4 Basic data management

This chapter covers

- Manipulating dates and missing values
- Understanding data type conversions
- Creating and recoding variables
- Sorting, merging, and subsetting datasets
- Selecting and dropping variables

In chapter 2, we covered a variety of methods for importing data into R. Unfortunately, getting our data in the rectangular arrangement of a matrix or data frame is the first step in preparing it for analysis. To paraphrase Captain Kirk in the Star Trek episode "A Taste of Armageddon" (and proving my geekiness once and for all): "Data is a messy business—a very, very messy business." In my own work, as much as 60 percent of the time I spend on data analysis is focused on preparing the data for analysis. I'll go out a limb and say that the same is probably true in one form or another for most real-world data analysts. Let's take a look at an example.

4.1 A working example

One of the topics that I study in my current job is how men and women differ in the ways they lead their organizations. Typical questions might be

- Do men and women in management positions differ in the degree to which they defer to superiors?

- Does this vary from country to country, or are these gender differences universal?

One way to address these questions is to have bosses in multiple countries rate their managers on deferential behavior, using questions like the following:

This manager asks my opinion before making personnel decisions.

1	**2**	**3**	**4**	**5**
strongly disagree	disagree	neither agree nor disagree	agree	strongly agree

The resulting data might resemble those in table 4.1. Each row represents the ratings given to a manager by his or her boss.

Table 4.1 Gender differences in leadership behavior

Manager	Date	Country	Gender	Age	q1	q2	q3	q4	q5
1	10/24/08	US	M	32	5	4	5	5	5
2	10/28/08	US	F	45	3	5	2	5	5
3	10/01/08	UK	F	25	3	5	5	5	2
4	10/12/08	UK	M	39	3	3	4		
5	05/01/09	UK	F	99	2	2	1	2	1

Here, each manager is rated by their boss on five statements (q1 to q5) related to deference to authority. For example, manager 1 is a 32-year-old male working in the US and is rated deferential by his boss, while manager 5 is a female of unknown age (99 probably indicates missing) working in the UK and is rated low on deferential behavior. The date column captures when the ratings were made.

Although a dataset might have dozens of variables and thousands of observations, we've only included 10 columns and 5 rows to simplify the examples. Additionally, we've limited the number of items pertaining to the managers' deferential behavior to 5. In a real-world study, you'd probably use 10–20 such items to improve the reliability and validity of the results. You can create a data frame containing the data in table 4.1 using the following code.

Listing 4.1 Creating the leadership data frame

```
manager <- c(1, 2, 3, 4, 5)
date <- c("10/24/08", "10/28/08", "10/1/08", "10/12/08", "5/1/09")
country <- c("US", "US", "UK", "UK", "UK")
gender <- c("M", "F", "F", "M", "F")
age <- c(32, 45, 25, 39, 99)
q1 <- c(5, 3, 3, 3, 2)
q2 <- c(4, 5, 5, 3, 2)
q3 <- c(5, 2, 5, 4, 1)
q4 <- c(5, 5, 5, NA, 2)
q5 <- c(5, 5, 2, NA, 1)
leadership <- data.frame(manager, date, country, gender, age,
                         q1, q2, q3, q4, q5, stringsAsFactors=FALSE)
```

In order to address the questions of interest, we must first address several data management issues. Here's a partial list:

- The five ratings (q1 to q5) will need to be combined, yielding a single mean deferential score from each manager.
- In surveys, respondents often skip questions. For example, the boss rating manager 4 skipped questions 4 and 5. We'll need a method of handling incomplete data. We'll also need to recode values like 99 for age to missing.
- There may be hundreds of variables in a dataset, but we may only be interested in a few. To simplify matters, we'll want to create a new dataset with only the variables of interest.
- Past research suggests that leadership behavior may change as a function of the manager's age. To examine this, we may want to recode the current values of age into a new categorical age grouping (for example, young, middle-aged, elder).
- Leadership behavior may change over time. We might want to focus on deferential behavior during the recent global financial crisis. To do so, we may want to limit the study to data gathered during a specific period of time (say, January 1, 2009 to December 31, 2009).

We'll work through each of these issues in the current chapter, as well as other basic data management tasks such as combining and sorting datasets. Then in chapter 5 we'll look at some advanced topics.

