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

last data set created. This is not the case in R, where data may be used simultaneously from several sources, providing flexibility but with the possibility of interference and confusion.

There are essentially two ways to work with data in a data frame, both of which we will explain in this section:

1. Attach the data frame to the search path via the `attach` command, making the variables in the data frame directly visible to the R interpreter.
2. Access the variables in the data frame as they are required without attaching the data frame.

New users of R generally prefer the first approach, probably because it is superficially similar to other statistical software in which commands reference an *active* data set. For reasons that we will explain, however, experienced users of R usually prefer *not* to attach data frames to the search path.

2.2.1 THE SEARCH PATH

When you type the name of a variable in a command, the R interpreter looks for an object of that name in the locations specified by the *search path*. We attach the `Duncan` data frame to the search path with the `attach` function and then use the `search` function to view the current path:

```
> attach(Duncan)
> search()

 [1] ".GlobalEnv"        "Duncan"            "package:car"
 [4] "package:survival"  "package:splines"   "package:leaps"
 [7] "package:nnet"      "package:MASS"      "package:stats"
[10] "package:graphics"  "package:grDevices" "package:utils"
[13] "package:datasets"  "package:methods"   "Autoloads"
[16] "package:base"
```

Now if we type the name of the variable `prestige` at the command prompt, R will look first in the global environment (`.GlobalEnv`), the region of memory in which R stores working data. If no variable named `prestige` is found in the global environment, then the data frame `Duncan` will be searched, because it was placed by the `attach` command in the second position on the search list. There is in fact no variable named `prestige` in the working data, but there is a variable by this name in the `Duncan` data frame, and so when we type `prestige`, we retrieve the `prestige` variable from `Duncan`, as we may readily verify:

```
> prestige

 [1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20  7  3 16
[41]  6 11  8 41 10
```

```
> Duncan$prestige

 [1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20  7  3 16
[41]  6 11  8 41 10
```

Typing `Duncan$prestige` directly extracts the column named `prestige` from the `Duncan` data frame.[6]

Had `prestige` not been found in `Duncan`, then the sequence of attached packages would have been searched in the order shown, followed by a special list of objects (`Autoloads`) that are loaded automatically as needed (and which we will subsequently ignore), and finally the R **base** package. The packages in the search path shown above, beginning with the **stats** package, are part of the basic R system and are loaded by default when R starts up.

Suppose, now, that we attach the `Prestige` data frame to the search path. The default behavior of the `attach` function is to attach a data frame in the *second* position on the search path, after the global environment:

```
> attach(Prestige)

        The following object(s) are masked from Duncan :

         education income prestige type


        The following object(s) are masked from package:datasets :

         women

> search()

 [1] ".GlobalEnv"        "Prestige"          "Duncan"
 [4] "package:car"       "package:survival"  "package:splines"
 [7] "package:leaps"     "package:nnet"      "package:MASS"
[10] "package:stats"     "package:graphics"  "package:grDevices"
[13] "package:utils"     "package:datasets"  "package:methods"
[16] "Autoloads"         "package:base"
```

Consequently, the data frame `Prestige` is attached *before* the data frame `Duncan`; and if we now simply type `prestige`, then the `prestige` variable in `Prestige` will be located *before* the `prestige` variable in `Duncan` is encountered:

```
> prestige

  [1] 68.8 69.1 63.4 56.8 73.5 77.6 72.6 78.1 73.1 68.8
 [11] 62.0 60.0 53.8 62.2 74.9 55.1 82.3 58.1 58.3 72.8
 [21] 84.6 59.6 66.1 87.2 66.7 68.4 64.7 34.9 72.1 69.3
 . . .
 [91] 38.9 36.2 29.9 42.9 26.5 66.1 48.9 35.9 25.1 26.1
[101] 42.2 35.2
```

[6] Information on indexing data frames is presented in Section 2.3.4.

The `prestige` variable in `Duncan` is still there—it is just being *shadowed* or *masked* (i.e., hidden) by `prestige` in `Prestige`, as the `attach` command warned us:

```
> Duncan$prestige

 [1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20  7  3 16
[41]  6 11  8 41 10
```

Because the variables in one data frame can shadow the variables in another, attaching more than one data frame at a time can lead to unanticipated problems and should generally be avoided. You can remove a data frame from the search path with the `detach` command:

```
> detach(Prestige)
> search()

 [1] ".GlobalEnv"        "Duncan"            "package:car"
 [4] "package:survival"  "package:splines"   "package:leaps"
 [7] "package:nnet"      "package:MASS"      "package:stats"
[10] "package:graphics"  "package:grDevices" "package:utils"
[13] "package:datasets"  "package:methods"   "Autoloads"
[16] "package:base"
```

Calling `detach` with no arguments detaches the second entry in the search path and, thus, produces the same effect as `detach(Prestige)`.

