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"
```

The vector `sex` requires using `rep` twice, first to generate five `"male"` character strings followed by five `"female"` character strings, and then to repeat this pattern twice to get all 20 values.

Finally, it is convenient to put the three variables together in a data frame:

```
> (Guyer <- data.frame(cooperation, condition, sex))

   cooperation condition    sex
1           49    public   male
2           64    public   male
3           37    public   male
. . .
19          34 anonymous female
20          44 anonymous female
```

The original variables `condition` and `sex` are character vectors. When vectors of character strings are put into a data frame, they are converted by default into factors, which is almost always appropriate for subsequent statistical analysis of the data. (The important distinction between character vectors and factors is discussed in Section 2.2.4.)

R has a bare-bones, spreadsheet-like data editor that may be used to enter, examine, and modify data frames. We find this editor useful primarily for modifying individual values—for example, to fix an error in the data. If you prefer to enter data in a spreadsheet-like environment, we suggest using one of the popular and more general spreadsheet programs such as Excel or OpenOffice Calc, and then importing your data into R.

To enter data into a *new* data frame using the editor, we may type the following:

```
> Guyer <- edit(as.data.frame(NULL))
```

This command opens the data editor, into which we may type variable names and data values. An *existing* data frame can be edited with the `fix` function, as in `fix(Guyer)`.

The `fix` function can also be used to examine an existing data frame, but the `View` function is safer and more convenient: safer because `View` cannot modify the data, and more convenient because the `View` spreadsheet window can remain open while we continue to work at the R command prompt. In contrast, the R interpreter waits for `fix` or `edit` to terminate before returning control to the command prompt.

2.1.2 FILE INPUT TO A DATA FRAME

DELIMITED DATA IN A LOCAL FILE

The previous example shows how to construct a data frame from preexisting variables. More frequently, we read data from a plain-text (ASCII) file into an R data frame using the `read.table` function. We assume that the input file is organized in the following manner:

- The first line of the file gives the names of the variables separated by white space consisting of one or more blanks or tabs; these names are valid R variable names, and in particular must not contain blanks. If the first entry in each line of the data file is to provide row names for the data frame, then there is one fewer variable name than columns of data; otherwise, there is one variable name for each column.
- Each subsequent line contains data for one observation or case, with the data values separated by white space. The data values need not appear in the same place in each line as long as the *number* of values and their *order* are the same in all lines. Character data either contain no embedded blanks (our preference) or are enclosed in single or double quotes. Thus, for example, `white.collar`, `'white collar'`, and `"white collar"` are valid character data values, but `white collar` without quotes is not acceptable and will be interpreted as two separate values, `white` and `collar`. Character and logical data are converted to factors on input. You may avoid this conversion by specifying the argument `as.is=TRUE` to `read.table`, but representing categorical data as factors is generally desirable.
- Many spreadsheet programs and other programs create plain-text files with data values separated by commas—so-called *comma-delimited* or *comma-separated* files. Supplying the argument `sep=","` to `read.table` accommodates this form of data. Alternatively, the function `read.csv`, a convenience interface to `read.table` that sets `header=TRUE` and `sep=","` by default, may be used to read comma-separated data files. In comma-delimited data, blanks *may* be included in unquoted character strings, but commas may not be included.
- Missing data appear explicitly, preferably encoded by the characters `NA` (Not Available); in particular, missing data are *not* left blank. There is, therefore, the same number of data values in each line of the input file, even if some of the values represent missing data. In a comma-separated file, however, missing values *may* be left blank. If characters other than `NA` are used to encode missing data, and if it is inconvenient to replace them in an editor, then you may specify the missing-data code in the `na.strings` argument to `read.table`. For example, both SAS and SPSS recognize the period (`.`) as an input missing-data indicator; to read a file with periods encoding missing data, use `na.strings="."`. Different missing-data codes can be supplied for different variables by specifying `na.strings` as a vector. For more details, see the online documentation for `read.table`.

This specification is more rigid than it needs to be, but it is clear and usually is easy to satisfy. Most spreadsheet, database, and statistical programs are capable of producing plain-text files of this format, or produce files that can be put in this form with minimal editing. Use a plain-text editor (such as Windows Notepad or a programming editor) to edit data files. If you use a word-processing program (such as Word or OpenOffice Writer), be careful

to save the file as a plain-text file; `read.table` cannot read data saved in the default formats used by word-processing programs.

We use the data in the file Prestige.txt to illustrate.[1] This data set, with occupations as observations, is similar to the Duncan occupational-prestige data employed as an example in the previous chapter. Here are a few lines of the data file (recall that the ellipses represent omitted lines—there are 102 occupations in all):