4.2 Creating new variables

In a typical research project, you'll need to create new variables and transform existing ones. This is accomplished with statements of the form

```
variable <- expression
```

A wide array of operators and functions can be included in the *expression* portion of the statement. Table 4.2 lists R's arithmetic operators. Arithmetic operators are used when developing formulas.

Table 4.2 Arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^ or **	Exponentiation
x%%y	Modulus (x mod y) 5%%2 is 1
x%/%y	Integer division 5%/%2 is 2

Let's say that you have a data frame named `mydata`, with variables `x1` and `x2`, and you want to create a new variable `sumx` that adds these two variables and a new variable called `meanx` that averages the two variables. If you use the code

```
sumx  <-  x1 + x2
meanx <- (x1 + x2)/2
```

you'll get an error, because R doesn't know that `x1` and `x2` are from data frame `mydata`. If you use this code instead

```
sumx  <-  mydata$x1 + mydata$x2
meanx <- (mydata$x1 + mydata$x2)/2
```

the statements will succeed but you'll end up with a data frame (`mydata`) and two separate vectors (`sumx` and `meanx`). This is probably not what you want. Ultimately, you want to incorporate new variables into the original data frame. The following listing provides three separate ways to accomplish this goal. The one you choose is up to you; the results will be the same.

Listing 4.2 Creating new variables

```
mydata<-data.frame(x1 = c(2, 2, 6, 4),
                   x2 = c(3, 4, 2, 8))

mydata$sumx  <-  mydata$x1 + mydata$x2
mydata$meanx <- (mydata$x1 + mydata$x2)/2


attach(mydata)
mydata$sumx  <-  x1 + x2
mydata$meanx <- (x1 + x2)/2
detach(mydata)

mydata <- transform(mydata,
                    sumx  =  x1 + x2,
                    meanx = (x1 + x2)/2)
```

Personally, I prefer the third method, exemplified by the use of the `transform()` function. It simplifies inclusion of as many new variables as desired and saves the results to the data frame.

4.3 Recoding variables

Recoding involves creating new values of a variable conditional on the existing values of the same and/or other variables. For example, you may want to

- Change a continuous variable into a set of categories
- Replace miscoded values with correct values
- Create a pass/fail variable based on a set of cutoff scores

To recode data, you can use one or more of R's logical operators (see table 4.3). Logical operators are expressions that return `TRUE` or `FALSE`.

Table 4.3 Logical operators

Operator	Description
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Exactly equal to
`!=`	Not equal to
`!x`	Not *x*
`x | y`	*x* or *y*
`x & y`	*x* and *y*
`isTRUE(x)`	Test if *x* is `TRUE`

Let's say that you want to recode the ages of the managers in our leadership dataset from the continuous variable `age` to the categorical variable `agecat (Young, Middle Aged, Elder)`. First, you must recode the value 99 for age to missing with code such as

```
leadership$age[leadership$age  == 99]     <- NA
```

The statement `variable[condition] <- expression` will only make the assignment when `condition` is `TRUE`.

Once missing values for age have been specified, you can then use the following code to create the `agecat` variable:

```
leadership$agecat[leadership$age  > 75]   <- "Elder"
leadership$agecat[leadership$age >= 55 &
                  leadership$age <= 75]   <- "Middle Aged"
leadership$agecat[leadership$age  < 55]   <- "Young"
```

You include the data frame names in `leadership$agecat` to ensure that the new variable is saved back to the data frame. You define middle aged as 55 to 75 so that I won't feel so old. Note that if you hadn't recoded 99 as missing for `age` first, manager 5 would've erroneously been given the value "Elder" for `agecat`.

This code can be written more compactly as

```
leadership <- within(leadership,{
                     agecat <- NA
                     agecat[age > 75]              <- "Elder"
                     agecat[age >= 55 & age <= 75] <- "Middle Aged"
                     agecat[age < 55]              <- "Young" })
```

The `within()` function is similar to the `with()` function (section 2.2.4), but allows you to modify the data frame. First, the variable `agecat` variable is created and set to missing for each row of the data frame. Then the remaining statements within the

braces are executed in order. Remember that `agecat` is a character variable; you're likely to want to turn it into an ordered factor, as explained in section 2.2.5.