Now that `Prestige` has been detached, `prestige` again refers to the variable by that name in the `Duncan` data frame:

```
> prestige

 [1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20  7  3 16
[41]  6 11  8 41 10
```

The working data are the first item in the search path, and so globally defined variables shadow variables with the same names anywhere else along the path. This is why we use an uppercase letter at the beginning of the name of a data frame. Had we, for example, named the data frame `prestige` rather than `Prestige`, then the variable `prestige` within the data frame would have been shadowed by the data frame itself. To access the variable would then require a potentially confusing expression, such as `prestige$prestige`.

Our focus here is on manipulating data, but it is worth mentioning that R locates functions in the same way that it locates data. Consequently, functions earlier on the path can shadow functions of the same name later on the path.

In Section 1.1.3, we defined a function called `myMean`, avoiding the name `mean` so that the `mean` function in the **base** package would not be shadowed. The **base** function `mean` can calculate trimmed means as well as the ordinary arithmetic mean; for example,

```
> mean(prestige)

[1] 47.68889

> mean(prestige, trim=0.1)

[1] 47.2973
```

Specifying `mean(prestige, trim=0.1)` removes the top and bottom 10% of the data, calculating the mean of the middle 80% of observations. Trimmed means provide more efficient estimates of the center of a heavy-tailed distribution—for example, when outliers are present; in this example, however, trimming makes little difference.

Suppose that we define our own `mean` function, making no provision for trimming:

```
> mean <- function(x){
+     warning("the mean function in the base package is shadowed")
+     sum(x)/length(x)
+ }
```

The first line in our `mean` function prints a warning message. The purpose of the warning is simply to verify that our function executes in place of the `mean` function in the **base** package. Had we *carelessly* shadowed the standard `mean` function, we would not have politely provided a warning:

```
> mean(prestige)

[1] 47.68889
Warning message:
In mean(prestige):the mean function in the base package is shadowed
```

The essential point here is that because our `mean` function resides in the global environment, it is encountered on the search path *before* the `mean` function in the **base** package. Shadowing the standard `mean` function is inconsequential as long as our function is equivalent; but if, for example, we try to calculate a trimmed mean, our function does not work:

```
> mean(prestige, trim=0.1)

Error in mean(prestige, trim = 0.1):unused argument(s)(trim = 0.1)
```

Shadowing standard R functions is a practice generally to be avoided. Suppose, for example, that a robust-regression function tries to calculate a trimmed mean, but fails because the standard `mean` function is shadowed by our redefined `mean` function. If we are not conscious of this problem, the resulting error message may prove cryptic.

We can, however, use the same name for a variable and a function, as long as the two do not reside in the working data. Consider the following example:

```
> mean <- mean(prestige)  # uses then overwrites our mean function
> mean

[1] 47.68889
```

Specifying `mean <- mean(prestige)` causes our `mean` function to calculate the mean `prestige` and then stores the result in a variable called `mean`, which has the effect of destroying our `mean` function (and good riddance to it). The *variable* `mean` in the working data does not, however, shadow the *function* `mean` in the **base** package:

```
> mean(prestige, trim=0.1)

[1] 47.2973
```

Before proceeding, let us tidy up a bit:

```
> remove(mean)
> detach(Duncan)
```

2.2.2 AVOIDING `attach`

Here are some compelling reasons for *not* attaching data frames to the search path:

- Variables in attached data frames may mask other objects, and variables in attached data frames themselves may be masked by objects of the same name—for example, in the global environment.
- Attaching a data frame makes a *copy* of the data frame; the attached version is a snapshot of the data frame at the moment when it is attached. If changes are made to the data frame, these are *not* reflected in the attached version of the data. Consequently, after making such a change, it is necessary to detach and reattach the data frame. We find this procedure awkward, and inexperienced users of R may not remember to detach and reattach the data, leading to confusion about the current state of the attached data.
- We have observed that new users of R tend not to detach data frames after they are done with them. Often they will attach multiple versions of a data frame in the same session, which potentially results in confusion.

