IN THIS CHAPTER

- Getting R and RStudio on your computer
- Plunging into a session with R
- Working with R functions
- Working with R structures

Chapter 1

R: What It Does and How It Does It

So you're ready to journey into the wonderful world of R! Designed by and for statisticians and data scientists, R has a short but illustrious history.

In the 1990s, Ross Ihaka and Robert Gentleman developed R at the University of Auckland, New Zealand. The Foundation for Statistical Computing supports R, which is growing more popular by the day.

Getting R

If you don't already have R on your computer, the first thing to do is to download R and install it.

You'll find the appropriate software on the website of the Comprehensive R Archive Network (CRAN). In your browser, type this web address if you work in Windows:

```
cran.r-project.org/bin/windows/base
```

Type this one if you work on the Mac:

```
cran.r-project.org/bin/macosx
```

Click the link to download R. This puts a `win.exe` file in your Windows computer or a `pkg` file in your Mac. In either case, follow the usual installation procedures. When installation is complete, Windows users see two R icons on their desktop, one for 32-bit processors and one for 64-bit processors (pick the one that's right for you). Mac users see an R icon in their `Application` folder.

TIP

Both addresses provide helpful links to FAQs. The windows-related one also has the link Installation and Other Instructions.

Getting RStudio

Working with R is a lot easier if you do it through an application called RStudio. Computer honchos refer to RStudio as an IDE (*I*ntegrated *D*evelopment *E*nvironment). Think of it as a tool that helps you write, edit, run, and keep track of your R code, and as an environment that connects you to a world of helpful hints about R.

Here's the web address for this terrific tool:

```
www.rstudio.com/products/rstudio/download
```

Click the link for the installer for your computer's operating system — Windows, Mac, or a flavor of Linux — and again follow the usual installation procedures.

TIP

In this book, I work with R version 3.4.0 and RStudio version 1.0.143. By the time you read this, later versions of both might be available.

After you finish installing R and RStudio, click on your brand-new RStudio icon to open the window shown in Figure 1-1.

The large Console pane on the left runs R code. One way to run R code is to type it directly into the Console pane. I show you another in a moment.

The other two panes provide helpful information as you work with R. The Environment/History pane is in the upper right. The Environment tab keeps track of the things you create (which R calls objects) as you work with R. The History tab tracks R code that you enter.

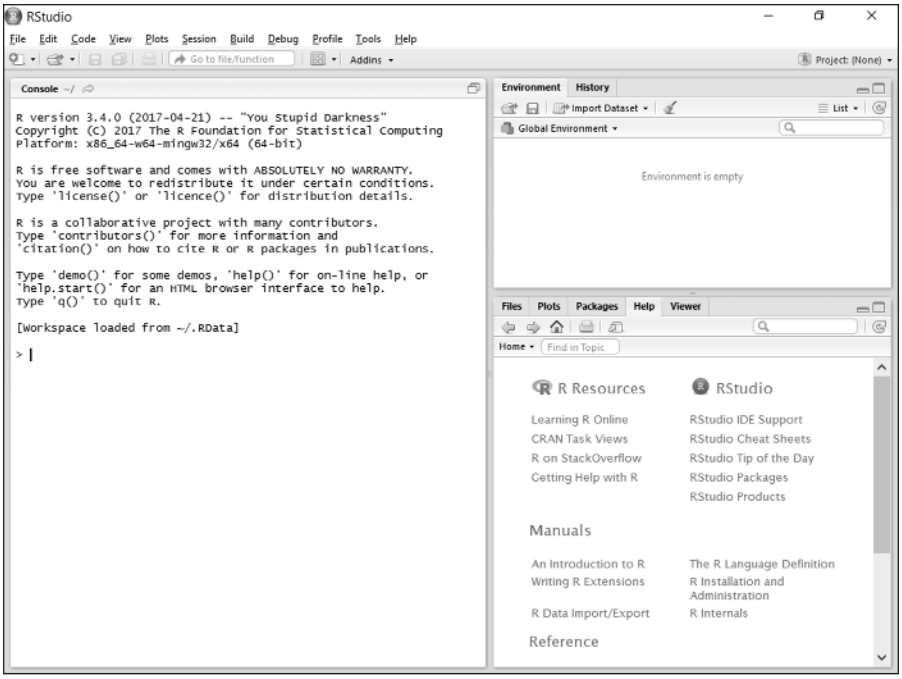

FIGURE 1-1: RStudio, immediately after you install it and click on its icon.