Several packages offer useful recoding functions; in particular, the `car` package's `recode()` function recodes numeric and character vectors and factors very simply. The package `doBy` offers `recodevar()`, another popular function. Finally, R ships with `cut()`, which allows you to divide the range of a numeric variable into intervals, returning a factor.

4.4 Renaming variables

If you're not happy with your variable names, you can change them interactively or programmatically. Let's say that you want to change the variables `manager` to `managerID` and `date` to `testDate`. You can use the statement

```
fix(leadership)
```

to invoke an interactive editor, click on the variable names, and rename them in the dialogs that are presented (see figure 4.1).

Programmatically, the `reshape` package has a `rename()` function that's useful for altering the names of variables. The format of the `rename()` function is

```
rename(dataframe, c(oldname="newname", oldname="newname",…))
```

Here's an example:

```
library(reshape)
leadership <- rename(leadership,
                     c(manager="managerID", date="testDate")
)
```

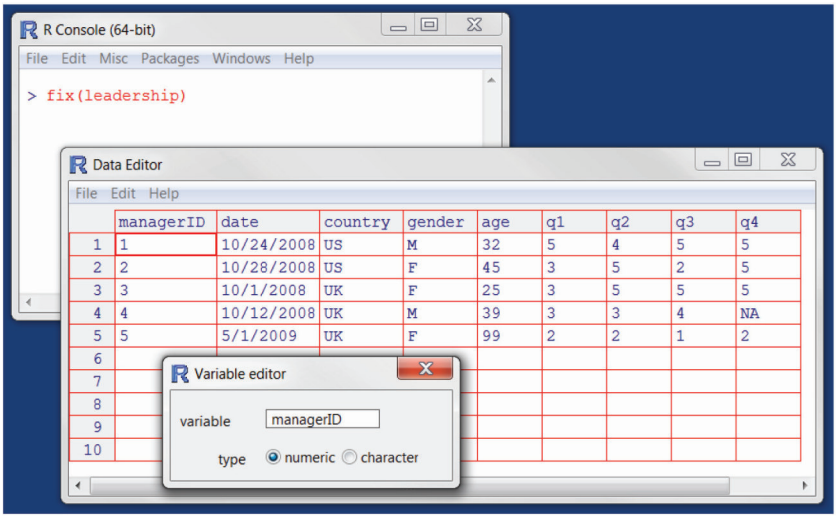

Figure 4.1 Renaming variables interactively using the `fix()` function

The `reshape` package isn't installed by default, so you'll need to install it on first use using the `install.packages("reshape")` command. The `reshape` package has a powerful set of functions for altering the structure of a dataset. We'll explore several in chapter 5.

Finally, you can rename variables via the `names()` function. For example:

```
names(leadership)[2] <- "testDate"
```

would rename `date` to `testDate` as demonstrated in the following code:

```
> names(leadership)
 [1] "manager" "date"     "country" "gender"   "age"      "q1"       "q2"
 [8] "q3"      "q4"       "q5"
> names(leadership)[2] <- "testDate"
> leadership
  manager testDate country gender age q1 q2 q3 q4 q5
1       1 10/24/08      US      M  32  5  4  5  5  5
2       2 10/28/08      US      F  45  3  5  2  5  5
3       3  10/1/08      UK      F  25  3  5  5  5  2
4       4 10/12/08      UK      M  39  3  3  4 NA NA
5       5   5/1/09      UK      F  99  2  2  1  2  1
```

In a similar fashion,

```
names(leadership)[6:10] <- c("item1", "item2", "item3", "item4", "item5")
```

would rename `q1` through `q5` to `item1` through `item5`.

4.5 Missing values

In a project of any size, data is likely to be incomplete because of missed questions, faulty equipment, or improperly coded data. In R, missing values are represented by the symbol `NA` (not available). Impossible values (for example, dividing by 0) are represented by the symbol `NaN` (not a number). Unlike programs such as SAS, R uses the same missing values symbol for character and numeric data.

R provides a number of functions for identifying observations that contain missing values. The function `is.na()` allows you to test for the presence of missing values. Assume that you have a vector:

```
y <- c(1, 2, 3, NA)
```

then the function

```
is.na(y)
```

returns `c(FALSE, FALSE, FALSE, TRUE)`.

Notice how the `is.na()` function works on an object. It returns an object of the same size, with the entries replaced by `TRUE` if the element is a missing value, and `FALSE` if the element is not a missing value. Listing 4.3 applies this to our leadership example.