There are several strategies that we can use to avoid attaching a data frame:

- We can reference the variables in a data frame explicitly: for example,

  ```
  > mean(Duncan$prestige)

  [1] 47.68889
  ```

- Statistical-modeling functions in R usually include a `data` argument, which can be set to a data frame, conveniently specifying where the data for the model are to be found: for example,

  ```
  > (lm(prestige ~ income + education, data=Duncan))

  Call:
  lm(formula = prestige ~ income + education, data = Duncan)

  Coefficients:
  (Intercept)       income    education
      -6.0647       0.5987       0.5458
  ```

- The `with` command can be used to evaluate an R expression in the environment of a data frame: for example,

```
> with(Duncan, mean(prestige))
[1] 47.68889
> with(Duncan, lm(prestige ~ income + education))

Call:
lm(formula = prestige ~ income + education)

Coefficients:
(Intercept)       income    education
    -6.0647       0.5987       0.5458
```

We will use `with` frequently in the rest of this *Companion*.

2.2.3 MISSING DATA

Missing data are a regrettably common feature of real data sets. Two kinds of issues arise in handling missing data:

- There are relatively profound statistical issues concerning how best to use available information when missing data are encountered (see, e.g., Little and Rubin, 2002; and Schafer, 1997). We will ignore these issues here, except to remark that R is well designed to make use of sophisticated approaches to missing data.[7]
- There are intellectually trivial but often practically vexing mechanical issues concerning computing with missing data in R. These issues, which are the subject of the present section, arise partly because of the diverse data structures and kinds of functions available simultaneously to the R user. Similar issues arise in *all* statistical software, however, although they may sometimes be disguised.

As we have explained, on data input, missing values are typically encoded by the characters `NA`. The same characters are used to print missing information. Many functions in R know how to handle missing data, although sometimes they have to be explicitly told what to do.

To illustrate, let us examine the data set `Freedman` in the **car** package:

```
> head(Freedman)  # first 6 rows

           population nonwhite density crime
Akron             675      7.3     746  2602
Albany            713      2.6     322  1388
Albuquerque        NA      3.3      NA  5018
Allentown         534      0.8     491  1182
Anaheim          1261      1.4    1612  3341
Atlanta          1330     22.8     770  2805

> dim(Freedman)   # number of rows and columns

[1] 110   4
```

[7]Notable packages for handling missing data include **amelia**, **mi**, **mice**, and **norm**, which perform various versions of multiple imputation of missing data.

These data, on 110 U.S. metropolitan areas, were originally from the *1970 Statistical Abstract of the United States* and were used by Freedman (1975) as part of a wide-ranging study of the social and psychological effects of crowding. Freedman argues, by the way, that high density tends to intensify social interaction, and thus the effects of crowding are not simply negative. The variables in the data set are as follows:

- `population`: Total 1968 population, in thousands.
- `nonwhite`: Percent nonwhite population in 1960.
- `density`: Population per square mile in 1968.
- `crime`: Number of serious crimes per 100,000 residents in 1969.

Some of Freedman's data are missing—for example, the `population` and `density` for Albuquerque. Here are the first few values for `density`:

```
> head(Freedman$density, 20)  # first 20 values

 [1]  746  322   NA  491 1612  770   41  877  240  147  272 1831
[13] 1252  832  630   NA   NA  328  308 1832
```

Suppose, now, that we try to calculate the median `density`; as we will see shortly, the `density` values are highly positively skewed, so using the *mean* as a measure of the center of the distribution would be a bad idea:

```
> median(Freedman$density)

[1] NA
```

R tells us that the median `density` is missing. This is the pedantically correct answer: Several of the `density` values are missing, and consequently we cannot in the absence of those values know the median, but this is probably not what we had in mind when we asked for the median `density`. By setting the `na.rm` (NA-remove) argument of `median` to `TRUE`, we instruct R to calculate the median of the remaining, nonmissing values:

```
> median(Freedman$density, na.rm=TRUE)

[1] 412
```

Several other R functions that calculate statistical summaries, such as `mean`, `var` (variance), `sd` (standard deviation), and `quantile` (quantiles), also have `na.rm` arguments, but not all R functions handle missing data in this manner.

