# 1

# Basics

The purpose of this chapter is to get you started using R. It is assumed that you have a working installation of the software and of the `ISwR` package that contains the data sets for this book. Instructions for obtaining and installing the software are given in Appendix A.

The text that follows describes R version 2.6.2. As of this writing, that is the latest version of R. As far as possible, I present the issues in a way that is independent of the operating system in use and assume that the reader has the elementary operational knowledge to select from menus, move windows around, etc. I do, however, make exceptions where I am aware of specific difficulties with a particular platform or specific features of it.

## 1.1 First steps

This section gives an introduction to the R computing environment and walks you through its most basic features.

Starting R is straightforward, but the method will depend on your computing platform. You will be able to launch it from a system menu, by double-clicking an icon, or by entering the command "`R`" at the system command line. This will either produce a console window or cause R to start up as an interactive program in the current terminal window. In

P. Dalgaard, *Introductory Statistics with R*,
DOI: 10.1007/978-0-387-79054-1_1, 

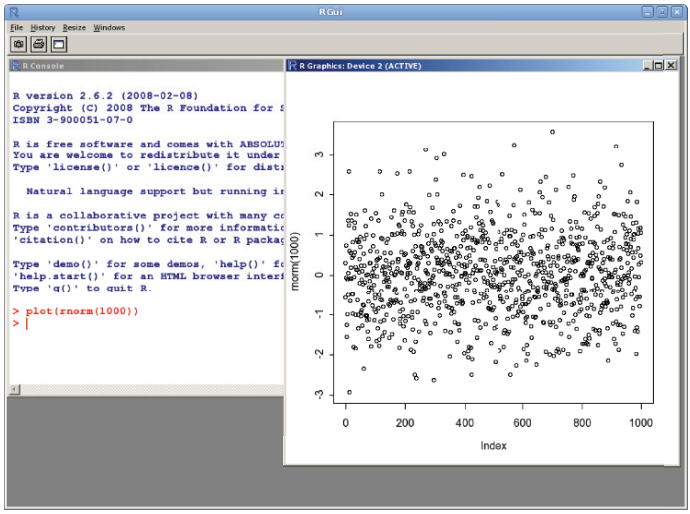


Figure 1.1. Screen image of R for Windows.

either case, R works fundamentally by the question-and-answer model: You enter a line with a command and press Enter (↩). Then the program does something, prints the result if relevant, and asks for more input. When R is ready for input, it prints out its prompt, a ">". It is possible to use R as a text-only application, and also in batch mode, but for the purposes of this chapter, I assume that you are sitting at a graphical workstation.

All the examples in this book should run if you type them in exactly as printed, *provided* that you have the `ISwR` package not only installed but also loaded into your current search path. This is done by entering

```
> library(ISwR)
```

at the command prompt. You do not need to understand what the command does at this point. It is explained in Section 2.1.5.

For a first impression of what R can do, try typing the following:

```
> plot(rnorm(1000))
```

This command draws 1000 numbers at random from the normal distribution (rnorm = *r*andom *norm*al) and plots them in a pop-up graphics window. The result on a Windows machine can be seen in Figure 1.1.

Of course, you are not expected at this point to guess that you would obtain this result in that particular way. The example is chosen because it shows several components of the user interface in action. Before the style

of commands will fall naturally, it is necessary to introduce some concepts and conventions through simpler examples.

Under Windows, the graphics window will have taken the keyboard focus at this point. Click on the console to make it accept further commands.

### 1.1.1 An overgrown calculator

One of the simplest possible tasks in R is to enter an arithmetic expression and receive a result. (The second line is the answer from the machine.)

```
> 2 + 2
[1] 4
```

So the machine knows that 2 plus 2 makes 4. Of course, it also knows how to do other standard calculations. For instance, here is how to compute $e^{-2}$:

```
> exp(-2)
[1] 0.1353353
```

The `[1]` in front of the result is part of R's way of printing numbers and vectors. It is not useful here, but it becomes so when the result is a longer vector. The number in brackets is the index of the first number on that line. Consider the case of generating 15 random numbers from a normal distribution:

```
> rnorm(15)
 [1] -0.18326112 -0.59753287 -0.67017905  0.16075723  1.28199575
 [6]  0.07976977  0.13683303  0.77155246  0.85986694 -1.01506772
[11] -0.49448567  0.52433026  1.07732656  1.09748097 -1.09318582
```

Here, for example, the `[6]` indicates that 0.07976977 is the sixth element in the vector. (For typographical reasons, the examples in this book are made with a shortened line width. If you try it on your own machine, you will see the values printed with six numbers per line rather than five. The numbers themselves will also be different since random number generation is involved.)