Listing 4.3 Applying the `is.na()` function

```
> is.na(leadership[,6:10])
      q1    q2    q3    q4    q5
[1,] FALSE FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE FALSE
[4,] FALSE FALSE FALSE  TRUE  TRUE
[5,] FALSE FALSE FALSE FALSE FALSE
```

Here, `leadership[,6:10]` limited the data frame to columns 6 to 10, and `is.na()` identified which values are missing.

> **NOTE** Missing values are considered noncomparable, even to themselves. This means that you can't use comparison operators to test for the presence of missing values. For example, the logical test `myvar == NA` is never `TRUE`. Instead, you have to use missing values functions, like those in this section, to identify the missing values in R data objects.

4.5.1 Recoding values to missing

As demonstrated in section 4.3, you can use assignments to recode values to missing. In our leadership example, missing age values were coded as 99. Before analyzing this dataset, you must let R know that the value 99 means missing in this case (otherwise the mean age for this sample of bosses will be way off!). You can accomplish this by recoding the variable:

```
leadership$age[leadership$age == 99] <- NA
```

Any value of age that's equal to 99 is changed to NA. Be sure that any missing data is properly coded as missing before analyzing the data or the results will be meaningless.

4.5.2 Excluding missing values from analyses

Once you've identified the missing values, you need to eliminate them in some way before analyzing your data further. The reason is that arithmetic expressions and functions that contain missing values yield missing values. For example, consider the following code:

```
x <- c(1, 2, NA, 3)
y <- x[1] + x[2] + x[3] + x[4]
z <- sum(x)
```

Both `y` and `z` will be `NA` (missing) because the third element of `x` is missing.

Luckily, most numeric functions have a `na.rm=TRUE` option that removes missing values prior to calculations and applies the function to the remaining values:

```
x <- c(1, 2, NA, 3)
y <- sum(x, na.rm=TRUE)
```

Here, `y` is equal to 6.

When using functions with incomplete data, be sure to check how that function handles missing data by looking at its online help (for example, `help(sum)`). The

`sum()` function is only one of many functions we'll consider in chapter 5. Functions allow you to transform data with flexibility and ease.

You can remove *any* observation with missing data by using the `na.omit()` function. `na.omit()` deletes any rows with missing data. Let's apply this to our leadership dataset in the following listing.

Listing 4.4 Using `na.omit()` to delete incomplete observations

```
> leadership
  manager     date country gender age q1 q2 q3 q4 q5
1       1 10/24/08      US      M  32  5  4  5  5  5
2       2 10/28/08      US      F  40  3  5  2  5  5
3       3 10/01/08      UK      F  25  3  5  5  5  2
4       4 10/12/08      UK      M  39  3  3  4 NA NA
5       5 05/01/09      UK      F  99  2  2  1  2  1

> newdata <- na.omit(leadership)
> newdata
  manager     date country gender age q1 q2 q3 q4 q5
1       1 10/24/08      US      M  32  5  4  5  5  5
2       2 10/28/08      US      F  40  3  5  2  5  5
3       3 10/01/08      UK      F  25  3  5  5  5  2
5       5 05/01/09      UK      F  99  2  2  1  2  1
```

Data frame with missing data

Data frame with complete cases only

Any rows containing missing data are deleted from `leadership` before the results are saved to `newdata`.

Deleting all observations with missing data (called listwise deletion) is one of several methods of handling incomplete datasets. If there are only a few missing values or they're concentrated in a small number of observations, listwise deletion can provide a good solution to the missing values problem. But if missing values are spread throughout the data, or there's a great deal of missing data in a small number of variables, listwise deletion can exclude a substantial percentage of your data. We'll explore several more sophisticated methods of dealing with missing values in chapter 15. Next, let's take a look at dates.

4.6 Date values

Dates are typically entered into R as character strings and then translated into date variables that are stored numerically. The function `as.Date()` is used to make this translation. The syntax is `as.Date(x, "input_format")`, where `x` is the character data and `input_format` gives the appropriate format for reading the date (see table 4.4).