Most plotting functions simply ignore missing data. For example, to construct a scatterplot of `crime` against `density`, including only the observations with valid data for both variables, we enter

```
> with(Freedman, {
+     plot(density, crime)
+     identify(density, crime, row.names(Freedman))
+ })
```

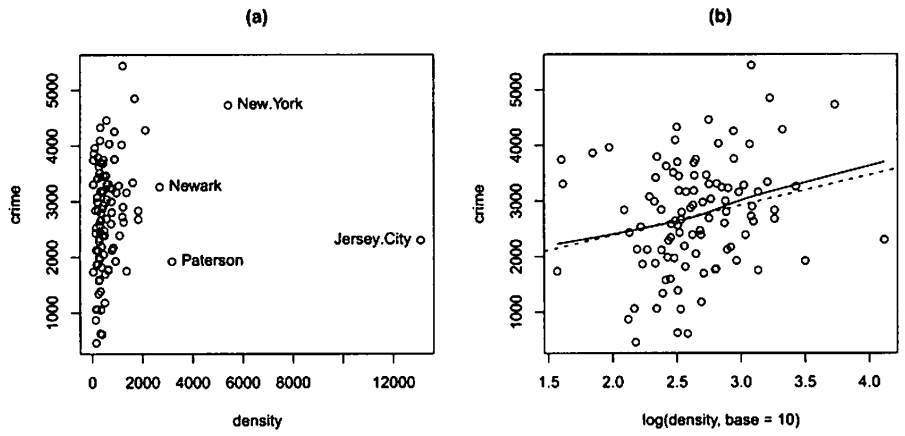

Figure 2.1 Scatterplot of `crime` by population `density` for Freedman's data. (a) Original density scale, with a few high-density cities identified interactively with the mouse, and (b) log-density scale, showing linear least-squares (broken) and lowess nonparametric-regression (solid) lines. Cases with one or both values missing are silently omitted from both graphs.

The resulting graph, including several observations identified with the mouse, appears in Figure 2.1a. Recall that we identify observations by pointing at them with the mouse and clicking the left mouse button; exit from `identify` by pressing the esc key in Mac OS X or, in Windows, by clicking the right mouse button and selecting *Stop*. It is apparent that `density` is highly positively skewed, making the plot very difficult to read. We would like to try plotting `crime` against the log of `density` but wonder whether the missing data will spoil the computation.[8] The `log` function in R behaves sensibly, however: The result has a missing entry wherever—and only where—there was a missing entry in the argument:

```
> log(c(1, 10, NA, 100), base=10)

[1]  0  1 NA  2
```

Other functions that compute on vectors in an element-wise fashion—such as the arithmetic operators—behave similarly.

We, therefore, may proceed as follows, producing the graph in Figure 2.1b:[9]

```
> with(Freedman, plot(log(density, base=10), crime))
```

This graph is much easier to read, and it now appears that there is a weak, positive relationship between `crime` and `density`. We will address momentarily how to produce the lines in the plot.

Statistical-modeling functions in R have a special argument, `na.action`, which specifies how missing data are to be handled; `na.action` is set to

[8] Transformations, including the log transformation, are the subject of Section 3.4.

[9] An alternative would have been to plot `crime` against `density` using a log axis for `density`: `plot(density, crime, log="x")`. See Chapters 3 and 7 for general discussions of plotting data in R.

a function that takes a data frame as an argument and returns a similar data frame composed entirely of valid data (see Section 4.8.5). The default `na.action` is `na.omit`, which removes all observations with missing data on *any* variable in the computation. All the examples in this *Companion* use `na.omit`. An alternative, for example, would be to supply an `na.action` that imputes the missing values.

The prototypical statistical-modeling function in R is `lm`, which is described extensively in Chapter 4. For example, to fit a linear regression of `crime` on the log of `density`, removing observations with missing data on either `crime` or `density`, enter the command

```
> lm(crime ~ log(density, base=10), data=Freedman)

Call:
lm(formula = crime ~ log(density, base = 10), data = Freedman)

Coefficients:
            (Intercept)  log(density, base = 10)
                 1297.3                   542.6
```

The `lm` function returns a linear-model object; because the returned object was not saved in a variable, the interpreter simply printed a brief report of the regression. To plot the least-squares line on the scatterplot in Figure 2.1:

```
> abline(lm(crime ~ log(density, base=10), data=Freedman),
  lty="dashed")
```

The linear-model object returned by `lm` is passed to `abline`, which draws the regression line; specifying the line type `lty="dashed"` produces a broken line.

