5 An Overview of the R Language

Learning a computer language is a lot like learning a spoken language (only much simpler). If you're just visiting a foreign country, you might learn enough phrases to get by without really understanding how the language is structured. Similarly, if you're just trying to do a couple of simple things with R (like drawing some charts), you can probably learn enough from examples to get by.

However, if you want to learn a new spoken language really well, you have to learn about syntax and grammar: verb conjugation, proper articles, sentence structure, and so on. The same is true with R: if you want to learn how to program effectively in R, you'll have to learn more about the syntax and grammar.

This chapter gives an overview of the R language, designed to help you understand R code and write your own. I'll assume that you've spent a little time looking at R syntax (maybe from reading Chapter 3). Here's a quick overview of how R works.

Expressions

R code is composed of a series of *expressions*. Examples of expressions in R include assignment statements, conditional statements, and arithmetic expressions. Here are a few examples of expressions:

```
> x <- 1
> if (1 > 2) "yes" else "no"
[1] "no"
> 127 %% 10
[1] 7
```

Expressions are composed of objects and functions. You may separate expressions with new lines or with semicolons. For example, here is a series of expressions separated by semicolons:

```
> "this expression will be printed"; 7 + 13; exp(0+1i*pi)
[1] "this expression will be printed"
[1] 20
[1] -1+0i
```

Objects

All R code manipulates *objects*. The simplest way to think about an object is as a "thing" that is represented by the computer. Examples of objects in R include numeric vectors, character vectors, lists, and functions. Here are some examples of objects:

```
> # a numerical vector (with five elements)
> c(1,2,3,4,5)
[1] 1 2 3 4 5

> # a character vector (with one element)
> "This is an object too"
[1] "This is an object too"

> # a list
> list(c(1,2,3,4,5),"This is an object too", " this whole thing is a list")
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] "This is an object too"

[[3]]
[1] " this whole thing is a list"

> # a function
> function(x,y) {x + y}
function(x,y) {x + y}
```

Symbols

Formally, variable names in R are called *symbols*. When you assign an object to a variable name, you are actually assigning the object to a symbol in the current environment. (Somewhat tautologically, an *environment* is defined as the set of symbols that are defined in a certain context.) For example, the statement:

```
> x <- 1
```

assigns the symbol "*x*" to the object "1" in the current environment. For a more complete discussion of symbols and environments, see Chapter 8.

Functions

A *function* is an object in R that takes some input objects (called the *arguments* of the function) and returns an output object. All work in R is done by functions. Every statement in R—setting variables, doing arithmetic, repeating code in a loop—can be written as a function. For example, suppose that you had defined a variable `animals` pointing to a character vector with four elements: "cow," "chicken," "pig," and "tuba." Here is a statement that assigns this `variable`:

```
> animals <- c("cow", "chicken", "pig", "tuba")
```

Suppose that you wanted to change the fourth element to the word "duck." Normally, you would use a statement like this:

```
> animals[4] <- "duck"
```

This statement is parsed into a call to the `[<-` function. So you could actually use this equivalent expression:[1]

```
> `[<-`(animals,4,"duck")
```

In practice, you would probably never write this statement as a function call; the bracket notation is much more intuitive and much easier to read. However, it is helpful to know that every operation in R is a function. Because you know that this assignment is really a function call, it means that you can inspect the code of the underlying function, search for help on this function, or create methods with the same name for your own object classes.[2]

Here are a few more examples of R syntax and the corresponding function calls:

```
> # pretty assignment
> apples <- 3
> # functional form of assignment
> `<-`(apples,3)
> apples
[1] 3

> # another assignment statement, so that we can compare apples and oranges
> `<-`(oranges,4)
> oranges
[1] 4

> # pretty arithmetic expression
> apples + oranges
[1] 7
> # functional form of arithmetic expression
> `+`(apples,oranges)
[1] 7

> # pretty form of if-then statement
> if (apples > oranges) "apples are better" else "oranges are better"
[1] "oranges are better"
> # functional form of if-then statement
> `if`(apples > oranges,"apples are better","oranges are better")
[1] "oranges are better"
> x <- c("apple","orange","banana","pear")

> # pretty form of vector reference
> x[2]
[1] "orange"
```

1. This expression acts slightly differently, because the result is not printed on the R console. However, the result is the same:

```
> animals
[1] "cow"     "chicken" "pig"     "duck"
```

2. See Chapter 10 for more information on object-oriented programming using R.

```
> # functional form or vector reference
> `[`(x,2)
[1] "orange"
```

Objects Are Copied in Assignment Statements

In assignment statements, most objects are immutable. Immutable objects are a good thing: for multithreaded programs, immutable objects help prevent errors. R will copy the object, not just the reference to the object. For example:

```
> u <- list(1)
> v <- u
> u[[1]] <- "hat"
> u
[[1]]
[1] "hat"

> v
[[1]]
[1] 1
```

This applies to vectors, lists, and most other primitive objects in R.

This is also true in function calls. Consider the following function, which takes two arguments: a vector `x` and an index `i`. The function sets the `i`th element of `x` to 4 and does nothing else:

```
> f <- function(x,i) {x[i] = 4}
```

Suppose that we define a vector `w` and call `f` with `x = w` and `i = 1`:

```
> w <- c(10, 11, 12, 13)
> f(w,1)
```

The vector `w` is copied when it is passed to the function, so it is not modified by the function:

```
> w
[1] 10 11 12 13
```

The value `x` is modified inside the context of the function. Technically, the R interpreter copies the object assigned to `w` and then assigns the symbol `x` to point at the copy. We will talk about how you can actually create mutable objects, or pass references to objects, when we talk about environments.

