrix. Note the way we do this to avoid duplicating  $A$ ; for very large data, this may be more compute efficient. The `is.matrix(X)` function will return `TRUE` if  $X$  has these attributes, and `FALSE` otherwise. You can coerce a data frame to a matrix by using the `as.matrix` function, but be aware that this method will produce a character matrix if there are any nonnumeric columns. We will never use anything but numbers in matrices in this book. When we have character data, we will use lists and data frames:

```
> A <- c(1.00, 0.14, 0.35, 0.14, 1.00, 0.09, 0.35, 0.09, 1.00)
> dim(A) <- c(3,3)
> A
      [,1] [,2] [,3]
[1,] 1.00 0.14 0.35
[2,] 0.14 1.00 0.09
[3,] 0.35 0.09 1.00
> dim(A)
[1] 3 3
```

Some useful matrix operators in R are displayed in [Table 1-3](#).

Table 1-3: Matrix operators in R

Operator	Operator	Code Example
Transposition	t	t(A)
Matrix Multiplication	%%	A %% B
Inversion	solve()	solve(A)

Because the correlation matrix is square and symmetric, its transpose is the same as  $A$ . The inverse multiplied by the original matrix should give us the identity matrix. The matrix inversion algorithm accumulates some degree of rounding error, but not very much at all, and the matrix product of  $A^{-1}$  and  $A$  is the identity matrix, which rounding makes apparent:

```
> Ainv <- solve(A)
> matProd <- Ainv % * % A
> round(matProd)
[ ,1] [ ,2] [ ,3] [ 1, ] 1 0 0 [ 2, ] 0 1 0
[ 3, ] 0 0 1
```

If  $A$  has an inverse, you can either premultiply or postmultiply  $A$  by  $A^{-1}$  and you will get an identity matrix in either case.

1.3.5 More on Lists

Recall our first R session in which you created a list with your name and alma mater. Lists are unusual in a couple of ways, and are very helpful when we have "ragged" data arrays in which the variables have unequal numbers of observations. For example, assume that my coauthor, Dr Pace, taught three sections of the same statistics course, each of which had a different number of students. The final grades might look like the following:

```
> section1 <- c(57.3, 70.6, 73.9, 61.4, 63.0, 66.6, 74.8, 71.8, 63.2, 72.3, 61.9, 70.0)
> section2 <- c(74.6, 74.5, 75.9, 77.4, 79.6, 70.2, 67.5, 75.5, 68.2, 81.0, 69.6, 75.6,
69.5, 72.4, 77.1)
> section3 <- c(80.5, 79.2, 83.6, 74.9, 81.9, 80.3, 79.5, 77.3, 92.7, 76.4, 82.0, 68.9,
77.6, 74.6)
> allSections <- list(section1,section2,section3)
```



```

> allSections
[[1]]
[1] 57.3 70.6 73.9 61.4 63.0 66.6 74.8 71.8 63.2 72.3 61.9 70.0

[[2]]
[1] 74.6 74.5 75.9 77.4 79.6 70.2 67.5 75.5 68.2 81.0 69.6 75.6 69.5 72.4 77.1

[[3]]
[1] 80.5 79.2 83.6 74.9 81.9 80.3 79.5 77.3 92.7 76.4 82.0 68.9 77.6 74.6

> section_means <- sapply(allSections, mean)
> round(section_means, 2)
[1] 67.23 73.91 79.24
> section_sdev <- sapply(allSections, sd)
> round(section_sdev, 2)
[1] 5.74 4.17 5.40

```

We combined the three classes into a list and then used the `sapply` function to find the means and standard deviations for the three classes. As with the name and address data, the list uses two square brackets for indexing. The `[[1]]` indicates the first element of the list, which is a number contained in another list. The `sapply` function produces a simplified view of the means and standard deviations. Note that the `lapply` function works here as well, as the calculation of the variances for the separate sections shows, but produces a different kind of output from that of `sapply`, making it clear that the output is yet another list:

```

> lapply(allSections, var)
[[1]]
[1] 32.99515

[[2]]
[1] 17.3521

[[3]]
[1] 29.18879

```

### 1.3.6 A Quick Introduction to Data Frames

As I mentioned earlier, the most common data structure for statistics is the data frame. A data frame is a list, but rectangular like a matrix. Every column represents a variable or a factor in the dataset. Every row in the data frame represents a *case*, either an object or an individual about whom data have been collected, so that, ideally, each case will have a score for every variable and a level for every factor. Of course, as we will discuss in more detail in Chapter 2, real data are far from ideal.