TIP

Get used to the word *object.* Everything in R is an object.The Files/Plots/Packages/Help pane is in the lower right. The Files tab shows files you create. The Plots tab holds graphs you create from your data. The Packages tab shows add-ons (called *packages*) that have downloaded with R. Bear in mind that *downloaded* doesn't mean "ready to use." To use a package's capabilities, one more step is necessary, and trust me — you'll want to use packages.

Figure 1-2 shows the Packages tab. I discuss packages later in this chapter.

The Help tab, shown in Figure 1-3, links you to a wealth of information about R and RStudio.

To tap into the full power of RStudio as an IDE, click the icon in the upper right corner of the Console pane. That changes the appearance of RStudio so that it looks like Figure 1-4.

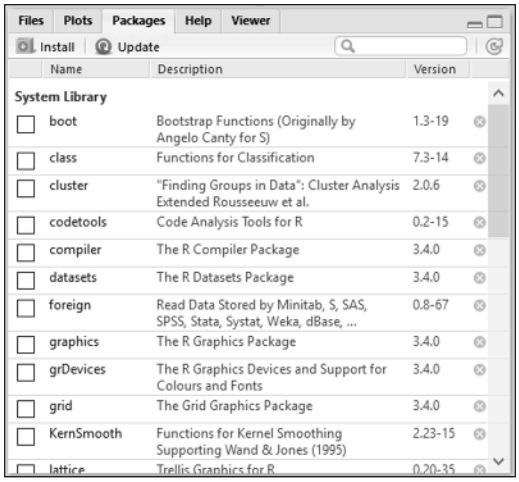

FIGURE 1-2: The RStudio Packages tab.

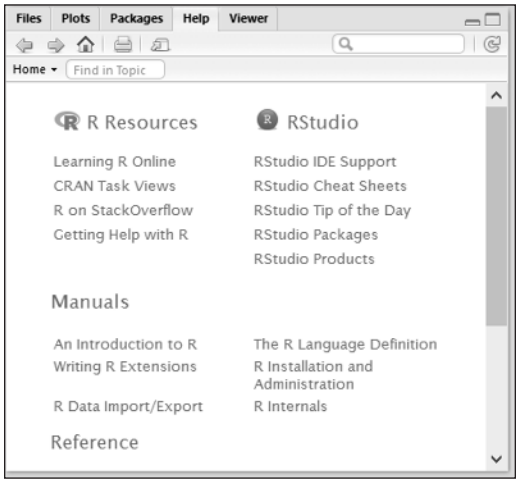

FIGURE 1-3: The RStudio Help tab.

The Console pane relocates to the lower left. The new pane in the upper left is the Scripts pane. You type and edit code in the Scripts pane by pressing Ctrl+R (Command+Enter on the Mac), and then the code executes in the Console pane.

TIP

Ctrl+Enter works just like Ctrl+R. You can also highlight lines of code in the Scripts pane and select Code➪ Run Selected Line(s) from RStudio's main menu.