Although R will behave as if every assignment makes a new copy of an object, in many cases R will actually modify the object in place. For example, consider the following code fragment:

```
> v <- 1:100
> v[50] <- 27
```

R does not actually copy the vector when the 50th element is altered; instead, R modifies the vector in place. Semantically, this is identical, but the performance is much better. See the R Internals Guide for more information about how this works.

Everything in R Is an Object

In the last few sections, most examples of objects were objects that stored data: vectors, lists, and other data structures. However, everything in R is an object: functions, symbols, and even R expressions.

For example, function names in R are really symbol objects that point to function objects. (That relationship is, in turn, stored in an environment object.) You can assign a symbol to refer to a numeric object and then change the symbol to refer to a function:

```
> x <- 1
> x
[1] 1
> x(2)
Error: could not find function "x"
> x <- function(i) i^2
> x
function(i) i^2
> x(2)
[1] 4
```

You can even use R code to construct new functions. If you really wanted to, you could write a function that modifies its own definition.

Special Values

There are a few special values that are used in R.

NA

In R, the NA values are used to represent missing values. (NA stands for "not available.") You may encounter NA values in text loaded into R (to represent missing values) or in data loaded from databases (to replace NULL values).

If you expand the size of a vector (or matrix or array) beyond the size where values were defined, the new spaces will have the value NA:

```
> v <- c(1,2,3)
> v
```

```
[1] 1 2 3
> length(v) <- 4
> v
[1]  1  2  3 NA
```

Inf and -Inf

If a computation results in a number that is too big, R will return `Inf` for a positive number and `-Inf` for a negative number (meaning positive and negative infinity, respectively):

```
> 2 ^ 1024
[1] Inf
> - 2 ^ 1024
[1] -Inf
```

This is also the value returned when you divide by 0:

```
> 1 / 0
[1] Inf
```

NaN

Sometimes, a computation will produce a result that makes little sense. In these cases, R will often return `NaN` (meaning "not a number"):

```
> Inf - Inf
[1] NaN
> 0 / 0
[1] NaN
```

NULL

Additionally, there is a null object in R, represented by the symbol `NULL`. (The symbol `NULL` always points to the same object.) `NULL` is often used as an argument in functions to mean that no value was assigned to the argument. Additionally, some functions may return `NULL`. Note that `NULL` is not the same as `NA`, `Inf`, `-Inf`, or `NaN`.

Coercion

When you call a function with an argument of the wrong type, R will try to coerce values to a different type so that the function will work. There are two types of coercion that occur automatically in R: coercion with formal objects and coercion with built-in types.

With generic functions, R will look for a suitable method. If no exact match exists, then R will search for a coercion method that converts the object to a type for which a suitable method does exist. (The method for creating coercion functions is described in "Creating Coercion Methods" on page 131.)

Additionally, R will automatically convert between built-in object types when appropriate. R will convert from more specific types to more general types. For example, suppose that you define a vector `x` as follows:

```
> x <- c(1, 2, 3, 4, 5)
> x
[1] 1 2 3 4 5
> typeof(x)
[1] "double"
> class(x)
[1] "numeric"
```

Let's change the second element of the vector to the word "hat." R will change the object class to `character` and change all the elements in the vector to `char`:

```
> x[2] <- "hat"
> x
[1] "1"   "hat" "3"   "4"   "5"
> typeof(x)
[1] "character"
> class(x)
[1] "character"
```

Here is an overview of the coercion rules:

- Logical values are converted to numbers: `TRUE` is converted to `1` and `FALSE` to `0`.
- Values are converted to the simplest type required to represent all information.
- The ordering is roughly logical < integer < numeric < complex < character < list.
- Objects of type `raw` are not converted to other types.
- Object attributes are dropped when an object is coerced from one type to another.

You can inhibit coercion when passing arguments to functions by using the `AsIs` function (or, equivalently, the `I` function). For more information, see the help file for `AsIs`.

Many newcomers to R find coercion nonintuitive. Strongly typed languages (like Java) will raise exceptions when the object passed to a function is the wrong type but will not try to convert the object to a compatible type. As John Chambers (who developed the S language) describes:

> In the early coding, there was a tendency to make as many cases "work" as possible. In the later, more formal, stages the conclusion was that converting richer types to simpler automatically in all situations would lead to confusing, and therefore untrustworthy, results.[3]

In practice, I rarely encounter situations where values are coerced in undesirable ways. Usually, I use R with numeric vectors that are all the same type, so coercion simply doesn't apply.

The R Interpreter

R is an interpreted language. When you enter expressions into the R console (or run an R script in batch mode), a program within the R system, called the *interpreter*,

3. From [Chambers2008], p. 154.

executes the actual code that you wrote. Unlike C, C++, and Java, there is no need to compile your programs into an object language. Other examples of interpreted languages are Common Lisp, Perl, and JavaScript.

All R programs are composed of a series of expressions. These expressions often take the form of function calls. The R interpreter begins by parsing each expression, translating syntactic sugar into functional form. Next, R substitutes objects for symbols (where appropriate). Finally, R evaluates each expression, returning an *object*. For complex expressions, this process may be recursive. In some special cases (such as conditional statements), R does not evaluate all arguments to a function. As an example, let's consider the following R expression:

```
> x <- 1
```

On an R console, you would typically type `x <- 1` and then press the Enter key. The R interpreter will first translate this expression into the following function call:

```
`<-`(x, 1)
```

Next, the interpreter evaluates this function. It assigns the constant value `1` to the symbol `x` in the current environment and then returns the value `1`.