Some functions in R, especially the older ones, make no provision for missing data and simply fail if an argument has a missing entry. In these cases, we need somewhat tediously to handle the missing data ourselves. A relatively straightforward way to do so is to use the `complete.cases` function to test for missing data, and then to exclude the missing data from the calculation.

For example, to locate all observations with valid data for both `crime` and `density`, we enter:

```
> good <- with(Freedman, complete.cases(crime, density))
> head(good, 20)  # first 20 values

 [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[11]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
```

We then use `good` to select the valid observations by indexing (a topic described in Section 2.3.4). For example, it is convenient to use the `lowess` function to add a nonparametric-regression smooth to our scatterplot (Figure 2.1b), but `lowess` makes no provision for missing data:[10]

[10]The `lowess` function is described in Section 3.2.1.

```
> with(Freedman,
+   lines(lowess(log(density[good], base=10), crime[good], f=1.0)))
```

By indexing the predictor `density` and response `crime` with the logical vector `good`, we extract only the observations that have valid data for *both* variables. The argument `f` to the `lowess` function specifies the *span* of the lowess smoother—that is, the fraction of the data included in each local-regression fit; large spans (such as the value `1.0` employed here) produce smooth regression curves.

Suppose, as is frequently the case, that we analyze a data set with a complex pattern of missing data, fitting several statistical models to the data. If the models do not all use exactly the same variables, then it is likely that they will be fit to different subsets of nonmissing observations. Then if we compare the models with a likelihood ratio test, for example, the comparison will be invalid.[11]

To avoid this problem, we can first use `na.omit` to filter the data frame for missing data, including all the variables that we intend to use in our data analysis. For example, for Freedman's data, we may proceed as follows, assuming that we want subsequently to use all four variables in the data frame:

```
> Freedman.good <- na.omit(Freedman)
> head(Freedman.good)  # first 6 rows
```

```
           population nonwhite density crime
Akron              675      7.3     746  2602
Albany             713      2.6     322  1388
Allentown          534      0.8     491  1182
Anaheim           1261      1.4    1612  3341
Atlanta           1330     22.8     770  2805
Bakersfield        331      7.0      41  3306
```

```
> dim(Freedman.good)   # number of rows and columns
```

```
[1] 100   4
```

A note of caution: Filtering for missing data on variables that we *do not* intend to use can result in discarding data unnecessarily. We have seen cases where students and researchers inadvertently and needlessly threw away most of their data by filtering an entire data set for missing values, even when they intended to use only a few variables in the data set.

Finally, a few words about testing for missing data in R: A common error is to assume that one can check for missing data using the == (equals) operator, as in

```
> NA == c(1, 2, NA, 4)
```

```
[1] NA NA NA NA
```

Testing equality of `NA` against *any* value in R returns `NA` as a result. After all, if the value is missing, how can we know whether it's equal to something else? The proper way to test for missing data is with the function `is.na`:

[11] How statistical-modeling functions in R handle missing data is described in Section 4.8.5.

```
> is.na(c(1, 2, NA, 4))

[1] FALSE FALSE  TRUE FALSE
```

For example, to count the number of missing values in the `Freedman` data frame:

```
> sum(is.na(Freedman))

[1] 20
```

This command relies on the automatic *coercion* of the logical values `TRUE` and `FALSE` to one and zero, respectively.

2.2.4 NUMERIC VARIABLES AND FACTORS

If we construct R data frames by reading data from text files using `read.table` or from numeric and character vectors using `data.frame`, then our data frames will consist of two kinds of data: numeric variables and factors. Both `read.table` and `data.frame` by default translate character data and logical data into factors.