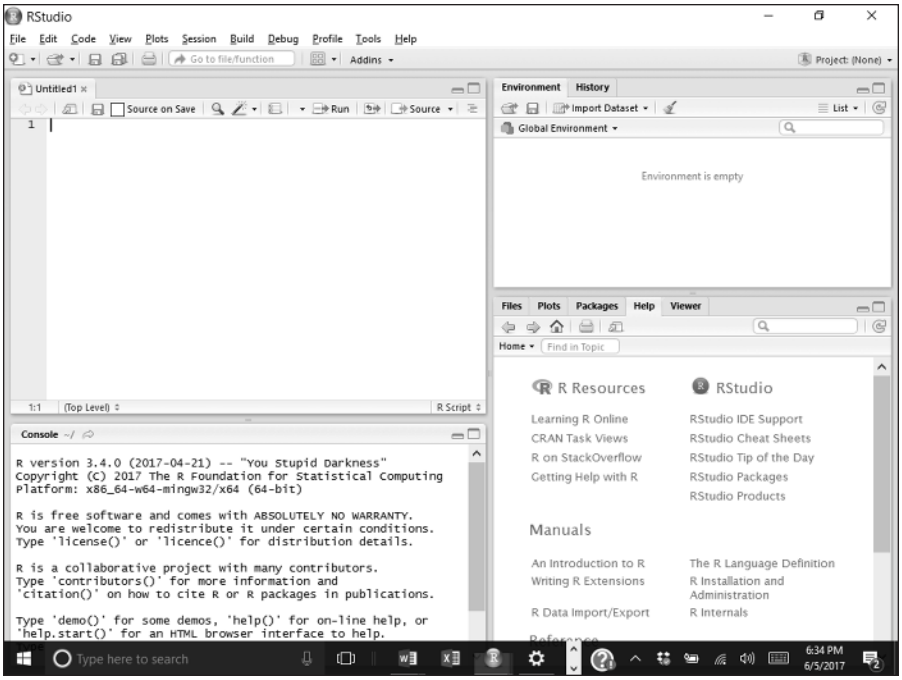

FIGURE 1-4: RStudio after you click the icon in the upper right corner of the Console pane.

A Session with R

Before you start working, select File➪ Save As from the main menu and then save your work file as My First R Session. This relabels the tab in the Scripts pane with the name of the file and adds the `.R` extension. This also causes the filename (along with the `.R` extension) to appear on the Files tab.

The working directory

What exactly does R save, and where does R save it? What R saves is called the *workspace*, which is the environment you're working in. R saves the workspace in the *working directory.* In Windows, the default working directory is

```
C:\Users\<User Name>\Documents
```

On a Mac, it's

```
/Users/<User Name>
```

If you ever forget the path to your working directory, type

```
> getwd()
```

in the Console pane, and R returns the path onscreen.

TIP

In the Console pane, you don't type the right-pointing arrowhead at the beginning of the line. That's a prompt.

My working directory looks like this:

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

Note the direction the slashes are slanted. They're opposite to what you typically see in Windows file paths. This is because R uses \ as an *escape character* — whatever follows the \ means something different from what it usually means. For example, \t in R means *Tab key.*

TIP

You can also write a Windows file path in R as

```
C:\\Users\\<User Name>\\Documents
```

If you like, you can change the working directory:

```
> setwd(<file path>)
```

Another way to change the working directory is to select Session➪ Set Working Directory➪ Choose Directory from the main menu.

Getting started

Let's get down to business and start writing R code. In the Scripts pane, type

```
x <- c(5,10,15,20,25,30,35,40)
```

and then press Ctrl+R.

That puts this line into the Console pane:

```
> x <- c(5,10,15,20,25,30,35,40)
```

As I say in an earlier Tip paragraph, the right-pointing arrowhead (the greater-than sign) is a prompt that R puts in the Console pane. You don't see it in the Scripts pane.

Here's what R just did: The arrow-sign says that x gets assigned whatever is to the right of the arrow-sign. Think of the arrow-sign as R's *assignment operator.* So the set of numbers 5, 10, 15, 20 . . . 40 is now assigned to x.

REMEMBER

In R-speak, a set of numbers like this is a *vector.* I tell you more about this concept in the later section "R Structures."

You can read that line of code as "x gets the vector 5, 10, 15, 20."

Type **x** into the Scripts pane and press Ctrl+R, and here's what you see in the Console pane:

```
> x
[1]  5 10 15 20 25 30 35 40
```

The 1 in square brackets is the label for the first line of output. So this signifies that 5 is the first value.

Here you have only one line, of course. What happens when R outputs many values over many lines? Each line gets a bracketed numeric label, and the number corresponds to the first value in the line. For example, if the output consists of 23 values and the eighteenth value is the first one on the second line, the second line begins with [18].

Creating the vector x causes the Environment tab to look like Figure 1-5.