Let's consider another example. (We'll assume it's from the same session, so that the symbol `x` is mapped to the value `1`.)

```
> if (x > 1) "orange" else "apple"
[1] "apple"
```

Here is how the R interpreter would evaluate this expression. I typed `if (x > 1) "orange" else "apple"` into the R console and pressed the Enter key. The entire line is the expression that was evaluated by the R interpreter. The R interpreter parsed this expression and identified it as a set of R expressions in an if-then-else control structure. To evaluate that expression, the R interpreter begins by evaluating the *condition* `(x > 1)`. If the condition is true, then R would evaluate the next statement (in this example, `"orange"`). Otherwise, R would evaluate the statement after the `else` keyword (in this example, `"apple"`). We know that *x* is equal to 1. When R evaluates the condition statement, the result is false. So R does not evaluate the statement after the condition. Instead, R will evaluate the expression after the `else` keyword. The result of this expression is the character vector `"apple"`. As you can see, this is the value that is returned on the R console.

If you are entering R expressions into the R console, then the interpreter will pass objects returned to the console to the `print` function.

Some functionality is implemented internally within the R system. These calls are made using the `.Internal` function. Many functions use `.Internal` to call internal R system code. For example, the graphics function `plot.xy` is implemented using `.Internal`:

```
> plot.xy
function (xy, type, pch = par("pch"), lty = par("lty"), col = par("col"),
    bg = NA, cex = 1, lwd = par("lwd"), ...)
.Internal(plot.xy(xy, type, pch, lty, col, bg, cex, lwd, ...))
```

```
<bytecode: 0x11949f828>
<environment: namespace:graphics>
```

In a few cases, the overhead for calling `.Internal` within an R function is too high. R includes a mechanism to define functions that are implemented completely internally.

You can identify these functions because the body of the function contains a call to the function `.Primitive`. For example, the assignment operator is implemented through a primitive function:

```
> `<-`
.Primitive("<-")
```

This mechanism is used for only a few basic functions where performance is critical. You can find a current list of these functions in [RInternals2009].

Seeing How R Works

To end this overview of the R language, I wanted to share a few functions that are convenient for seeing how R works. As you may recall, R expressions are R objects. This means that it is possible to parse expressions in R, or partially evaluate expressions in R, and see how R interprets them. This can be very useful for learning how R works or for debugging R code.

As noted above, the R interpreter goes through several steps when evaluating statements. The first step is to parse a statement, changing it into proper functional form. It is possible to view the R interpreter to see how a given expression is evaluated. As an example, let's use the same R code fragment that we used in "The R Interpreter" on page 57:

```
> if (x > 1) "orange" else "apple"
[1] "apple"
```

To show how this expression is parsed, we can use the `quote()` function. This function will parse its argument but not evaluate it. By calling `quote`, an R expression returns a "language" object:

```
> typeof(quote(if (x > 1) "orange" else "apple"))
[1] "language"
```

Unfortunately, the `print` function for language objects is not very informative:

```
> quote(if (x > 1) "orange" else "apple")
if (x > 1) "orange" else "apple"
```

However, it is possible to convert a language object into a list. By displaying the language object as a list, it is possible to see how R evaluates an expression. This is the *parse tree* for the expression:

```
> as(quote(if (x > 1) "orange" else "apple"),"list")
[[1]]
`if`

[[2]]
x > 1
```

```
[[3]]
[1] "orange"

[[4]]
[1] "apple"
```

We can also apply the `typeof` function to every element in the list to see the type of each object in the parse tree:[4]

```
> lapply(as(quote(if (x > 1) "orange" else "apple"), "list"),typeof)
[[1]]
[1] "symbol"

[[2]]
[1] "language"

[[3]]
[1] "character"

[[4]]
[1] "character"
```

In this case, we can see how this expression is interpreted. Notice that some parts of the if-then statement are not included in the parsed expression (in particular, the `else` keyword). Also, notice that the first item in the list is a *symbol*. In this case, the symbol refers to the `if` function. So, although the syntax for the if-then statement is different from a function call, the R parser translates the expression into a function call before evaluating the expression. The function name is the first item, and the arguments are the remaining items in the list.

For constants, there is only one item in the returned list:

```
> as.list(quote(1))
[[1]]
[1] 1
```

By using the `quote` function, you can see that many constructions in the R language are just *syntactic sugar* for function calls. For example, let's consider looking up the second item in a vector `x`. The standard way to do this is through R's bracket notation, so the expression would be `x[2]`. An alternative way to represent this expression is as a function: `` `[`(x,2) ``. (Function names that contain special characters need to be encapsulated in backquotes.) Both of these expressions are interpreted the same way by R:

```
> as.list(quote(x[2]))
[[1]]
`[`
```

4. As a convenient shorthand, you can omit the `as` function because R will automatically coerce the language object to a list. This means you can just use a command like:

```
> lapply(quote(if (x > 1) "orange" else "apple"),typeof)
```

Coercion is explained in "Coercion" on page 56.

```
[[2]]
x

[[3]]
[1] 2

> as.list(quote(`[`(x,2)))
[[1]]
`[`

[[2]]
x

[[3]]
[1] 2
```

As you can see, R interprets both of these expressions identically. Clearly, the operation is not reversible (because both expressions are translated into the same parse tree). The `deparse` function can take the parse tree and turn it back into properly formatted R code. (The `deparse` function will use proper R syntax when translating a language object back into the original code.) Here's how it acts on these two bits of code:

```
> deparse(quote(x[2]))
[1] "x[2]"
> deparse(quote(`[`(x,2)))
[1] "x[2]"
```

As you read through this book, you might want to try using `quote`, `substitute`, `typeof`, `class`, and `methods` to see how the R interpreter parses expressions.