```
                    education income women prestige census type
gov.administrators      13.11  12351 11.16     68.8   1113 prof
general.managers        12.26  25879  4.02     69.1   1130 prof
accountants             12.77   9271 15.70     63.4   1171 prof
. . .
commercial.artists      11.09   6197 21.03     57.2   3314 prof
radio.tv.announcers     12.71   7562 11.15     57.6   3337   wc
athletes                11.44   8206  8.13     54.1   3373   NA
secretaries             11.59   4036 97.51     46.0   4111   wc
. . .
elevator.operators       7.58   3582 30.08     20.1   6193   bc
farmers                  6.84   3643  3.60     44.1   7112   NA
farm.workers             8.60   1656 27.75     21.5   7182   bc
rotary.well.drillers     8.88   6860  0.00     35.3   7711   bc
```

The variables in the data file are defined as follows:

- `education`: The average number of years of education for occupational incumbents in the 1971 Census of Canada.
- `income`: The average income of occupational incumbents, in dollars, in the 1971 Census.
- `women`: The percentage of occupational incumbents in the 1971 Census who were women.
- `prestige`: The average prestige rating for the occupation obtained in a sample survey conducted in Canada in 1966.
- `census`: The code of the occupation in the standard 1971 Census occupational classification.
- `type`: Professional and managerial (`prof`), white collar (`wc`), blue collar (`bc`), or missing (`NA`).

To read the data into R, we enter:

```
> (Prestige <- read.table("D:/data/Prestige.txt", header=TRUE))

                    education income women prestige census type
gov.administrators      13.11  12351 11.16     68.8   1113 prof
general.managers        12.26  25879  4.02     69.1   1130 prof
accountants             12.77   9271 15.70     63.4   1171 prof
. . .
commercial.artists      11.09   6197 21.03     57.2   3314 prof
radio.tv.announcers     12.71   7562 11.15     57.6   3337   wc
```

[1] You can download this file and other data files referenced in this chapter from the website for the book, most conveniently with the `carWeb` function—see `?carWeb`. The data sets are also available, with the same names, in the **car** package.

```
athletes                 11.44   8206  8.13     54.1   3373   NA
secretaries              11.59   4036 97.51     46.0   4111   wc
. . .
elevator.operators        7.58   3582 30.08     20.1   6193   bc
farmers                   6.84   3643  3.60     44.1   7112   NA
farm.workers              8.60   1656 27.75     21.5   7182   bc
rotary.well.drillers      8.88   6860  0.00     35.3   7711   bc
```

The first argument to `read.table` specifies the location of the input file, in this case the location in our local file system where we placed Prestige.txt. As we will see, with an active Internet connection, it is also possible to read a data file from a URL (web address). We suggest naming the data frame for the file from which the data were read and to begin the name of the data frame with an uppercase letter: `Prestige`.

Even though we are using R on a Windows system, the directories in the file system are separated by a / (forward slash) rather than by the standard Windows \ (back slash), because the back slash serves as a so-called *escape* character in an R character string, indicating that the next character has a special meaning: For example, `\n` represents a new-line character (i.e., go to the beginning of the next line), while `\t` is the tab character. Such special characters can be useful in creating printed output. A (single) back slash may be entered in a character string as `\\`.

You can avoid having to specify the full path to a data file if you first tell R where to look for data by specifying the *working directory*. For example, `setwd("D:/data")` sets the working directory to D:\data, and the command `read.table("mydata.txt")` would then look for the file mydata.txt in the D:\data directory. On Windows or Mac OS X, the command `setwd(choose.dir())` allows the user to select the working directory interactively.[2] The command `getwd()` displays the working directory.

Under Windows or Mac OS X, you can also browse the file system to find the file you want using the `file.choose` function:

```
> Prestige <- read.table(file.choose(), header=TRUE)
```

The `file.choose` function returns the path to the selected file as a character string, which is then passed to `read.table`.

The second argument, `header=TRUE`, tells `read.table` that the first line in the file contains variable names. It is not necessary to specify `header=TRUE` when, as here, the initial variable-names line has one fewer entry than the data lines that follow. The first entry on each data line in the file is an observation label. Nevertheless, it does not hurt to specify `header=TRUE`, and getting into the habit of doing so will save you trouble when you read a file with variable names but *without* row names.

[2] Alternatively, you can use the menus to select *File* → *Change dir* under Windows or *Misc* → *Change Working Directory* under Mac OS X.

DATA FILES FROM THE INTERNET

If you have an active Internet connection, files can also conveniently be read from their URLs. For example, we can read the Canadian occupational-prestige data set from the website for the book:[3]