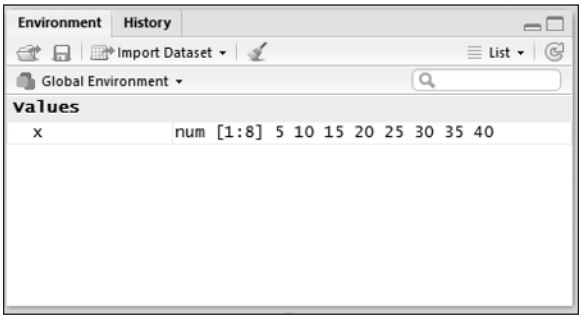

FIGURE 1-5: The RStudio Environment tab after creating the vector x.

TIP

Another way to see the objects in the environment is to type **ls()** into the Scripts pane and then press Ctrl+R. Or you can type **> ls()** directly into the Console pane and press Enter. Either way, the result in the Console pane is

```
[1] "x"
```

Now you can work with x. First, add all numbers in the vector. Typing **sum(x)** in the Scripts pane (be sure to follow with Ctrl+R) executes the following line in the Console pane:

```
> sum(x)
[1] 180
```

How about the average of the numbers in vector x?

That would involve typing **mean(x)** in the Scripts pane, which (when followed by Ctrl+R) executes

```
> mean(x)
[1] 22.5
```

in the Console pane.

TIP

As you type in the Scripts pane or in the Console pane, you see that helpful information pops up. As you become experienced with RStudio, you learn how to use that information.

Variance is a measure of how much a set of numbers differ from their mean. Here's how to use R to calculate variance:

```
> var(x)
[1] 150
```

What, exactly, is variance and what does it mean? (Shameless plug alert.) For the answers to these and numerous other questions about statistics and analysis, read one of the most classic works in the English language: *Statistical Analysis with R For Dummies* (written by yours truly and published by Wiley).

After R executes all these commands, the History tab looks like Figure 1-6.

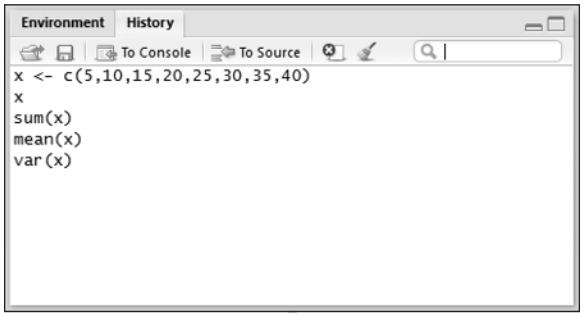

FIGURE 1-6: The History tab, after creating and working with a vector.

To end a session, select File➪Quit Session from the main menu or press Ctrl+Q. As Figure 1-7 shows, a dialog box opens and asks what you want to save from the session. Saving the selections enables you to reopen the session where you left off the next time you open RStudio (although the Console pane doesn't save your work).

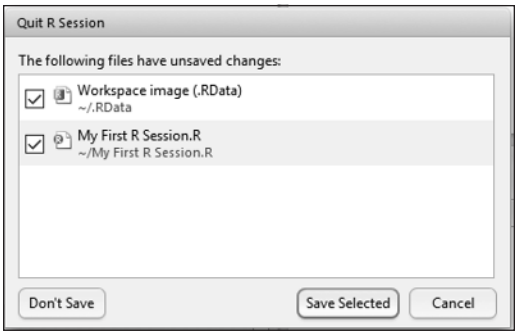

FIGURE 1-7: The Quit R Session dialog box.

REMEMBER

Moving forward, most of the time I don't say "Type this code into the Scripts pane and press Ctrl+Enter" whenever I take you through an example. I just show you the code and its output, as in the `var()` example.

REMEMBER

Also, sometimes I show code with the > prompt, and sometimes without. Generally, I show the prompt when I want you to see R code and its results. I don't show the prompt when I just want you to see R code that I create in the Scripts pane.

R Functions

The examples in the preceding section use `c()`, `sum()`, and `var()`. These are three *functions* built into R. Each one consists of a function name immediately followed by parentheses. Inside the parentheses are *arguments*. In the context of a function, *argument* doesn't mean "debate" or "disagreement" or anything like that. It's the math name for whatever a function operates on.

REMEMBER

Sometimes a function takes no arguments (as is the case with `ls()`). You still include the parentheses.

The functions in the examples I showed you are pretty simple: Supply an argument, and each one gives you a result. Some R functions, however, take more than one argument.