6 R Syntax

Every expression in R can be rewritten as a function call. However, R has some special syntax to write common operations like assignments, lookups, and numerical expressions more naturally. This chapter gives an overview of how to write valid R expressions. It's not intended to be a formal or complete description of all valid syntax in R, but just a readable description of how to write valid R expressions.[1]

Constants

Let's start by looking at constants. Constants are the basic building blocks for data objects in R: numbers, character values, and symbols.

Numeric Vectors

Numbers are interpreted literally in R:

```
> 1.1
[1] 1.1
> 2
[1] 2
> 2^1023
[1] 8.988466e+307
```

You may specify values in hexadecimal notation by prefixing them with `0x`:

```
> 0x1
[1] 1
> 0xFFFF
[1] 65535
```

1. You could write R code as a series of function calls with lots of function calls. This would look a lot like LISP code, with all the parentheses. Incidentally, the S language was inspired by LISP and uses many of the same data structures and evaluation techniques that are used by LISP interpreters.

By default, numbers in R expressions are interpreted as double-precision floating-point numbers, even when you enter simple integers:

```
> typeof(1)
[1] "double"
```

If you want an integer, you can use the sequence notation or the `as` function to obtain an integer:

```
> typeof(1:1)
[1] "integer"
> typeof(as(1, "integer"))
[1] "integer"
```

The sequence operator *`a`*`:`*`b`* will return a vector of integers between *`a`* and *`b`*. To combine an arbitrary set of numbers into a vector, use the `c` function:

```
> v <- c(173, 12, 1.12312, -93)
```

R allows a lot of flexibility when entering numbers. However, there is a limit to the size and precision of numbers that R can represent:

```
> # limits of precision
> (2^1023 + 1) == 2^1023
[1] TRUE
> # limits of size
> 2^1024
[1] Inf
```

In practice, this is rarely a problem. Most R users will load data from other sources on a computer (like a database) that also can't represent very large numbers.

R also supports complex numbers. Complex values are written as *`real_part`* `+`*`imaginary_part`*`i`. For example:

```
> 0+1i ^ 2
[1] -1+0i
> sqrt(-1+0i)
[1] 0+1i
> exp(0+1i * pi)
[1] -1+0i
```

Note that the `sqrt` function returns a value of the same type as its input; it will return the value `0+1i` when passed `-1+0i` but will return an `NaN` value when just passed the numeric value `-1`:

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

Character Vectors

A character object contains all the text between a pair of quotes. Most commonly, character objects are denoted by double quotes:

```
> "hello"
[1] "hello"
```

A character string may also be enclosed by single quotes:

```
> 'hello'
[1] "hello"
```

This can be convenient if the enclosed text contains double quotes (or vice versa). Equivalently, you may also escape the quotes by placing a backslash in front of each quote:

```
> identical("\"hello\"", '"hello"')
[1] TRUE

> identical('\'hello\'', "'hello'")
[1] TRUE
```

These examples are all vectors with only one element. To stitch together longer vectors, use the `c` function:

```
> numbers <- c("one", "two", "three", "four", "five")
> numbers
[1] "one"   "two"   "three" "four"  "five"
```

Symbols

An important class of constants is symbols. A symbol is an object in R that refers to another object; a symbol is the name of a variable in R. For example, let's assign the numeric value `1` to the symbol `x`:

```
> x <- 1
```

In this expression, `x` is a symbol. The statement `x <- 1` means "map the symbol `x` to the numeric value `1` in the current environment." (We'll discuss environments in Chapter 8.)

A symbol that begins with a character and contains other characters, numbers, periods, and underscores may be used directly in R statements. Here are a few examples of symbol names that can be typed without escape characters:

```
> x <- 1
> # case matters
> x1 <- 1
> X1 <- 2
> x1
[1] 1
> X1
[1] 2
> x1.1 <- 1
> x1.1_1 <- 1
```

Some symbols contain special syntax. In order to refer to these objects, you enclose them in backquotes. For example, to get help on the assignment operator (`<-`), you would use a command like this:

```
?`<-`
```

If you really wanted to, you could use backquotes to define a symbol that contains special characters or starts with a number:

```
> `1+2=3` <- "hello"
> `1+2=3`
[1] "hello"
```

Not all words are valid as symbols; some words are reserved in R. Specifically, you can't use `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_`, `...`, `..1`, `..2`, `..3`, `..4`, `..5`, `..6`, `..7`, `..8`, or `..9`.

You can redefine primitive functions that are not on this list. For example, when you start R, the symbol `c` normally refers to the primitive function `c`, which combines elements into vectors:

```
> c
function (..., recursive = FALSE)  .Primitive("c")
```

However, you can redefine the symbol `c` to point to something else:

```
> c <- 1
> c
[1] 1
```

Even after you redefine the symbol `c`, you can continue to use the "combine" function as before:

```
> v <- c(1, 2, 3)
> v
[1] 1 2 3
```

See Chapter 2 for more information on the combine function.