Table 4.4 Date formats

Symbol	Meaning	Example
%d	Day as a number (0–31)	01–31
%a %A	Abbreviated weekday Unabbreviated weekday	Mon Monday
%m	Month (00–12)	00–12

Table 4.4 Date formats *(continued)*

Symbol	Meaning	Example
%b	Abbreviated month	Jan
%B	Unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007

The default format for inputting dates is yyyy-mm-dd. The statement

```
mydates <- as.Date(c("2007-06-22", "2004-02-13"))
```

converts the character data to dates using this default format. In contrast,

```
strDates <- c("01/05/1965", "08/16/1975")
dates <- as.Date(strDates, "%m/%d/%Y")
```

reads the data using a mm/dd/yyyy format.

In our leadership dataset, date is coded as a character variable in mm/dd/yy format. Therefore:

```
myformat <- "%m/%d/%y"
leadership$date <- as.Date(leadership$date, myformat)
```

uses the specified format to read the character variable and replace it in the data frame as a date variable. Once the variable is in date format, you can analyze and plot the dates using the wide range of analytic techniques covered in later chapters.

Two functions are especially useful for time-stamping data. `Sys.Date()` returns today's date and `date()` returns the current date and time. As I write this, it's December 12, 2010 at 4:28pm. So executing those functions produces

```
> Sys.Date()
[1] "2010-12-01"
> date()
[1] "Wed Dec 01 16:28:21 2010"
```

You can use the `format(x, format="output_format")` function to output dates in a specified format, and to extract portions of dates:

```
> today <- Sys.Date()
> format(today, format="%B %d %Y")
[1] "December 01 2010"
> format(today, format="%A")
[1] "Wednesday"
```

The `format()` function takes an argument (a date in this case) and applies an output format (in this case, assembled from the symbols in table 4.4). The important result here is that there are only two more days until the weekend!

When R stores dates internally, they're represented as the number of days since January 1, 1970, with negative values for earlier dates. That means you can perform arithmetic operations on them. For example:

```
> startdate <- as.Date("2004-02-13")
> enddate   <- as.Date("2011-01-22")
```

```
> days      <- enddate - startdate
> days
Time difference of 2535 days
```

displays the number of days between February 13, 2004 and January 22, 2011.

Finally, you can also use the function `difftime()` to calculate a time interval and express it as seconds, minutes, hours, days, or weeks. Let's assume that I was born on October 12, 1956. How old am I?

```
> today <- Sys.Date()
> dob   <- as.Date("1956-10-12")
> difftime(today, dob, units="weeks")
Time difference of 2825 weeks
```

Apparently, I am 2825 weeks old. Who knew? Final test: On which day of the week was I born?

4.6.1 Converting dates to character variables

Although less commonly used, you can also convert date variables to character variables. Date values can be converted to character values using the `as.character()` function:

```
strDates <- as.character(dates)
```

The conversion allows you to apply a range of character functions to the data values (subsetting, replacement, concatenation, etc.). We'll cover character functions in detail in chapter 5.

4.6.2 Going further

To learn more about converting character data to dates, take a look at `help(as.Date)` and `help(strftime)`. To learn more about formatting dates and times, see `help(ISOdatetime)`. The `lubridate` package contains a number of functions that simplify working with dates, including functions to identify and parse date-time data, extract date-time components (for example, years, months, days, etc.), and perform arithmetic calculations on date-times. If you need to do complex calculations with dates, the `fCalendar` package can also help. It provides a myriad of functions for dealing with dates, can handle multiple time zones at once, and provides sophisticated calendar manipulations that recognize business days, weekends, and holidays.

4.7 Type conversions

In the previous section, we discussed how to convert character data to date values, and vice versa. R provides a set of functions to identify an object's data type and convert it to a different data type.

Type conversions in R work in a similar fashion to those in other statistical programming languages. For example, adding a character string to a numeric vector converts all the elements in the vector to character values. You can use the functions listed in table 4.5 to test for a data type and to convert it to a given type.

Table 4.5 Type conversion functions

Test	Convert
`is.numeric()`	`as.numeric()`
`is.character()`	`as.character()`
`is.vector()`	`as.vector()`
`is.matrix()`	`as.matrix()`
`is.data.frame()`	`as.data.frame()`
`is.factor()`	`as.factor()`
`is.logical()`	`as.logical()`

Functions of the form `is.`*`datatype`*`()` return `TRUE` or `FALSE`, whereas `as.`*`datatype`*`()` converts the argument to that type. The following listing provides an example.