### 1.1.2 Assignments

Even on a calculator, you will quickly need some way to store intermediate results, so that you do not have to key them in over and over again. R, like other computer languages, has *symbolic variables*, that is names that

can be used to represent values. To assign the value 2 to the variable `x`, you can enter

```
> x <- 2
```

The two characters <− should be read as a single symbol: an arrow pointing to the variable to which the value is assigned. This is known as the *assignment operator.* Spacing around operators is generally disregarded by R, but notice that adding a space in the middle of a <− changes the meaning to "less than" followed by "minus" (conversely, omitting the space when comparing a variable to a negative number has unexpected consequences!).

There is no immediately visible result, but from now on, `x` has the value 2 and can be used in subsequent arithmetic expressions.

```
> x
[1] 2
> x + x
[1] 4
```

Names of variables can be chosen quite freely in R. They can be built from letters, digits, and the period (dot) symbol. There is, however, the limitation that the name must not start with a digit or a period followed by a digit. Names that start with a period are special and should be avoided. A typical variable name could be `height.1yr`, which might be used to describe the height of a child at the age of 1 year. Names are case-sensitive: `WT` and `wt` do not refer to the same variable.

Some names are already used by the system. This can cause some confusion if you use them for other purposes. The worst cases are the single-letter names `c`, `q`, `t`, `C`, `D`, `F`, `I`, and `T`, but there are also `diff`, `df`, and `pt`, for example. Most of these are functions and do not usually cause trouble when used as variable names. However, `F` and `T` are the standard abbreviations for `FALSE` and `TRUE` and no longer work as such if you redefine them.

### *1.1.3 Vectorized arithmetic*

You cannot do much statistics on single numbers! Rather, you will look at data from a group of patients, for example. One strength of R is that it can handle entire *data vectors* as single objects. A data vector is simply an array of numbers, and a vector variable can be constructed like this:

```
> weight <- c(60, 72, 57, 90, 95, 72)
> weight
[1] 60 72 57 90 95 72
```

The construct `c(...)` is used to define vectors. The numbers are made up but might represent the weights (in kg) of a group of normal men.

This is neither the only way to enter data vectors into R nor is it generally the preferred method, but short vectors are used for many other purposes, and the `c(...)` construct is used extensively. In Section 2.4, we discuss alternative techniques for reading data. For now, we stick to a single method.

You can do calculations with vectors just like ordinary numbers, as long as they are of the same length. Suppose that we also have the heights that correspond to the weights above. The body mass index (BMI) is defined for each person as the weight in kilograms divided by the square of the height in meters. This could be calculated as follows:

```
> height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
> bmi <- weight/height^2
> bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```

Notice that the operation is carried out elementwise (that is, the first value of `bmi` is $60/1.75^2$ and so forth) and that the `^` operator is used for raising a value to a power. (On some keyboards, `^` is a "dead key" and you will have to press the spacebar afterwards to make it show.)

It is in fact possible to perform arithmetic operations on vectors of different length. We already used that when we calculated the `height^2` part above since `2` has length 1. In such cases, the shorter vector is *recycled*. This is mostly used with vectors of length 1 (scalars) but sometimes also in other cases where a repeating pattern is desired. A warning is issued if the longer vector is not a multiple of the shorter in length.

These conventions for vectorized calculations make it very easy to specify typical statistical calculations. Consider, for instance, the calculation of the mean and standard deviation of the `weight` variable.

First, calculate the mean, $\bar{x} = \sum x_i/n$:

```
> sum(weight)
[1] 446
> sum(weight)/length(weight)
[1] 74.33333
```

Then save the mean in a variable `xbar` and proceed with the calculation of SD $= \sqrt{(\sum(x_i - \bar{x})^2)/(n-1)}$. We do this in steps to see the individual components. The deviations from the mean are