Operators

Many functions in R can be written as operators. An *operator* is a function that takes one or two arguments and can be written without parentheses.

One familiar set of operators is binary operators for arithmetic. R supports arithmetic operations:

```
> # addition
> 1 + 19
[1] 20

> # multiplication
> 5 * 4
[1] 20
```

R also includes notation for other mathematical operations, including moduli, exponents, and integer division:

```
> # modulus
> 41 %% 21
[1] 20?

> # exponents
> 20 ^ 1
[1] 20
```

```
> # integer division
> 21 %/% 2
[1] 10
```

You can define your own binary operators. User-defined binary operators consist of a string of characters between two % characters. To do this, create a function of two variables and assign it to an appropriate symbol. For example, let's define an operator `%myop%` that doubles each operand and then adds them together:

```
> `%myop%` <- function(a, b) {2*a + 2*b}
> 1 %myop% 1
[1] 4
> 1 %myop% 2
[1] 6
```

Some language constructs are also binary operators. For example, assignment, indexing, and function calls are binary operators:[2]

```
> # assignment is a binary operator
> # the left side is a symbol, the right is a value
> x <- c(1, 2, 3, 4, 5)

> # indexing is a binary operator too
> # the left side is a symbol, the right is an index
> x[3]
[1] 3

> # a function call is also a binary operator
> # the left side is a symbol pointing to the function argument
> # the right side are the arguments
> max(1, 2)
[1] 2
```

There are also unary operators that take only one variable. Here are two familiar examples:

```
> # negation is a unary operator
> -7
[1] -7

> # ? (for help) is also a unary operator
> ?`?`
```

Order of Operations

You may remember from high school math that you always evaluate mathematical expressions in a certain order. For example, when you evaluate the expression 1 + 2 • 5, you first multiply 2 and 5 and then add 1. The same thing is true in computer

2. Don't be confused by the closing bracket in an indexing operation or the closing parenthesis in a function call; although this syntax uses two symbols, both operations are still technically binary operators. For example, a function call has the form `f(arguments)`, where `f` is a function and *`arguments`* are the arguments for the function.

languages like R. When you enter an expression in R, the R interpreter will always evaluate some expressions first.

In order to resolve ambiguity, operators in R are always interpreted in the same order. Here is a summary of the precedence rules:

- Function calls and grouping expressions
- Index and lookup operators
- Arithmetic
- Comparison
- Formulas
- Assignment
- Help

Table 6-1 shows a complete list of operators in R and their precedence.

Table 6-1. Operator precedence, from the help(syntax) file

Operators (in order of priority)	Description
`( {`	Function calls and grouping expressions (respectively)
`[ [[`	Indexing
`:: :::`	Access variables in a namespace
`$ @`	Component / slot extraction
`^`	Exponentiation (right to left)
`- +`	*Unary* operators for minus and plus
`:`	Sequence operator
`%any%`	Special operators
`* /`	Multiply, divide
`+ -`	*Binary* operators for add, subtract
`< > <= >= == !=`	Ordering and comparison
`!`	Negation
`& &&`	And
`| ||`	Or
`~`	As in formulas
`-> ->>`	Rightward assignment
`=`	Assignment (right to left)
`<- <<-`	Assignment (right to left)
`?`	Help (unary and binary)

For a current list of built-in operators and their precedence, see the help file for `syntax`.

Assignments

Most assignments that we've seen so far simply assign an object to a symbol. For example:

```
> x <- 1
> y <- list(shoes="loafers", hat="Yankees cap", shirt="white")
> z <- function(a, b, c) {a ^ b / c}
> v <- c(1, 2, 3, 4, 5, 6, 7, 8)
```

There is an alternative type of assignment statement in R that acts differently: assignments with a function on the left-hand side of the assignment operator. These statements replace an object with a new object that has slightly different properties. Here are a few examples:

```
> dim(v) <- c(2, 4)

> v[2, 2] <- 10

> formals(z) <- alist(a=1, b=2, c=3)
```

There is a little bit of magic going on behind the scenes. An assignment statement of the form:

```
fun(sym) <- val
```

is really syntactic sugar for a function of the form:

```
`fun<-`(sym,val)
```

Each of these functions replaces the object associated with *sym* in the current environment. By convention, *fun* refers to a property of the object represented by *sym*. If you write a method with the name *method_name*<-, then R will allow you to place *method_name* on the left-hand side of an assignment statement.

Expressions

R provides different constructs for grouping together expressions: semicolons, parentheses, and curly braces.

Separating Expressions

You can write a series of expressions on separate lines:

```
> x <- 1
> y <- 2
> z <- 3
```

Alternatively, you can place them on the same line, separated by semicolons:

```
> x <- 1; y <- 2; z <- 3
```

Parentheses

The parentheses notation returns the result of evaluating the expression inside the parentheses:

```
(expression)
```

The operator has the same precedence as a function call. In fact, grouping a set of expressions inside parentheses is equivalent to evaluating a function of one argument that just returns its argument:

```
> 2 * (5 + 1)
[1] 12
> # equivalent expression
> f <- function (x) x
> 2 * f(5 + 1)
[1] 12
```

Grouping expressions with parentheses can be used to override the default order of operations. For example:

```
> 2 * 5 + 1
[1] 11
> 2 * (5 + 1)
[1] 12
```

Curly Braces

Curly braces are used to evaluate a series of expressions (separated by new lines or semicolons) and return only the last expression:

```
{expression_1; expression_2; ... expression_n}
```

Often, curly braces are used to group a set of operations in the body of a function:

```
> f <- function() {x <- 1; y <- 2; x + y}
> f()
[1] 3
```

However, curly braces can also be used as expressions in other contexts:

```
> {x <- 1; y <- 2; x + y}
[1] 3
```

The contents of the curly braces are evaluated inside the current environment; a new environment is created by a function call but *not* by the use of curly braces:

```
> # when evaluated in a function, u and v are assigned
> # only inside the function environment
> f <- function() {u <- 1; v <- 2; u + v}
> u
Error: object "u" not found
> v
Error: object "v" not found
> # when evaluated outside the function, u and v are
> # assigned in the current environment
> {u <- 1; v <- 2; u + v}
[1] 3
```

```
> u
[1] 1
> v
[1] 2
```

For more information about variable scope and environments, see Chapter 8.