Listing 4.5 Converting from one data type to another

```
> a <- c(1,2,3)
> a
[1] 1 2 3
> is.numeric(a)
[1] TRUE
> is.vector(a)
[1] TRUE

> a <- as.character(a)
> a
[1] "1" "2" "3"
> is.numeric(a)
[1] FALSE
> is.vector(a)
[1] TRUE
> is.character(a)
[1] TRUE
```

When combined with the flow controls (such as `if-then`) that we'll discuss in chapter 5, the `is.`*`datatype`*`()` function can be a powerful tool, allowing you to handle data in different ways, depending on its type. Additionally, some R functions require data of a specific type (character or numeric, matrix or data frame) and the `as.`*`datatype`*`()` will let you transform your data into the format required prior to analyses.

4.8 Sorting data

Sometimes, viewing a dataset in a sorted order can tell you quite a bit about the data. For example, which managers are most deferential? To sort a data frame in R, use the `order()` function. By default, the sorting order is ascending. Prepend the sorting variable with a minus sign to indicate a descending order. The following examples illustrate sorting with the leadership data frame.

The statement

```
newdata <- leadership[order(leadership$age),]
```

creates a new dataset containing rows sorted from youngest manager to oldest manager. The statement

```
attach(leadership)
newdata <- leadership[order(gender, age),]
detach(leadership)
```

sorts the rows into female followed by male, and youngest to oldest within each gender.

Finally,

```
attach(leadership)
newdata <-leadership[order(gender, -age),]
detach(leadership)
```

sorts the rows by gender, and then from oldest to youngest manager within each gender.

4.9 Merging datasets

If your data exist in multiple locations, you'll need to combine them before moving forward. This section shows you how to add columns (variables) and rows (observations) to a data frame.

4.9.1 Adding columns

To merge two data frames (datasets) horizontally, you use the `merge()` function. In most cases, two data frames are joined by one or more common key variables (that is an inner join). For example:

```
total <- merge(dataframeA, dataframeB, by="ID")
```

merges `dataframeA` and `dataframeB` by `ID`. Similarly,

```
total <- merge(dataframeA, dataframeB, by=c("ID","Country"))
```

merges the two data frames by `ID` and `Country`. Horizontal joins like this are typically used to add variables to a data frame.

> **NOTE** If you're joining two matrices or data frames horizontally and don't need to specify a common key, you can use the `cbind()` function:
>
> ```
> total <- cbind(A, B)
> ```
>
> This function will horizontally concatenate the objects A and B. For the function to work properly, each object has to have the same number of rows and be sorted in the same order.

4.9.2 Adding rows

To join two data frames (datasets) vertically, use the `rbind()` function:

```
total <- rbind(dataframeA, dataframeB)
```

The two data frames must have the same variables, but they don't have to be in the same order. If `dataframeA` has variables that `dataframeB` doesn't, then before joining them do one of the following:

- Delete the extra variables in `dataframeA`
- Create the additional variables in `dataframeB` and set them to `NA` (missing)

Vertical concatenation is typically used to add observations to a data frame.

4.10 Subsetting datasets

R has powerful indexing features for accessing the elements of an object. These features can be used to select and exclude variables, observations, or both. The following sections demonstrate several methods for keeping or deleting variables and observations.

4.10.1 Selecting (keeping) variables

It's a common practice to create a new dataset from a limited number of variables chosen from a larger dataset. In chapter 2, you saw that the elements of a data frame are accessed using the notation *`dataframe[row indices, column indices]`*. You can use this to select variables. For example:

```
newdata <- leadership[, c(6:10)]
```

selects variables `q1`, `q2`, `q3`, `q4`, and `q5` from the `leadership` data frame and saves them to the data frame `newdata`. Leaving the row indices blank (,) selects all the rows by default.

The statements

```
myvars <- c("q1", "q2", "q3", "q4", "q5")
newdata <-leadership[myvars]
```

accomplish the same variable selection. Here, variable names (in quotes) have been entered as column indices, thereby selecting the same columns.

Finally, you could've used

```
myvars <- paste("q", 1:5, sep="")
newdata <- leadership[myvars]
```

This example uses the `paste()` function to create the same character vector as in the previous example. The `paste()` function will be covered in chapter 5.