Before proceeding, let us clean up a bit:

```
> objects()

[1] "condition"      "cooperation"    "Freedman.good"
[4] "good"           "Guyer"          "sex"

> remove(good, Freedman.good)
```

Near the beginning of this chapter, we entered data from Fox and Guyer's (1978) experiment on anonymity and cooperation into the global variables `cooperation`, `condition`, and `sex`.[12] The latter two variables are character vectors, as we verify for `condition`:

```
> condition

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

We can confirm that this is a vector of character values using the *predicate function* `is.character`, which tests whether its argument is of mode `"character"`:

```
> is.character(condition)

[1] TRUE
```

[12] Variables created by assignment at the command prompt are global variables defined in the working data.

After entering the data, we defined the data frame Guyer, which also contains variables named cooperation, condition, and sex. We will remove the global variables and will work instead with the data frame:

```
> remove(cooperation, condition, sex)
```

Look at the variable condition in the Guyer data frame:

```
> Guyer$condition

 [1] public    public    public    public    public    public
 [7] public    public    public    public    anonymous anonymous
[13] anonymous anonymous anonymous anonymous anonymous anonymous
[19] anonymous anonymous
Levels: anonymous public

> is.character(Guyer$condition)

[1] FALSE

> is.factor(Guyer$condition)

[1] TRUE
```

As we explained, condition in the data frame is a *factor* rather than a character vector. A factor is a representation of a categorical variable; factors are stored more economically than character vectors, and the manner in which they are stored saves information about the *levels* (category set) of a factor. When a factor is printed, its values are not quoted, as are the values of a character vector, and the levels of the factor are listed.

Many functions in R, including statistical-modeling functions such as lm, know how to deal with factors. For example, when the generic summary function is called with a data frame as its argument, it prints various statistics for a numeric variable but simply counts the number of observations in each level of a factor:

```
> summary(Guyer)

  cooperation        condition      sex
 Min.   :27.00   anonymous:10   female:10
 1st Qu.:38.50   public   :10   male  :10
 Median :46.50
 Mean   :48.30
 3rd Qu.:58.75
 Max.   :79.00
```

Factors have unordered levels. An extension, called *ordered factors*, is discussed, along with factors, in the context of linear models in Section 4.6.

2.2.5 MODIFYING AND TRANSFORMING DATA

Data modification in R usually occurs naturally and unremarkably. When we wanted to plot crime against the log of density in Freedman's data, for

example, we simply specified `log(density, base=10)`.[13] Similarly, in regressing `crime` on the log of `density`, we just used `log(density, base=10)` on the right-hand side of the linear-model formula.

Creating new variables that are functions of other variables is straightforward. For example, the variable `cooperation` in the `Guyer` data set counts the number of cooperative choices out of a total of 120 choices. To create a new variable with the percentage of cooperative choices in each group:

```
> perc.coop <- 100*Guyer$cooperation/120
```

The new variable `perc.coop` resides in the working data, not in the `Guyer` data frame. It is generally advantageous to add new variables such as this to the data frame from which they originate: Keeping related variables together in a data frame avoids confusion, for example.

```
> Guyer$perc.coop <- 100*Guyer$cooperation/120
> head(Guyer)  # first 6 rows

  cooperation condition    sex perc.coop
1          49    public   male  40.83333
2          64    public   male  53.33333
3          37    public   male  30.83333
4          52    public   male  43.33333
5          68    public   male  56.66667
6          54    public female  45.00000
```

A similar procedure may be used to *modify* an existing variable in a data frame. The following command, for example, replaces the original `cooperation` variable in `Guyer` with the logit (log-odds) of cooperation:

```
> Guyer$cooperation <- with(Guyer, log(perc.coop/(100 - perc.coop)))
> head(Guyer)

  cooperation condition    sex perc.coop
1  -0.3708596    public   male  40.83333
2   0.1335314    public   male  53.33333
3  -0.8079227    public   male  30.83333
4  -0.2682640    public   male  43.33333
5   0.2682640    public   male  56.66667
6  -0.2006707    public female  45.00000
```

The `transform` function can be used to create and modify several variables in a data frame at once. For example, if we have a data frame called `Data` with variables named `a`, `b`, and `c`, then the command

```
> Data <- transform(Data, c=-c, asq=a^2, a.over.b=a/b)
```

replaces `Data` by a new data frame in which the variables `a` and `b` are included unchanged, `c` is replaced by `-c`, and two new variables are added—`asq`, with the squares of `a`, and `a.over.b`, with the ratios of `a` to `b`.

[13] We did not have to create a new variable, say `log.density <- log(density, 10)`, as one may be required to do in a typical statistical package such as SAS or SPSS.

Transforming numeric data is usually a straightforward operation—simply using mathematical operators and functions. Categorizing numeric data and recoding categorical variables are often more complicated matters. Several functions in R are employed to create factors from numeric data and to manipulate categorical data, but we will limit our discussion to three that we find particularly useful: (1) the standard R function `cut`, (2) the `recode` function in the **car** package, and (3) the standard `ifelse` function.

The `cut` function dissects the range of a numeric variable into class intervals, or *bins*. The first argument to the function is the variable to be binned; the second argument gives either the number of equal-width bins or a vector of cut points at which the division is to take place. For example, to divide the range of `perc.coop` into four equal-width bins, we specify