R has a couple of ways for you to deal with multi-argument functions. One way is to list the arguments in the order that they appear in the function's definition. R calls this *positional mapping*.

Here's an example. Remember when I created the vector x?

```
x <- c(5,10,15,20,25,30,35,40)
```

Another way to create a vector of those numbers is with the function `seq()`:

```
> y <- seq(5,40,5)
> y
[1]  5 10 15 20 25 30 35 40
```

Think of `seq()` as creating a "sequence." The first argument to `seq()` is the number to start the sequence *from* (5). The second argument is the number that ends the sequence — the number the sequence goes *to* (40). The third argument is the increment of the sequence — the amount the sequence increases *by* (5).

If you *name* the arguments, it doesn't matter how you order them:

```
> z <- seq(to=40,by=5,from=5)
> z
[1]  5 10 15 20 25 30 35 40
```

So when you use a function, you can place its arguments out of order, if you name them. R calls this *keyword matching*. This comes in handy when you use an R function that has many arguments. If you can't remember their order, use their names, and the function works.


TIP

For help on a particular function — `seq()`, for example — type **?seq**. When you run that code, helpful information appears on the Help tab and useful information pops up in a little window right next to where you're typing.

User-Defined Functions

R enables you to create your own functions, and here are the fundamentals on how to do it.

The form of an R function is

```
myfunction <- function(argument1, argument2, ...){
  statements
  return(object)
}
```

Here's a function for dealing with right triangles. Remember them? A right triangle has two sides that form a right angle, and a third side called a *hypotenuse*. You might also remember that a guy named Pythagoras showed that if one side has length *a* and the other side has length *b*, the length of the hypotenuse, *c*, is

$$c = \sqrt{a^2 + b^2}$$

So here's a simple function called `hypotenuse()` that takes two numbers `a` and `b`, (the lengths of the two sides of a right triangle) and returns `c`, the length of the hypotenuse:

```
hypotenuse <- function(a,b){
  hyp <- sqrt(a^2+b^2)
  return(hyp)
}
```

Type that code snippet into the Scripts pane and highlight it. Then press Ctrl+Enter. Here's what appears in the Console pane:

```
> hypotenuse <- function(a,b){
+   hyp <- sqrt(a^2+b^2)
+   return(hyp)
+ }
```

Each plus sign is a *continuation prompt*. It just indicates that a line continues from the preceding line.

And here's how to use the function:

```
> hypotenuse(3,4)
[1] 5
```

TIP

Writing R functions can encompass *way* more than I've shown you here. To learn more, take a look at *R For Dummies*, by Andrie de Vries and Joris Meys (Wiley).

Comments

A *comment* is a way of annotating code. Begin a comment with the # symbol, which, as everyone knows, is called an *octothorpe*. (Wait. What? "Hashtag?" Get atta here!) This symbol tells R to ignore everything to the right of it.

Comments help someone who has to read the code you've written. For example:

```
hypotenuse <- function(a,b){ # list the arguments
  hyp <- sqrt(a^2+b^2) # perform the computation
  return(hyp) # return the value
}
```

Here's a heads-up: I don't typically add comments to lines of code in this book. Instead, I provide detailed descriptions. In a book like this, I feel it's the best way to get the message across.

R Structures

As I mention in the "R Functions" section, earlier in this chapter, an R function can have many arguments. An R function can also have many outputs. To understand the possible inputs and outputs, you must understand the structures that R works with.

Vectors

The *vector* is the fundamental structure in R. I show it to you in earlier examples. It's an array of elements of the same type. The data elements in a vector are called *components*.

To create a vector, use the function `c()`, as I do in the earlier example:

```
x <- c(5,10,15,20,25,30,35,40)
```

In the vector `x`, of course, the components are numbers.