4.10.2 Excluding (dropping) variables

There are many reasons to exclude variables. For example, if a variable has several missing values, you may want to drop it prior to further analyses. Let's look at some methods of excluding variables.

You could exclude variables `q3` and `q4` with the statements

```
myvars <- names(leadership) %in% c("q3", "q4")
newdata <- leadership[!myvars]
```

In order to understand why this works, you need to break it down:

1. `names(leadership)` produces a character vector containing the variable names. `c("managerID","testDate","country","gender","age","q1", "q2","q3","q4","q5")`.
2. `names(leadership) %in% c("q3", "q4")` returns a logical vector with `TRUE` for each element in `names(leadership)` that matches `q3` or `q4` and `FALSE` otherwise. `c(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)`.
3. The not (`!`) operator reverses the logical values `c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE)`.
4. `leadership[c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE)]` selects columns with `TRUE` logical values, so `q3` and `q4` are excluded.

Knowing that `q3` and `q4` are the 8th and 9th variable, you could exclude them with the statement

```
newdata <- leadership[c(-8,-9)]
```

This works because prepending a column index with a minus sign (`-`) excludes that column.

Finally, the same deletion can be accomplished via

```
leadership$q3 <- leadership$q4 <- NULL
```

Here you set columns `q3` and `q4` to undefined (`NULL`). Note that `NULL` isn't the same as `NA` (missing).

Dropping variables is the converse of keeping variables. The choice will depend on which is easier to code. If there are many variables to drop, it may be easier to keep the ones that remain, or vice versa.

4.10.3 Selecting observations

Selecting or excluding observations (rows) is typically a key aspect of successful data preparation and analysis. Several examples are given in the following listing.

Listing 4.6 Selecting observations

```
newdata <- leadership[1:3,]

newdata <- leadership[which(leadership$gender=="M" &
                            leadership$age > 30),]


attach(leadership)
newdata <- leadership[which(gender=='M' & age > 30),]
detach(leadership)
```

In each of these examples, you provide the row indices and leave the column indices blank (therefore choosing all columns). In the first example, you ask for rows 1 through 3 (the first three observations).

In the second example, you select all men over 30. Let's break down this line of code in order to understand it:

1. The logical comparison `leadership$gender=="M"` produces the vector `c(TRUE, FALSE, FALSE, TRUE, FALSE)`.
2. The logical comparison `leadership$age > 30` produces the vector `c(TRUE, TRUE, FALSE, TRUE, TRUE)`.
3. The logical comparison `c(TRUE, FALSE, FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE, TRUE, TRUE)` produces the vector `c(TRUE, FALSE, FALSE, TRUE, FALSE)`.
4. The function `which()` gives the indices of a vector that are `TRUE`. Thus, `which(c(TRUE, FALSE, FALSE, TRUE, FALSE))` produces the vector `c(1, 4)`.
5. `leadership[c(1,4),]` selects the first and fourth observations from the data frame. This meets our selection criteria (men over 30).

In the third example, the `attach()` function is used so that you don't have to prepend the variable names with the data frame names.

At the beginning of this chapter, I suggested that you might want to limit your analyses to observations collected between January 1, 2009 and December 31, 2009. How can you do this? Here's one solution:

```
leadership$date <- as.Date(leadership$date, "%m/%d/%y")
startdate <- as.Date("2009-01-01")
enddate   <- as.Date("2009-10-31")
newdata <- leadership[which(leadership$date >= startdate &
leadership$date <= enddate),]
```

Convert the date values read in originally as character values to date values using the format mm/dd/yy. Then, create starting and ending dates. Because the default for the `as.Date()` function is yyyy-mm-dd, you don't have to supply it here. Finally, select cases meeting your desired criteria as you did in the previous example.

4.10.4 The subset() function

The examples in the previous two sections are important because they help describe the ways in which logical vectors and comparison operators are interpreted within R. Understanding how these examples work will help you to interpret R code in general. Now that you've done things the hard way, let's look at a shortcut.

The `subset` function is probably the easiest way to select variables and observations. Here are two examples:

```
newdata <- subset(leadership, age >= 35 | age < 24,
                  select=c(q1, q2, q3, q4))

newdata <- subset(leadership, gender=="M" & age > 25,
                  select=gender:q4)
```

In the first example, you select all rows that have a value of `age` greater than or equal to 35 *or* `age` less than 24. You keep the variables `q1` through `q4`. In the second example,

you select all men over the age of 25 and you keep variables `gender` through `q4` (`gender`, `q4`, and all columns between them). You've seen the `colon` operator `from:to` in chapter 2. Here, it provides all variables in a data frame between the `from` variable and the `to` variable, inclusive.

