`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.