In a *character vector*, the components are quoted text strings:

```
> beatles <- c("john","paul","george","ringo")
```

It's also possible to have a *logical vector*, whose components are `TRUE` and `FALSE`, or the abbreviations `T` and `F`:

```
> w <- c(T,F,F,T,T,F)
```

To refer to a specific component of a vector, follow the vector name with a bracketed number:

```
> beatles[2]
[1] "paul"
```

Within the brackets, you can use a colon (`:`) to refer to two consecutive components:

```
> beatles[2:3]
[1] "paul"   "george"
```

Want to refer to nonconsecutive components? That's a bit more complicated, but doable via `c()`:

```
> beatles[c(2,4)]
[1] "paul"  "ringo"
```

Numerical vectors

In addition to `c()`, R provides two shortcut functions for creating numerical vectors. One, `seq()`, I showed you earlier:

```
> y <- seq(5,40,5)
> y
[1]  5 10 15 20 25 30 35 40
```

Without the third argument, the sequence increases by 1:

```
> y <- seq(5,40)
> y
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
[20] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

REMEMBER

On my screen, and probably on yours too, all the elements in `y` appear on one line. The printed page, however, is not as wide as the Console pane. Accordingly, I separated the output into two lines and added the R-style bracketed number `[20]` to the beginning of the second line.

TIP

R has a special syntax for creating a numerical vector whose elements increase by 1:

```
> y <- 5:40
> y
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
[20] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Another function, `rep()`, creates a vector of repeating values:

```
> quadrifecta <- c(7,8,4,3)
> repeated_quadrifecta <- rep(quadrifecta,3)
> repeated_quadrifecta
 [1] 7 8 4 3 7 8 4 3 7 8 4 3
```

You can also supply a vector as the second argument:

```
> rep_vector <-c(1,2,3,4)
> repeated_quadrifecta <- rep(quadrifecta,rep_vector)
```

The vector specifies the number of repetitions for each element. So here's what happens:

```
> repeated_quadrifecta
 [1] 7 8 8 4 4 4 3 3 3 3
```

The first element repeats once; the second, twice; the third, three times; and the fourth, four times.

You can use `append()` to add an item at the end of a vector:

```
> xx <- c(3,4,5)
> xx
[1] 3 4 5
> xx <- append(xx,6)
> xx
[1] 3 4 5 6
```

and you can use `prepend()` to add an item at the beginning of a vector:

```
> xx <- prepend(xx,2)
> xx
[1] 2 3 4 5 6
```

How many items are in a vector? That's

```
> length(xx)
[1] 5
```

Matrices

A *matrix* is a 2-dimensional array of data elements of the same type. You can have a matrix of numbers:

5	30	55	80
10	35	60	85
15	40	65	90
20	45	70	95
25	50	75	100

or a matrix of character strings:

"john"	"paul"	"george"	"ringo"
"groucho"	"harpo"	"chico"	"zeppo"
"levi"	"duke"	"larry"	"obie"

The numbers are a 5 (rows) X 4 (columns) matrix. The character strings matrix is 3 X 4.

To create this particular 5 X 4 numerical matrix, first create the vector of numbers from 5 to 100 in steps of 5:

```
> num_matrix <- seq(5,100,5)
```

Then you use R's `dim()` function to turn the vector into a 2-dimensional matrix:

```
> dim(num_matrix) <- c(5,4)
> num_matrix
     [,1] [,2] [,3] [,4]
[1,]    5   30   55   80
[2,]   10   35   60   85
[3,]   15   40   65   90
[4,]   20   45   70   95
[5,]   25   50   75  100
```

Note how R displays the bracketed row numbers along the side, and the bracketed column numbers along the top.

Transposing a matrix interchanges the rows with the columns. The `t()` function takes care of that:

```
> t(num_matrix)
     [,1] [,2] [,3] [,4] [,5]
[1,]    5   10   15   20   25
[2,]   30   35   40   45   50
[3,]   55   60   65   70   75
[4,]   80   85   90   95  100
```

The function `matrix()` gives you another way to create matrices:

```
> num_matrix <- matrix(seq(5,100,5),nrow=5)
> num_matrix
     [,1] [,2] [,3] [,4]
[1,]    5   30   55   80
[2,]   10   35   60   85
[3,]   15   40   65   90
[4,]   20   45   70   95
[5,]   25   50   75  100
```

If you add the argument `byrow=T`, R fills the matrix by rows, like this:

```
> num_matrix <- matrix(seq(5,100,5),nrow=5,byrow=T)
> num_matrix
     [,1] [,2] [,3] [,4]
[1,]    5   10   15   20
[2,]   25   30   35   40
[3,]   45   50   55   60
[4,]   65   70   75   80
[5,]   85   90   95  100
```

How do you refer to a specific matrix component? You type the matrix name and then, in brackets, the row number, a comma, and the column number:

```
> num_matrix[5,4]
[1] 100
```

To refer to a whole row (like the third one):

```
> num_matrix[3,]
[1] 45 50 55 60
```

and to a whole column (like the second one):

```
> num_matrix[,2]
[1] 10 30 50 70 90
```

Although it's a column, R displays it as a row in the Console pane.