```
> Prestige <- read.table(
+     "http://socserv.socsci.mcmaster.ca/jfox/books/Companion/data/
          Prestige.txt",
+     header=TRUE)
```

DEBUGGING DATA FILES

Not all data files can be read on the first try. Here are some simple steps to follow to correct a problematic data file.

Whenever you read a file for the first time, it is a good idea to run a few checks to make sure the file doesn't have any problems. The `summary` function can be helpful for this purpose. For the `Prestige` data frame:

```
> summary(Prestige)
```

```
   education          income          women
 Min.   : 6.380   Min.   :  611   Min.   : 0.000
 1st Qu.: 8.445   1st Qu.: 4106   1st Qu.: 3.592
 Median :10.540   Median : 5930   Median :13.600
 Mean   :10.738   Mean   : 6798   Mean   :28.979
 3rd Qu.:12.648   3rd Qu.: 8187   3rd Qu.:52.203
 Max.   :15.970   Max.   :25879   Max.   :97.510
    prestige         census       type
 Min.   :14.80   Min.   :1113   bc  :44
 1st Qu.:35.23   1st Qu.:3120   prof:31
 Median :43.60   Median :5135   wc  :23
 Mean   :46.83   Mean   :5402   NA's: 4
 3rd Qu.:59.27   3rd Qu.:8312
 Max.   :87.20   Max.   :9517
```

From this output, we see that all the variables except `type` are numeric, because numeric summaries are reported, while `type` is a factor with levels `bc`, `wc`, and `prof`; furthermore, `type` has four missing values, given by the `NA`s.

If we had made an error in entering a numeric value—for example, typing the letter `l` ("ell") instead of the numeral `1` (one), or the letter `O` ("oh") instead of the numeral `0` (zero)—then the corresponding variable would have been made into a factor with many levels rather than a numeric variable, and we would have to correct the data in order to read them properly. Similarly, numbers with embedded commas, such as `2,000,000`, are treated as character values rather than as numeric data, and so the corresponding variables would be incorrectly read as factors. Finally, if a value of `type` were entered incorrectly as `BC` rather than `bc`, then the `type` factor would acquire the additional level `BC`.

[3]The character string specifying the URL for the file `Prestige.txt` is broken across two lines to fit on the page but in fact must be given as one long string. Similar line breaks appear later in the chapter.

In some instances, an error in the data will prevent a file from being read at all, a problem that is usually caused by having too many or too few values in a line of the data file. Too many values can be produced by embedded blanks: For example, the characters - 2.3 in an input line, with a space between the minus sign and 2.3, would be read as two values, - and 2.3, not as the intended negative number −2.3. Too few items can be produced by coding missing data with blanks rather than with a proper missing-value indicator such as NA.

To simulate too few values in input lines, we prepared a version of the Prestige.txt data file in which NAs were erroneously replaced by blanks. Trying to read this file yields the following result:

```
> file <-
+ "http://socserv.socsci.mcmaster.ca/jfox/books/Companion/data/
      Prestige-bugged.txt"
> Prestige <- read.table(file, header=TRUE)

Error in scan(file = file, what = what, sep = sep, quote = quote, skip = 0,  :
line 34 did not have 7 elements
```

Because of the error, the data frame `Prestige` has not been created, and the error message tells us that at least one line in the file has the wrong number of elements. We can use the `count.fields` function to discover whether there are other errors as well, and, if there are, to determine their location:

```
> (counts <- count.fields(file))

[1] 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
[31] 7 7 7 7 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 6 7 7 7 7 7 7
[61] 7 7 7 6 7 7 7 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
[91] 7 7 7 7 7 7 7 7 7 7 7 7 7

> which(counts != 7)

[1] 35 54 64 68
```

Once we know the location of the errors, it is simple to fix the input file in a text editor that keeps track of line numbers. Because the data file has column names in its first line, the 34*th* data line is the 35*th* line in the file.

FIXED-FORMAT DATA

Although they are no longer common, you may encounter *fixed-format* data files, in which the data values are not separated by white space or commas and in which variables appear in fixed positions within each line of the file. To illustrate the process of reading this kind of data, we have created a fixed-format version of the Canadian occupational-prestige data set, which we placed in the file Prestige-fixed.txt. The file looks like this:

```
gov.administrators      13.111235111.1668.81113prof
general.managers        12.2625879 4.0269.11130prof
```

```
accountants                 12.77 927115.7063.41171prof
. . .
typesetters                 10.00 646213.5842.29511bc
bookbinders                  8.55 361770.8735.29517bc
```

The first 25 characters in each line are reserved for the `occupation` name, the next five spaces for the `education` value, the next five for `income`, and so on. Most of the data values run together, making the file difficult to decipher. If you have a choice, fixed-format input is best avoided. The `read.fwf` (`read` `fixed-width-format`) function can be used to read this file into a data frame:

```
> file <-
+ "http://socserv.socsci.mcmaster.ca/jfox/books/Companion/data/
        Prestige-fixed.txt"
> Prestige <- read.fwf(file,
+     col.names=c("occupation", "education", "income", "women",
+         "prestige", "census", "type"),
+     row.names="occupation",
+     widths=c(25, 5, 5, 5, 4, 4, 4))
```

Our sample file does not have an initial line with variable names, and so the default value of `header=FALSE` is appropriate. The `col.names` argument to `read.fwf` supplies names for the variables, and the argument `row.names` indicates that the variable `occupation` should be used to define row names. The `widths` argument gives the *field width* of each variable in the input file.

2.1.3 IMPORTING DATA

DATA FROM OTHER STATISTICAL PACKAGES

You are likely to encounter data sets that have been prepared in another statistical system, such as SAS or SPSS. If you have access to the other program, then exporting the data as a plain-text (ASCII) file for reading into R is generally quite straightforward. Alternatively, R provides facilities for *importing* data from other programs through functions in the **foreign** package, which is part of the standard R distribution. Currently, functions are available for importing data files from S version 3, SAS, SPSS, Minitab, Stata, and others. For example, the function `read.spss` is used to import SPSS data sets.

DATA FROM A SPREADSHEET

Raw data in a spreadsheet are probably the most common form of data that you will encounter. We can recommend three approaches to reading data from a spreadsheet. The simplest procedure may be to use the spreadsheet program to save the file in a format that can be read by `read.table`, usually a plain-text file with either white-space-delimited or comma-separated values. For this process to work, the data you want must be in the top-most worksheet of the spreadsheet, and the worksheet should look like a plain-text data file,

with optional variable names in the first row, optional row names in the first column, and the row-by-column matrix of data values filled in completely. No extraneous information, such as variable labels requiring two rows, should be included.

The second approach is to copy the data you want to the clipboard and then to use an R function such as `read.table` to read the data from the clipboard. Select the data to be read in the spreadsheet program (e.g., by left-clicking and dragging the mouse over the data); the first row of the blocked data can contain variable names and the first column can contain row names. Next, copy the selected data to the clipboard (e.g., in Windows by the key combination Control-c and in Mac OS X by command-c). Finally, switch to R, and enter a command such as `Data <- read.table("clipboard", header=TRUE)` to read the data into the data frame `Data`. If the first selected row in the spreadsheet doesn't contain variable names, omit the argument `header=TRUE`.

On Windows, the **RODBC** package[4] can be used to read spreadsheets in Excel format directly. For example, the file Datasets.xls on the website for the book is an Excel file containing two worksheets: `Duncan`, with Duncan's occupational-prestige data, and `Prestige`, with the Canadian occupational-prestige data. We placed this file in D:\data\Datasets.xls. To read the `Prestige` data into R,

```
> library(RODBC)
> channel <- odbcConnectExcel("D:/data/Datasets.xls")
> Prestige <- sqlQuery(channel, "select * from [Prestige$]")
> odbcClose(channel)
> head(Prestige) # first 6 rows

                       F1 education income women prestige census type
gov.administrators      13.11  12351 11.16     68.8   1113 prof
general.managers        12.26  25879  4.02     69.1   1130 prof
accountants             12.77   9271 15.70     63.4   1171 prof
purchasing.officers     11.42   8865  9.11     56.8   1175 prof
chemists                14.62   8403 11.68     73.5   2111 prof
physicists              15.64  11030  5.13     77.6   2113 prof
```

Recall that everything to the right of the # (pound sign) is a comment and is ignored by the R interpreter. For Excel 2007 files, use `odbcConnectExcel2007` in place of `odbcConnectExcel`.

The variable name `"F1"` was supplied automatically for the first column of the spreadsheet. We prefer to use this column to provide row names for the `Prestige` data frame:

```
> rownames(Prestige) <- Prestige$F1
> Prestige$F1 <- NULL  # remove F1 from Prestige
> head(Prestige)
```

[4] You can download and install this package from CRAN in the usual manner with the command `install.packages("RODBC")` or via the *Packages* menu. The **RODBC** package can also be made to work on other operating systems but not as conveniently as under Windows.