Here is the roster of the 2014-2015 Clemson University mens' basketball team, which I downloaded from the university's website. I saved the roster as a comma-separated value (CSV) file and then read it into R using the `read.csv` function. Please note that in this case, the file 'roster.csv' was saved in our working directory. Recall that earlier we discussed both `getwd()` and `setwd()`, these can be quite helpful. As you can see, when you create data using this method, the file will automatically become a data frame in R:

```

> roster <- read.csv("roster.csv")
> roster

```

	Jersey	Name	Position	Inches	Pounds	Class
1	0	Rooks, Patrick	G	74	190	freshman
2	1	Ajukwa, Austin	G	78	205	sophomore
3	3	Holmes, Avry	G	74	205	junior
4	5	Blossomgame, Jaron	F	79	215	sophomore
5	10	DeVoe, Gabe	G	75	200	freshman
6	12	Hall, Rod	G	73	205	senior
7	15	Grantham, Donte	F	80	205	freshman
8	20	Roper, Jordan	G	72	165	junior
9	21	Harrison, Damarcus	G	76	205	senior
10	33	Smith, Josh	F	80	245	junior
11	35	Nnoko, Landry	C	82	255	junior
12	44	McGillan, Riley	G	72	175	junior
13	50	Djitte, Sidy	C	82	240	sophomore

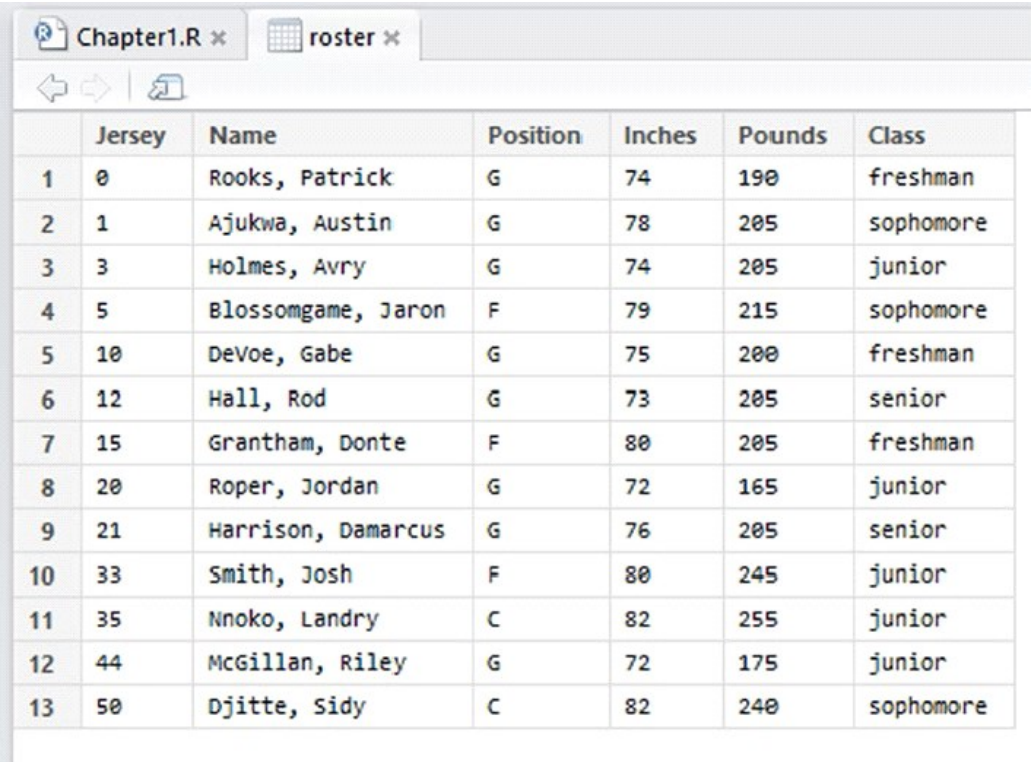
```

> str(roster)
' data.frame ': 13 obs. of 6 variables:
 $ Jersey : int 0 1 3 5 10 12 15 20 21 33 ...
 $ Name : Factor w/ 13 levels "Ajukwa, Austin", ...: 1

```

```
1 1 8 2 3 6 5 1 2 7 1 3 . . .
$ Position: Factor w/ 3 levels "C", "F", "G": 3 3 3 2
3
3 2 3 3 2 . . .
$ Inches: int 74 78 74 79 75 73 80 72 76 80 . . .
$ Pounds: int 190 205 205 215 200 205 205 165 20
5 245 . . .
$ Class: Factor w/ 4 levels "freshman", "junior",
. . : 1 4 2 4 1 3 1 2 3 2 . . .
```

To view your data without editing them, you can use the View command (see [Figure 1-2](#)).



	Jersey	Name	Position	Inches	Pounds	Class
1	0	Rooks, Patrick	G	74	190	freshman
2	1	Ajukwa, Austin	G	78	205	sophomore
3	3	Holmes, Avry	G	74	205	junior
4	5	Blossomgame, Jaron	F	79	215	sophomore
5	10	DeVoe, Gabe	G	75	200	freshman
6	12	Hall, Rod	G	73	205	senior
7	15	Grantham, Donte	F	80	205	freshman
8	20	Roper, Jordan	G	72	165	junior
9	21	Harrison, Damarcus	G	76	205	senior
10	33	Smith, Josh	F	80	245	junior
11	35	Nnoko, Landry	C	82	255	junior
12	44	McGillan, Riley	G	72	175	junior
13	50	Djitte, Sidy	C	82	240	sophomore

Figure 1-2: Data frame in the viewer window