BUT BEAR IN MIND . . .

As I mention, a matrix is a 2-dimensional array. In R, however, an array can have more than two dimensions. One well-known set of data (which I use as an example in Chapter 3) has three dimensions: Hair Color (Black, Brown, Red, Blond), Eye Color (Brown, Blue, Hazel, Green), and Gender (Male, Female). So this particular array is 4 X 4 X 2. It's called `HairEycColor` and it looks like this:

```
> HairEyeColor
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
  Black    32   11    10     3
  Brown    53   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8

, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    36    9     5     2
  Brown    66   34    29    14
  Red      16    7     7     7
  Blond     4   64     5     8
```

Each number represents the number of people in this group who have a particular combination of hair color, eye color, and gender — 16 brown-eyed, red-haired females, for example. (Why did I choose brown-eyed, red-haired females? Because I have the pleasure of looking at an extremely beautiful one every day!)

How would I refer to all the females? That's

```
HairEyeColor[,,2]
```

Lists

In R, a *list* is a collection of objects that aren't necessarily the same type. Suppose you're putting together some information on the Beatles:

```
> beatles <- c("john","paul","george","ringo")
```

One piece of important information might be each Beatle's age when he joined the group. John and Paul started singing together when they were 17 and 15, respectively, and 14–year-old George joined them soon after. Ringo, a late arrival, became a Beatle when he was 22. So

```
> ages <- c(17,15,14,22)
```

To combine the information into a list, you use the `list()` function:

```
> beatles_info <-list(names=beatles,age_joined=ages)
```

Naming each argument (`names, age_joined`) causes R to use those names as the names of the list components.

And here's what the list looks like:

```
> beatles_info
$names
[1] "john"   "paul"   "george" "ringo"

$age_joined
[1] 17 15 14 22
```

R uses the dollar sign ($) to indicate each component of the list. If you want to refer to a list component, you type the name of the list, the dollar sign, and the component name:

```
> beatles_info$names
[1] "john"   "paul"   "george" "ringo"
```

And to zero in on a particular Beatle, like the fourth one? You can probably figure out that it's

```
> beatles_info$names[4]
[1] "ringo"
```

R also allows you to use criteria inside the brackets. For example, to refer to members of the Fab Four who were older than 16 when they joined:

```
> beatles_info$names[beatles_info$age_joined > 16]
[1] "john"  "ringo"
```

Data frames

A list is a good way to collect data. A *data frame* is even better. Why? When you think about data for a group of individuals, you typically think in terms of rows that represent the individuals and columns that represent the data variables. And that's a data frame. If the terms *data set* or *data matrix* come to mind, you've got the right idea.

Here's an example. Suppose I have a set of six people:

> name <- c("al","barbara","charles","donna","ellen","fred") and that I have each person's height (in inches) and weight (in pounds):

```
> height <- c(72,64,73,65,66,71)
> weight <- c(195,117,205,122,125,199)
```

I also tabulate each person's gender:

```
> gender <- c("M","F","M","F","F","M")
```

Before I show you how to combine all these vectors into a data frame, I have to show you one more thing. The components of the `gender` vector are character strings. For purposes of data summary and analysis, it's a good idea to turn them into categories — the Male category and the Female category. To do this, I use the `factor()` function:

```
> factor_gender <-factor(gender)
> factor_gender
[1] M F M F F M
Levels: F M
```

In the last line of output, `Levels` is the term that R uses for "categories."