```
> Guyer$coop.4 <- cut(Guyer$perc.coop, 4)
> summary(Guyer$coop.4)

(22.5,33.3] (33.3,44.2]   (44.2,55]   (55,65.9]
          6           7           5           2
```

R responds by creating a factor, the levels of which are named for the end points of the bins. In the example above, the first level includes all values with `Guyer$perc.coop` greater than 22.5 (which is slightly smaller than the minimum value of `Guyer$perc.coop`) and less than or equal to 33.3, the cut point between the first two levels. Because `perc.coop` is not uniformly distributed across its range, the several levels of `coop.4` contain different numbers of observations. The output from the `summary` function applied to a factor gives a one-dimensional table of the number of observations in each level of the factor.

Suppose, alternatively, that we want to bin `perc.coop` into three levels containing roughly equal numbers of observations[14] and to name these levels `"low"`, `"med"`, and `"high"`; we may proceed as follows:

```
> Guyer$coop.groups <- with(Guyer, cut(perc.coop,
+     quantile(perc.coop, c(0, 1/3, 2/3, 1)),
+     include.lowest=TRUE,
+     labels=c("low", "med", "high")))
> summary(Guyer$coop.groups)

 low  med high
   7    6    7
```

The `quantile` function is used to locate the cut points. Had we wished to divide `perc.coop` into four groups, for example, we would simply have specified different quantiles, `c(0, .25, .5, .75, 1)`, and of course supplied four values for the `labels` argument.

[14] *Roughly* equal numbers of observations in the three bins are the best we can do because $n = 20$ is not evenly divisible by 3.

The `recode` function in the **car** package, which is more flexible than `cut`, can also be used to dissect a quantitative variable into class intervals: for example,

```
> (Guyer$coop.2 <- recode(Guyer$perc.coop, "lo:50=1; 50:hi=2"))

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

The `recode` function works as follows:

- The first argument is the variable to be recoded, here `perc.coop`.
- The second argument is a character string, enclosed in single or double quotes, containing the recode specifications.
- Recode specifications are of the form *`old.values=new.value`*. There may be several recode specifications, separated by semicolons.
- The *`old.values`* may be a single value, including `NA`; a range, of the form *`minimum:maximum`*, as in the example, where the special values `lo` and `hi` have been used to stand in for the smallest and largest values of the variable; a vector of values, typically specified with the `c` (combine) function; or the special symbol `else`, which, if present, should appear last.
- An observation that fits into more than one recode specification is assigned the value of the first one encountered. For example, a group with `perc.coop` exactly equal to `50` would get the new value `1`.
- Character data may appear both as *`old.values`* and as *`new.value`*. You must be careful with quotation marks, however: If single quotes are employed to enclose the recode specifications, then double quotes must be used for the values (and vice versa).
- When a factor is recoded, the *`old.values`* should be specified as character strings; the result is a factor, even if the *`new.values`* are numbers, unless the argument `as.factor.result` is set to `FALSE`.
- Character data may be recoded to numeric, and vice versa. To recode a character or numeric variable to a factor, set `as.factor.result =TRUE`.
- If an observation does not satisfy any of the recode specifications, then the *`old.value`* for that observation is carried over into the result.

To provide a richer context for additional illustrations of the use of `recode`, we turn our attention to the `Womenlf` data frame from the **car** package:

```
> set.seed(12345)  # for reproducibility
> (sample.20 <- sort(sample(nrow(Womenlf), 20)))  # 20 random obs.

 [1]   1   9  39  43  44  84  96  98 100 115 119 131 185 186 190
[16] 199 230 231 233 252

> Womenlf[sample.20, ]  # 20 randomly selected rows
```

```
      partic hincome children   region
1   not.work      15  present  Ontario
9   not.work      15  present  Ontario
39  not.work       9  present Atlantic
43  parttime      28   absent  Ontario
44  not.work      23  present  Ontario
84  fulltime      17  present  Ontario
96  not.work      17  present  Ontario
98  fulltime      15   absent  Ontario
100 not.work      15  present  Ontario
115 parttime      13  present  Prairie
119 fulltime      15   absent       BC
131 parttime      19  present  Ontario
185 not.work      13   absent  Ontario
186 parttime      15  present       BC
190 not.work      23  present       BC
199 fulltime      10   absent   Quebec
230 parttime      23  present   Quebec
231 not.work       7  present   Quebec
233 fulltime      15   absent   Quebec
252 not.work      23   absent   Quebec
```

