Section 2.1 describes how to read data into R variables and data frames. Section 2.2 explains how to work with data stored in data frames. Section 2.3 introduces matrices, higher-dimensional arrays, and lists. Section 2.4 explains how to manipulate character data in R. Section 2.5 discusses how to handle large data sets in R. Finally, Section 2.6 deals more abstractly with the organization of data in R, explaining the notions of classes, modes, and attributes, and describing the problems that can arise with floating-point calculations.

2.1 Data Input

Although there are many ways to read data into R, we will concentrate on the most important ones: typing data directly at the keyboard, reading data from a plain-text (also known as *ASCII*) file into an R data frame, importing data saved by another statistical package, importing data from a spreadsheet, and accessing data from one of R's many packages. In addition, we will explain how to generate certain kinds of patterned data. In Section 2.5.3, we will briefly discuss how to access data stored in a database management system.

2.1.1 KEYBOARD INPUT

Entering data directly into R using the keyboard can be useful in some circumstances. First, if the number of values is small, keyboard entry can be efficient. Second, as we will see, R has a number of functions for quickly generating patterned data. For large amounts of data, however, keyboard entry is likely to be inefficient and error-prone.

We saw in Chapter 1 how to use the `c` (combine) function to enter a vector of numbers:

```
> (x <- c(1, 2, 3, 4))

[1] 1 2 3 4
```

Recall that enclosing the command in parentheses causes the value of `x` to be printed. The same procedure works for vectors of other types, such as character data or logical data:

```
> (names <- c("John", "Sandy", 'Mary'))

[1] "John"  "Sandy" "Mary"

> (v <- c(TRUE, FALSE))

[1]  TRUE FALSE
```

Character strings may be input between single or double quotation marks: for example, `'John'` and `"John"` are equivalent. When R prints character

Table 2.1 Data from an experiment on anonymity and cooperation by Fox and Guyer (1978). The data consist of the number of cooperative choices made out of 120 total choices.

	Sex	
Condition	Male	Female
Public Choice	49	54
	64	61
	37	79
	52	64
	68	29
Anonymous	27	40
	58	39
	52	44
	41	34
	30	44

strings, it encloses them within double quotes, regardless of whether single or double quotes were used to enter the strings.

Entering data in this manner works well for very short vectors. You may have noticed in some of the previous examples that when an R statement is continued on additional lines, the > (greater than) prompt is replaced by the interpreter with the + (plus) prompt on the continuation lines. R recognizes that a line is to be continued when it is syntactically incomplete—for example, when a left parenthesis needs to be balanced by a right parenthesis or when the right argument to a binary operator, such as * (multiplication), has not yet been entered. Consequently, entries using `c` may be continued over several lines simply by omitting the terminal right parenthesis until the data are complete. It may be more convenient, however, to use the `scan` function, to be illustrated shortly, which prompts with the index of the next entry.

Consider the data in Table 2.1. The data are from an experiment (Fox and Guyer, 1978) in which each of 20 four-person groups of subjects played 30 trials of a prisoners' dilemma game in which subjects could make either cooperative or competitive choices. Half the groups were composed of women and half of men. Half the groups of each sex were randomly assigned to a public-choice condition, in which the choices of all the individuals were made known to the group after each trial, and the other groups were assigned to an anonymous-choice condition, in which only the aggregated choices were revealed. The data in the table give the number of cooperative choices made in each group, out of $30 \times 4 = 120$ choices in all.

To enter the number of cooperative choices as a vector, we could use the `c` function, typing the data values separated by commas, but instead we will illustrate the use of the `scan` function:

```
> (cooperation <- scan())
1:  49 64 37 52 68 54
7:  61 79 64 29
11:  27 58 52 41 30 40 39
18:  44 34 44
21:

Read 20 items
 [1] 49 64 37 52 68 54 61 79 64 29 27 58 52 41 30 40 39 44 34 44
```

The number before the colon on each input line is the index of the next observation to be entered and is supplied by `scan`; entering a blank line terminates `scan`. We entered the data for the Male, Public-Choice treatment first, followed by the data for the Female, Public-Choice treatment, and so on.

We could enter the condition and sex of each group in a similar manner, but because the data are patterned, it is more economical to use the `rep` (replicate) function. The first argument to `rep` specifies the data to be repeated; the second argument specifies the number of repetitions:

```
> rep(5, 3)

[1] 5 5 5

> rep(c(1, 2, 3), 2)

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

When the first argument to `rep` is a vector, the second argument can be a vector of the same length, specifying the number of times each entry of the first argument is to be repeated:

```
> rep(1:3, 3:1)

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

In the current context, we may proceed as follows:

```
> (condition <- rep(c("public", "anonymous"), c(10, 10)))

 [1] "public"    "public"    "public"    "public"    "public"
 [6] "public"    "public"    "public"    "public"    "public"
[11] "anonymous" "anonymous" "anonymous" "anonymous" "anonymous"
[16] "anonymous" "anonymous" "anonymous" "anonymous" "anonymous"
```

Thus, `condition` is formed by repeating each element of `c("public", "anonymous")` 10 times.

```
> (sex <- rep(rep(c("male", "female"), c(5, 5)), 2))

 [1] "male"   "male"   "male"   "male"   "male"   "female"
 [7] "female" "female" "female" "female" "male"   "male"
[13] "male"   "male"   "male"   "female" "female" "female"
[19] "female" "female"
```