The function `data.frame()` works with the vectors to create a data frame:

```
> d <- data.frame(name,factor_gender,height,weight)
> d
     name factor_gender height weight
```

```
1      al            M     72    195
2 barbara            F     64    117
3 charles            M     73    205
4   donna            F     65    122
5   ellen            F     66    125
6    fred            M     71    199
```

Want to know the height of the third person?

```
> d[3,3]
[1] 73
```

How about all the information for the fifth person:

```
> d[5,]
   name factor_gender height weight
5 ellen             F     66    125
```

Like lists, data frames use the dollar sign. In this context, the dollar sign identifies a column:

```
> d$height
[1] 72 64 73 65 66 71
```

You can calculate statistics, like the average height:

```
> mean(d$height)
[1] 68.5
```

As is the case with lists, you can put criteria inside the brackets. This is often done with data frames, to summarize and analyze data within categories. To find the average height of the females:

```
> mean(d$height[d$factor_gender == "F"])
[1] 65
```

The double equal sign (==) in the brackets is a *logical operator.* Think of it as "if d$factor_gender is equal to "F".

REMEMBER

The double equal sign (`a == b`) distinguishes the logical operator ("if a equals b") from the assignment operator (a=b; "set a equal to b").

TIP

Yes, I know — I went through an involved explanation about `factor()` and how it's better to have categories (levels) than character strings, and then I had to put quote marks around `F` inside the brackets. R is quirky that way.

TIP

If you'd like to eliminate $ signs from your R code, you can use the function `with()`. You put your code inside the parentheses after the first argument, which is the data you're using.

For example,

```
> with(d,mean(height[factor_gender == "F"]))
```

is equivalent to

```
> mean(d$height[d$factor_gender == "F"])
```

How many rows are in a data frame?

```
> nrow(d)
[1] 6
```

And how many columns?

```
> ncol(d)
[1] 4
```

To add a column to a data frame, I use `cbind()`. Begin with a vector of scores:

```
> aptitude <- c(35,20,32,22,18,15)
```

Then add that vector as a column:

```
> d.apt <- cbind(d,aptitude)
> d.apt
     name factor_gender height weight aptitude
1      al             M     72    195       35
2 barbara             F     64    117       20
3 charles             M     73    205       32
4   donna             F     65    122       22
5   ellen             F     66    125       18
6    fred             M     71    199       15
```

Of for Loops and if Statements

Like many programming languages, R provides a way to iterate through its structures to get things done. R's way is called the *for loop*. And, like many languages, R gives you a way to test against a criterion: the *if* statement.

The general format of a `for` loop is

```
for counter in start:end{
               statement 1
                                                                    .
                                                                    .
                                                                    .
statement n
}
```

As you might imagine, `counter` tracks the iterations.

The simplest general format of an `if` statement is

```
if(test){statement to execute if test is TRUE}
else{statement to execute if test is FALSE}
```

Here's an example that incorporates both. I have one vector `xx`:

```
> xx
[1] 2 3 4 5 6
```

And another vector `yy` with nothing in it at the moment:

```
> yy <-NULL
```

I want the components of `yy` to reflect the components of `xx`: If a number in `xx` is an odd number, I want the corresponding component of `yy` to be `"ODD"`, and if the `xx` number is even, I want the `yy` component to be `"EVEN"`.

How do I test a number to see whether it's odd or even? Mathematicians have developed *modular arithmetic*, which is concerned with the remainder of a division operation. If you divide *a* by *b* and the result has a remainder of *r*, mathematicians say that "a *modulo* b is r." So 10 divided by 3 leaves a remainder of 1, and 10 modulo 3 is 1. Typically, *modulo* gets shortened to *mod*, so that would be "10 mod 3 = 1."

Most computer languages write 10 mod 3 as `mod(10,3)`. (Excel does that, in fact.). R does it differently: R uses the double percent sign (`%%`) as its *mod operator:*

```
> 10 %% 3
[1] 1
> 5 %% 2
[1] 1
> 4 %% 2
[1] 0
```

I think you're getting the picture: `if xx[i] %% 2 == 0`, then `xx[i]` is even. Otherwise, it's odd.

Here, then, is the `for` loop and the `if` statement:

```
for(i in 1:length(xx)){
if(xx[i] %% 2 == 0){yy[i]<- "EVEN"}
else{yy[i] <- "ODD"}
}

> yy
[1] "EVEN" "ODD"  "EVEN" "ODD"  "EVEN"
```