4.10.5 Random samples

Sampling from larger datasets is a common practice in data mining and machine learning. For example, you may want to select two random samples, creating a predictive model from one and validating its effectiveness on the other. The `sample()` function enables you to take a random sample (with or without replacement) of size *n* from a dataset.

You could take a random sample of size 3 from the leadership dataset using the statement

```
mysample <- leadership[sample(1:nrow(leadership), 3, replace=FALSE),]
```

The first argument to the `sample()` function is a vector of elements to choose from. Here, the vector is 1 to the number of observations in the data frame. The second argument is the number of elements to be selected, and the third argument indicates sampling without replacement. The `sample()` function returns the randomly sampled elements, which are then used to select rows from the data frame.

GOING FURTHER

R has extensive facilities for sampling, including drawing and calibrating survey samples (see the `sampling` package) and analyzing complex survey data (see the `survey` package). Other methods that rely on sampling, including bootstrapping and resampling statistics, are described in chapter 11.

4.11 Using SQL statements to manipulate data frames

Until now, you've been using R statements to manipulate data. But many data analysts come to R well versed in Structured Query Language (SQL). It would be a shame to lose all that accumulated knowledge. Therefore, before we end, let me briefly mention the existence of the `sqldf` package. (If you're unfamiliar with SQL, please feel free to skip this section.)

After downloading and installing the package (`install.packages("sqldf")`), you can use the `sqldf()` function to apply SQL `SELECT` statements to data frames. Two examples are given in the following listing.

Listing 4.7 Using SQL statements to manipulate data frames

```
> library(sqldf)
> newdf <- sqldf("select * from mtcars where carb=1 order by mpg",
                 row.names=TRUE)
> newdf
               mpg cyl  disp  hp drat   wt qsec vs am gear carb
Valiant       18.1   6 225.0 105 2.76 3.46 20.2  1  0    3    1
```

```
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.21 19.4  1  0    3    1
Toyota Corona  21.5   4 120.1  97 3.70 2.46 20.0  1  0    3    1
Datsun 710     22.8   4 108.0  93 3.85 2.32 18.6  1  1    4    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.94 18.9  1  1    4    1
Fiat 128       32.4   4  78.7  66 4.08 2.20 19.5  1  1    4    1
Toyota Corolla 33.9   4  71.1  65 4.22 1.83 19.9  1  1    4    1

> sqldf("select avg(mpg) as avg_mpg, avg(disp) as avg_disp, gear
              from mtcars where cyl in (4, 6) group by gear")
  avg_mpg avg_disp gear
1    20.3      201    3
2    24.5      123    4
3    25.4      120    5
```

In the first example, you selected all the variables (columns) from the data frame `mtcars`, kept only automobiles (rows) with one carburetor (`carb`), sorted the automobiles in ascending order by `mpg`, and saved the results as the data frame `newdf`. The option `row.names=TRUE` carried the row names from the original data frame over to the new one. In the second example, you printed the mean `mpg` and `disp` within each level of `gear` for automobiles with four or six cylinders (`cyl`).

Experienced SQL users will find the `sqldf` package a useful adjunct to data management in R. See the project home page (http://code.google.com/p/sqldf/) for more details.

4.12 Summary

We covered a great deal of ground in this chapter. We looked at the way R stores missing and date values and explored various ways of handling them. You learned how to determine the data type of an object and how to convert it to other types. You used simple formulas to create new variables and recode existing variables. I showed you how to sort your data and rename your variables. You learned how to merge your data with other datasets both horizontally (adding variables) and vertically (adding observations). Finally, we discussed how to keep or drop variables and how to select observations based on a variety of criteria.

In the next chapter, we'll look at the myriad of arithmetic, character, and statistical functions that R makes available for creating and transforming variables. After exploring ways of controlling program flow, you'll see how to write your own functions. We'll also explore how you can use these functions to aggregate and summarize your data.

By the end of chapter 5 you'll have most of the tools necessary to manage complex datasets. (And you'll be the envy of data analysts everywhere!)