The curly brace notation is translated internally as a call to the `` `{` `` function. (Note, however, that the arguments are not evaluated the same way as in a standard function.)

Control Structures

Nearly every operation in R can be written as a function, but it isn't always convenient to do so. Therefore, R provides special syntax that you can use in common program structures. We've already described two important sets of constructions: operators and grouping brackets. This section describes a few other key language structures and explains what they do.

Conditional Statements

Conditional statements take the form:

```
if (condition) true_expression else false_expression
```

or, alternatively:

```
if (condition) expression
```

Because the expressions *`expression`*, *`true_expression`*, and *`false_expression`* are not always evaluated, the function `if` has the type `special`:

```
> typeof(`if`)
[1] "special"
```

Here are a few examples of conditional statements:

```
> if (FALSE) "this will not be printed"
> if (FALSE) "this will not be printed" else "this will be printed"
[1] "this will be printed"
> if (is(x, "numeric")) x/2 else print("x is not numeric")
[1] 5
```

In R, conditional statements are not vector operations. If the *`condition`* statement is a vector of more than one `logical` value, then only the first item will be used. For example:

```
> x <- 10
> y <- c(8, 10, 12, 3, 17)
> if (x < y) x else y
[1]  8 10 12  3 17
Warning message:
In if (x < y) x else y :
  the condition has length > 1 and only the first element will be used
```

If you would like a vector operation, use the `ifelse` function instead:

```
> a <- c("a", "a", "a", "a", "a")
> b <- c("b", "b", "b", "b", "b")
> ifelse(c(TRUE, FALSE, TRUE, FALSE, TRUE), a, b)
[1] "a" "b" "a" "b" "a"
```

Often, it's convenient to return different values (or call different functions) depending on a single input value. You can code these as

```
> switcheroo.if.then <- function(x) {
+   if (x == "a")
+     "camel"
+   else if (x == "b")
+     "bear"
+   else if (x == "c")
+     "camel"
+   else
+     "moose"
+ }
```

but that is verbose. A better alternative is to use the `switch` function:

```
> switcheroo.switch <- function(x) {
+   switch(x,
+     a="alligator",
+     b="bear",
+     c="camel",
+     "moose")
+ }
```

The first argument is a character value to switch on, the named arguments specify what to do for each value of the argument, and an unnamed argument specifies the default value. As you can see, these two expressions are equivalent:

```
> switcheroo.if.then("a")
[1] "camel"
> switcheroo.if.then("f")
[1] "moose"
> switcheroo.switch("a")
[1] "camel"
> switcheroo.switch("f")
[1] "moose"
```

Loops

There are three different looping constructs in R. Simplest is `repeat`, which just repeats the same expression:

```
repeat expression
```

To stop repeating the expression, you can use the keyword `break`. To skip to the next iteration in a loop, you can use the command `next`.

As an example, the following R code prints out multiples of 5 up to 25:

```
> i <- 5
> repeat {if (i > 25) break else {print(i); i <- i + 5;}}
```

```
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

If you do not include a break command, the R code will be an infinite loop. (This can be useful for creating an interactive application.)

Another useful construction is while loops, which repeat an expression while a condition is true:

```
while (condition) expression
```

As a simple example, let's rewrite the example above using a while loop:

```
> i <- 5
> while (i <= 25) {print(i); i <- i + 5}
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

You can also use break and next inside while loops. The break statement is used to stop iterating through a loop. The next statement skips to the next loop iteration without evaluating the remaining expressions in the loop body.

Finally, R provides for loops, which iterate through each item in a vector (or a list):

```
for (var in list) expression
```

Let's use the same example for a for loop:

```
> for (i in seq(from=5, to=25, by=5)) print(i)
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

You can also use break and next inside for loops.

There are two important properties of looping statements to remember. First, results are not printed inside a loop unless you explicitly call the print function. For example:

```
> for (i in seq(from=5, to=25, by=5)) i
```

Second, the variable *var* that is set in a for loop is changed in the calling environment:

```
> i <- 1
> for (i in seq(from=5, to=25, by=5)) i
> i
[1] 25
```

Like conditional statements, the looping functions `` `repeat` ``, `` `while` ``, and `` `for` `` have type special, because *expression* is not necessarily evaluated.

Looping Extensions

If you've used modern programming languages like Java, you might be disappointed that R doesn't provide `iterators` or `foreach` loops. Luckily, these mechanisms are available through add-on packages. (These packages were written by Revolution Computing and are available through CRAN.)

Iterators are abstract objects that return elements from another object. Using iterators can help make code easier to understand. Additionally, iterators can make code easier to parallelize. To use iterators, you'll need to install the `iterators` package. Iterators can return elements of a vector, array, data frame, or other object. You can even use an iterator to return values returned by a function (such as a function that returns random values). To create an iterator in R, you would use the `iter` function:

```
iter(obj, checkFunc=function(...) TRUE, recycle=FALSE,...)
```

The argument `obj` specifies the object, `recycle` specifies whether the iterator should reset when it runs out of elements, and `checkFunc` specifies a function that filters values returned by the iterator.

You fetch the next item with the function `nextElem`. This function will implicitly call `checkFunc`. If the next value matches `checkFunc`, it will be returned. If it doesn't match, then the function will try another value. `nextElem` will continue checking values until it finds one that matches `checkFunc`, or it runs out of values. When there are no elements left, the iterator calls stop with the message "StopIteration."

For example, let's create an iterator that returns values between 1 and 5:

```
> library(iterators)
> onetofive <- iter(1:5)
> nextElem(onetofive)
[1] 1
> nextElem(onetofive)
[1] 2
> nextElem(onetofive)
[1] 3
> nextElem(onetofive)
[1] 4
> nextElem(onetofive)
[1] 5
> nextElem(onetofive)
Error: StopIteration
```

A second extension is the `foreach` loop, available through the `foreach` package. `Foreach` provides an elegant way to loop through multiple elements of another object (such as a vector, matrix, data frame, or iterator), evaluate an expression for each element, and return the results. Within the `foreach` function, you assign elements to a temporary value, just like in a `for` loop.

Here is the prototype for the `foreach` function:

```
foreach(..., .combine, .init, .final=NULL, .inorder=TRUE,
       .multicombine=FALSE,
       .maxcombine=if (.multicombine) 100 else 2,
       .errorhandling=c('stop', 'remove', 'pass'),
       .packages=NULL, .export=NULL, .noexport=NULL,
       .verbose=FALSE)
```

Technically, the `foreach` function returns a `foreach` object. To actually evaluate the loop, you need to apply the `foreach` loop to an R expression using the `%do%` or `%dopar%` operators. That sounds weird, but it's actually pretty easy to use in practice. For example, you can use a `foreach` loop to calculate the square roots of numbers between 1 and 5:

```
> sqrts.1to5 <- foreach(i=1:5) %do% sqrt(i)
> sqrts.1to5
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2

[[5]]
[1] 2.236068
```

The `%do%` operator evaluates the expression in serial, while the `%dopar%` can be used to evaluate expressions in parallel. For more about parallel computing with R, see Chapter 26.

Accessing Data Structures

R has some specialized syntax for accessing data structures. You can fetch a single item from a structure, or multiple items (possibly as a multidimensional array) using R's index notation. You can fetch items by location within a data structure or by name.

Data Structure Operators

Table 6-2 shows the operators in R used for accessing objects in a data structure.

Table 6-2. Data structure access notation

Syntax	Objects	Description
x[*i*]	Vectors, lists	Returns objects from object *x*, described by *i*. *i* may be an integer vector, character vector (of object names), or logical vector. Does not allow partial matches. When used with lists, returns a list. When used with vectors, returns a vector.
x[[*i*]]	Vectors, lists	Returns a single element of *x*, matching *i*. *i* may be an integer or character vector of length 1. Allows partial matches (with `exact=FALSE` option).
x$*n*	Lists	Returns object with name *n* from object *x*.
x@*n*	S4 objects	Returns element stored in object *x* in slot named *n*.

Although the single-bracket notation and double-bracket notation look very similar, there are three important differences. First, double brackets always return a single element, while single brackets may return multiple elements. Second, when elements are referred to by name (as opposed to by index), single brackets match only named objects exactly, while double brackets allow partial matches. Finally, when used with lists, the single-bracket notation returns a list, but the double-bracket notation returns a vector.

I'll explain how to use this notation below.

Indexing by Integer Vector

The most familiar way to look up an element in R is by numeric vector. As an example, let's create a very simple vector of 20 integers:

```
>  v <- 100:119
```

You can look up individual elements by position in the vector using the bracket notation *x*[*s*], where *x* is the vector from which you want to return elements and *s* is a second vector representing the set of element indices you would like to query. You can use an integer vector to look up a single element or multiple elements:

```
> v[5]
[1] 104
> v[1:5]
[1] 100 101 102 103 104
> v[c(1, 6, 11, 16)]
[1] 100 105 110 115
```

As a special case, you can use the double-bracket notation to reference a single element:

```
> v[[3]]
[1] 102
```

The double-bracket notation works the same as the single-bracket notation in this case; see "Indexing by Name" on page 79 for an explanation of references that do not work with the single-bracket notation.

You can also use negative integers to return a vector consisting of all elements except the specified elements:

```
> # exclude elements 1:15 (by specifying indexes -1 to -15)
> v[-15:-1]
[1] 115 116 117 118 119
```

The same notation applies to lists:

```
> l <- list(a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8, i=9, j=10)
> l[1:3]
$a
[1] 1

$b
[1] 2
```

```
$c
[1] 3

> l[-7:-1]
$h
[1] 8

$i
[1] 9

$j
[1] 10
```

You can also use this notation to extract parts of multidimensional data structures:

```
> m <- matrix(data=c(101:112), nrow=3, ncol=4)
> m
     [,1] [,2] [,3] [,4]
[1,]  101  104  107  110
[2,]  102  105  108  111
[3,]  103  106  109  112
> m[3]
[1] 103
> m[3,4]
[1] 112
> m[1:2,1:2]
     [,1] [,2]
[1,]  101  104
[2,]  102  105
```

If you omit a vector specifying a set of indices for a dimension, then elements for all indices are returned:

```
> m[1:2, ]
     [,1] [,2] [,3] [,4]
[1,]  101  104  107  110
[2,]  102  105  108  111
> m[3:4]
[1] 103 104
> m[, 3:4]
     [,1] [,2]
[1,]  107  110
[2,]  108  111
[3,]  109  112
```

When selecting a subset, R will automatically coerce the result to the most appropriate number of dimensions. If you select a subset of elements that corresponds to a matrix, R will return a matrix object; if you select a subset that corresponds to only a vector, R will return a vector object. To disable this behavior, you can use the `drop=FALSE` option:

```
> a <- array(data=c(101:124), dim=c(2, 3, 4))
> class(a[1, 1, ])
[1] "integer"
> class(a[1, , ])
[1] "matrix"
> class(a[1:2, 1:2, 1:2])
```

```
[1] "array"
> class(a[1, 1, 1, drop=FALSE])
[1] "array"
```

It is possible to create an array object with dimensions of length 1. However, when selecting subsets, R simplifies the returned objects.

It is also possible to replace elements in a vector, matrix, or array using the same notation:

```
> m[1] <- 1000
> m
     [,1] [,2] [,3] [,4]
[1,] 1000  104  107  110
[2,]  102  105  108  111
[3,]  103  106  109  112
> m[1:2, 1:2] <- matrix(c(1001:1004), nrow=2, ncol=2)
> m
     [,1] [,2] [,3] [,4]
[1,] 1001 1003  107  110
[2,] 1002 1004  108  111
[3,]  103  106  109  112
```

It is even possible to extend a data structure using this notation. A special `NA` element is used to represent values that are not defined:

```
> v <- 1:12
> v[15] <- 15
> v
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 NA NA 15
```

You can also index a data structure by a factor; the factor is interpreted as an integer vector.

Indexing by Logical Vector

As an alternative to indexing by an integer vector, you can also index through a logical vector. As a simple example, let's construct a vector of alternating true and false elements to apply to `v`:

```
> rep(c(TRUE, FALSE), 10)
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[12] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
> v[rep(c(TRUE, FALSE), 10)]
 [1] 100 102 104 106 108 110 112 114 116 118
```

Often, it is useful to calculate a logical vector from the vector itself:

```
> # trivial example: return element that is equal to 103
> v[(v==103)]
> # more interesting example: multiples of three
> v[(v %% 3 == 0)]
[1] 102 105 108 111 114 117
```

The index vector does not need to be the same length as the vector itself. R will repeat the shorter vector, returning matching values:

```
> v[c(TRUE, FALSE, FALSE)]
[1] 100 103 106 109 112 115 118
```

As above, the same notation applies to lists:

```
> l[(l > 7)]
$h
[1] 8

$i
[1] 9

$j
[1] 10
```

Indexing by Name

With lists, each element may be assigned a name. You can index an element by name using the $ notation:

```
> l <- list(a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8, i=9, j=10)
> l$j
[1] 10
```

You can also use the single-bracket notation to index a set of elements by name:

```
> l[c("a", "b", "c")]
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

You can also index by name using the double-bracket notation when selecting a single element. It is even possible to index by partial name using the `exact=FALSE` option:

```
> dairy <- list(milk="1 gallon", butter="1 pound", eggs=12)
> dairy$milk
[1] "1 gallon"
> dairy[["milk"]]
[1] "1 gallon"
> dairy[["mil"]]
NULL
> dairy[["mil",exact=FALSE]]
[1] "1 gallon"
```

Sometimes, an object is a list of lists. You can also use the double-bracket notation to reference an element in this type of data structure. To do this, use a vector as an argument. R will iterate through the elements in the vector, referencing sublists:

```
> fruit <- list(apples=6, oranges=3, bananas=10)
> shopping.list <- list (dairy=dairy, fruit=fruit)
> shopping.list
$dairy
```

```
$dairy$milk
[1] "1 gallon"

$dairy$butter
[1] "1 pound"

$dairy$eggs
[1] 12


$fruit
$fruit$apples
[1] 6

$fruit$oranges
[1] 3

$fruit$bananas
[1] 10


> shopping.list[[c("dairy", "milk")]]
[1] "1 gallon"
> shopping.list[[c(1,2)]]
[1] "1 pound"
```

R Code Style Standards

Standards for code style aren't the same as syntax, although they are sort of related. It is usually wise to be careful about code style to maximize the readability of your code, making it easier for you and others to maintain.

In this book, I've tried to stick to Google's R Style Guide, which is available at *http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html*. Here is a summary of its suggestions:

Indentation
: Indent lines with two spaces, not tabs. If code is inside parentheses, indent to the innermost parentheses.

Spacing
: Use only single spaces. Add spaces between binary operators and operands. Do not add spaces between a function name and the argument list. Add a single space between items in a list, after each comma.

Blocks
: Don't place an opening brace ("{") on its own line. Do place a closing brace ("}") on its own line. Indent inner blocks (by two spaces).

Semicolons
: Omit semicolons at the end of lines when they are optional.

Naming
: Name objects with lowercase words, separated by periods. For function names, capitalize the name of each word that is joined together, with no periods. Try to make function names verbs.

Assignment
: Use `<-`, not `=` for assignment statements.

Don't be confused by the object names. You don't have to name objects things like "field.goals" or "sanfrancisco.home.prices" or "top.bacon.searching.cities." It's just convention.