```
                    education income women prestige census type
gov.administrators      13.11  12351 11.16     68.8   1113 prof
general.managers        12.26  25879  4.02     69.1   1130 prof
accountants             12.77   9271 15.70     63.4   1171 prof
purchasing.officers     11.42   8865  9.11     56.8   1175 prof
chemists                14.62   8403 11.68     73.5   2111 prof
physicists              15.64  11030  5.13     77.6   2113 prof

> remove(Prestige)  # clean up
```

The **RODBC** package can also be used to connect R to a database management system (a topic that we will discuss briefly in Section 2.5.3).

2.1.4 ACCESSING DATA IN R PACKAGES

Many R packages, including the **car** and **alr3** packages, contain data sets. R provides two mechanisms for accessing the data in packages:

1. A data set can be read into the *global environment* (i.e., working memory) via the `data` command.[5] For example, to read the `Prestige` data frame from the **car** package into memory:

```
> data(Prestige, package="car")
> head(Prestige)  # first 6 rows

                    education income women prestige census type
gov.administrators      13.11  12351 11.16     68.8   1113 prof
general.managers        12.26  25879  4.02     69.1   1130 prof
accountants             12.77   9271 15.70     63.4   1171 prof
purchasing.officers     11.42   8865  9.11     56.8   1175 prof
chemists                14.62   8403 11.68     73.5   2111 prof
physicists              15.64  11030  5.13     77.6   2113 prof

> remove(Prestige)  # clean up
```

   Had the **car** package already been loaded, the `package` argument to `data` could have been omitted.

2. Many packages allow data sets to be accessed directly, via the so-called *lazy-data* mechanism, without explicitly reading the data into memory. For example,

```
library(car)

. . .

> head(Prestige)

                    education income women prestige census type
gov.administrators      13.11  12351 11.16     68.8   1113 prof
general.managers        12.26  25879  4.02     69.1   1130 prof
accountants             12.77   9271 15.70     63.4   1171 prof
purchasing.officers     11.42   8865  9.11     56.8   1175 prof
chemists                14.62   8403 11.68     73.5   2111 prof
physicists              15.64  11030  5.13     77.6   2113 prof
```

[5] Environments in R are discussed in Section 8.9.1.

CLEANING UP

We have defined several variables in the course of this section, some of which are no longer needed, so it is time to clean up:

```
> objects()

[1] "channel"     "condition"   "cooperation" "Guyer"
[5] "names"       "sex"         "v"           "x"

> remove(channel, names, v, x)
> detach(package:RODBC)
```

We have retained the data frame `Guyer` and the vectors `condition`, `cooperation`, and `sex` for subsequent illustrations in this chapter. Because we are finished with the **RODBC** package, we have detached it.

2.1.5 GETTING DATA *OUT* OF R

We hope and expect that you will rarely have to get your data out of R to use with another program, but doing so is nevertheless quite straightforward. As in the case of reading data, there are many ways to proceed, but a particularly simple approach is to use the `write.table` or `write.csv` function to output a data frame to a plain-text file. The syntax for `write.table` is essentially the reverse of that for `read.table`. For example, the following command writes the `Duncan` data frame (from the attached **car** package) to a file:

```
> write.table(Duncan, "c:/temp/Duncan.txt")
```

By default, row labels and variable names are included in the file, data values are separated by blanks, and all character strings are in quotes, whether or not they contain blanks. This default behavior can be changed—see `?write.table`.

The **foreign** package also includes some functions for exporting R data to a variety of file formats: Consult the documentation for the **foreign** package, `help(package="foreign")`.

2.2 Working With Data Frames

It is perfectly possible in R to analyze data stored in vectors, but we generally prefer to begin with a data frame, typically read from a file via the `read.table` function or accessed from an R package. Almost all the examples in this *Companion* use data frames from the **car** package.

In many statistical packages, such as SPSS, a single data set is active at any given time; in other packages, such as SAS, individual statistical procedures typically draw their data from a single source, which by default in SAS is the