The `sample` function is used to pick a random sample of 20 rows in the data frame, selecting 20 random numbers without replacement from one to the number of rows in `Womenlf`; the numbers are placed in ascending order by the `sort` function.[15]

We use the `set.seed` function to specify the seed for R's pseudo-random number generator, ensuring that if we repeat the `sample` command, we will obtain the same sequence of pseudo-random numbers. Otherwise, the seed of the random-number generator will be selected unpredictably based on the system clock when the first random number is generated in an R session. Setting the random seed to a known value before a random simulation makes the result of the simulation reproducible. In serious work, we generally prefer to start with a known but randomly selected seed, as follows:

```
> (seed <- sample(2^31 - 1, 1))

[1] 974373618

> set.seed(seed)
```

The number $2^{31} - 1$ is the largest integer representable as a 32-bit binary number on most of the computer systems on which R runs (see Section 2.6.2).

The data in `Womenlf` originate from a social survey of the Canadian population conducted in 1977 and pertain to married women between the ages of 21 and 30, with the variables defined as follows:

- `partic`: Labor-force participation, `parttime`, `fulltime`, or `not.work` (not working outside the home).

[15] If the objective here were simply to sample 20 rows from `Womenlf`, then we could more simply use the `some` function in the **car** package, `some(Womenlf, 20)`, but we will reuse this sample to check on the results of our recodes.

- `hincome`: Husband's income, in $1,000s (actually, family income minus wife's income).
- `children`: Presence of children in the household: `present` or `absent`.
- `region`: `Atlantic`, `Quebec`, `Ontario`, `Prairie` (the Prairie provinces), or `BC` (British Columbia).

Now consider the following recodes:

```
> # recode in two ways:
> Womenlf$working <- recode(Womenlf$partic,
+     ' c("parttime", "fulltime")="yes"; "not.work"="no" ')
> Womenlf$working.alt <- recode(Womenlf$partic,
+     ' c("parttime", "fulltime")="yes"; else="no" ')
> Womenlf$working[sample.20]  # 20 sampled observations

 [1] no  no  no  yes no  yes no  yes no  yes yes yes no  yes no
[16] yes yes no  yes no
Levels: no yes

> with(Womenlf, all(working == working.alt))  # check

[1] TRUE

> Womenlf$fulltime <- recode(Womenlf$partic,
+     ' "fulltime"="yes"; "parttime"="no"; "not.work"=NA ')
> Womenlf$fulltime[sample.20]  # 20 sampled observations

 [1] <NA> <NA> <NA> no   <NA> yes  <NA> yes  <NA> no   yes  no
[13] <NA> no   <NA> yes  no   <NA> yes  <NA>
Levels: no yes

> Womenlf$region.4 <- recode(Womenlf$region,
+     ' c("Prairie", "BC")="West" ')
> Womenlf$region.4[sample.20]  # 20 sampled observations

 [1] Ontario  Ontario  Atlantic Ontario  Ontario  Ontario
 [7] Ontario  Ontario  Ontario  West     West     Ontario
[13] Ontario  West     West     Quebec   Quebec   Quebec
[19] Quebec   Quebec
Levels: Atlantic Ontario Quebec West
```

In all these examples, factors (either `partic` or `region`) are recoded, and consequently, `recode` returns factors as results:

- The first two examples yield identical results, with the second example illustrating the use of `else`. To verify that all the values in `working` and `working.alt` are the same, we use the `all` function along with the element-wise comparison operator == (equals).
- In the third example, the factor `fulltime` is created, indicating whether a woman who works outside the home works full-time or part-time; `fulltime` is NA (missing) for women who do not work outside the home.
- The fourth and final example illustrates how values that are *not* recoded (here `Atlantic`, `Quebec`, and `Ontario` in the factor `region`) are simply carried over to the result.

The standard R `ifelse` command (discussed further in Section 8.3.1) can also be used to recode data. For example,