```
> xbar <- sum(weight)/length(weight)
> weight - xbar
```

```
[1] -14.333333  -2.333333 -17.333333  15.666667  20.666667
[6]  -2.333333
```

Notice how `xbar`, which has length 1, is recycled and subtracted from each element of `weight`. The squared deviations will be

```
> (weight - xbar)^2
[1] 205.444444   5.444444 300.444444 245.444444 427.111111
[6]   5.444444
```

Since this command is quite similar to the one before it, it is convenient to enter it by editing the previous command. On most systems running R, the previous command can be recalled with the up-arrow key.

The sum of squared deviations is similarly obtained with

```
> sum((weight - xbar)^2)
[1] 1189.333
```

and all in all the standard deviation becomes

```
> sqrt(sum((weight - xbar)^2)/(length(weight) - 1))
[1] 15.42293
```

Of course, since R is a statistical program, such calculations are already built into the program, and you get the same results just by entering

```
> mean(weight)
[1] 74.33333
> sd(weight)
[1] 15.42293
```

### *1.1.4 Standard procedures*

As a slightly more complicated example of what R can do, consider the following: The rule of thumb is that the BMI for a normal-weight individual should be between 20 and 25, and we want to know if our data deviate systematically from that. You might use a one-sample *t* test to assess whether the six persons' BMI can be assumed to have mean 22.5 given that they come from a normal distribution. To this end, you can use the function `t.test`. (You might not know the theory of the *t* test yet. The example is included here mainly to give some indication of what "real" statistical output looks like. A thorough description of `t.test` is given in Chapter 5.)

```
> t.test(bmi, mu=22.5)
        One Sample t-test
data:  bmi
t = 0.3449, df = 5, p-value = 0.7442
alternative hypothesis: true mean is not equal to 22.5
95 percent confidence interval:
 18.41734 27.84791
sample estimates:
mean of x
 23.13262
```

The argument `mu=22.5` attaches a value to the formal argument `mu`, which represents the Greek letter $\mu$ conventionally used for the theoretical mean. If this is not given, `t.test` would use the default `mu=0`, which is not of interest here.

For a test like this, we get a more extensive printout than in the earlier examples. The details of the output are explained in Chapter 5, but you might focus on the $p$-value which is used for testing the hypothesis that the mean is 22.5. The $p$-value is not small, indicating that it is not at all unlikely to get data like those observed if the mean were in fact 22.5. (Loosely speaking; actually $p$ is the probability of obtaining a $t$ value bigger than 0.3449 or less than $-0.3449$.) However, you might also look at the 95% confidence interval for the true mean. This interval is quite wide, indicating that we really have very little information about the true mean.

### *1.1.5 Graphics*

One of the most important aspects of the presentation and analysis of data is the generation of proper graphics. R — like S before it — has a model for constructing plots that allows simple production of standard plots as well as fine control over the graphical components.

If you want to investigate the relation between `weight` and `height`, the first idea is to plot one versus the other. This is done by

```
> plot(height,weight)
```

leading to Figure 1.2.

You will often want to modify the drawing in various ways. To that end, there are a wealth of plotting parameters that you can set. As an example, let us try changing the plotting symbol using the keyword `pch` ("plotting character") like this:

```
> plot(height, weight, pch=2)
```

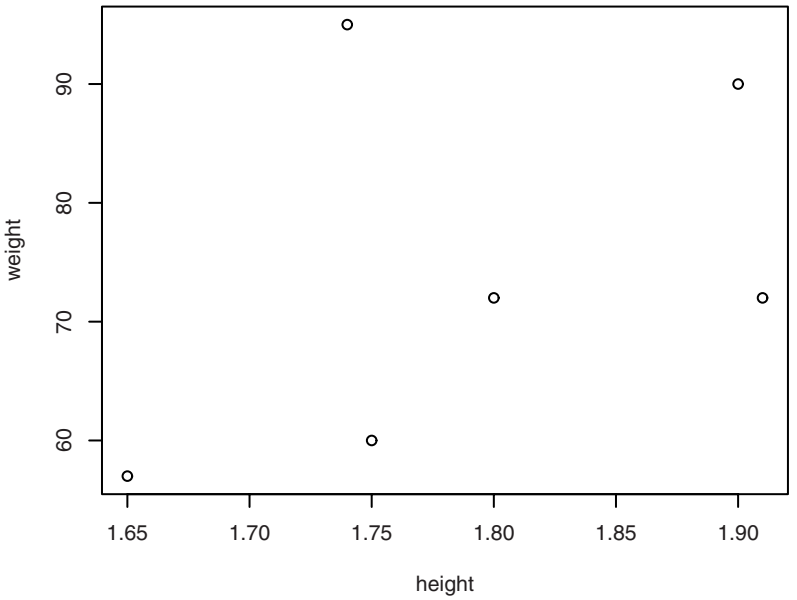


Figure 1.2. A simple *x*–*y* plot.

This gives the plot in Figure 1.3, with the points now marked with little triangles.

The idea behind the BMI calculation is that this value should be independent of the person's height, thus giving you a single number as an indication of whether someone is overweight and by how much. Since a normal BMI should be about 22.5, you would expect that $weight \approx 22.5 \times height^2$. Accordingly, you can superimpose a curve of expected weights at BMI 22.5 on the figure:

```
> hh <- c(1.65, 1.70, 1.75, 1.80, 1.85, 1.90)
> lines(hh, 22.5 * hh^2)
```

yielding Figure 1.4. The function `lines` will *add* $(x, y)$ values joined by straight lines to an existing plot.

The reason for defining a new variable (`hh`) with heights rather than using the original `height` vector is twofold. First, the relation between height and weight is a quadratic one and hence nonlinear, although it can be difficult to see on the plot. Since we are approximating a nonlinear curve with a piecewise linear one, it will be better to use points that are spread evenly along the *x*-axis than to rely on the distribution of the original data. Sec-

ond, since the values of `height` are not sorted, the line segments would not connect neighbouring points but would run back and forth between distant points.

## 1.2 R language essentials

This section outlines the basic aspects of the R language. It is necessary to do this in a slightly superficial manner, with some of the finer points glossed over. The emphasis is on items that are useful to know in interactive usage as opposed to actual programming, although a brief section on programming is included.

### *1.2.1 Expressions and objects*

The basic interaction mode in R is one of expression evaluation. The user enters an expression; the system evaluates it and prints the result. Some expressions are evaluated not for their result but for *side effects* such as

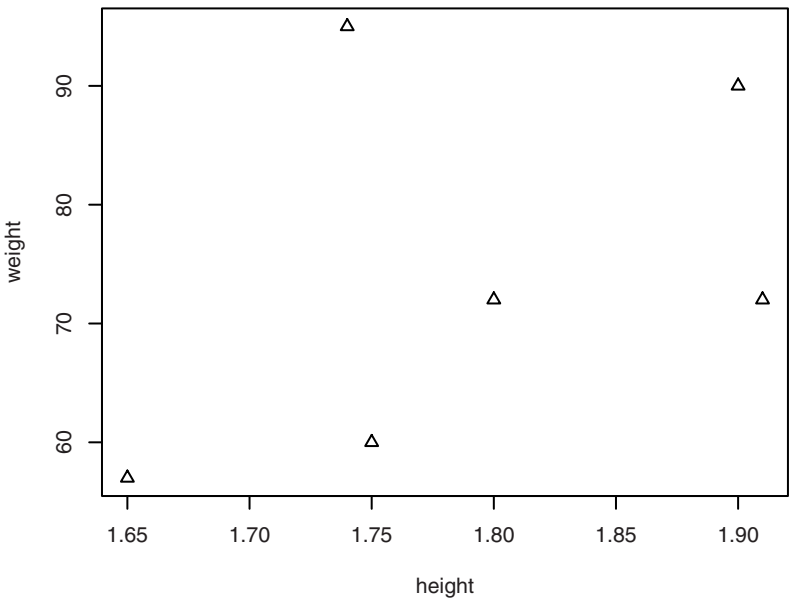


Figure 1.3. Plot with `pch = 2`.

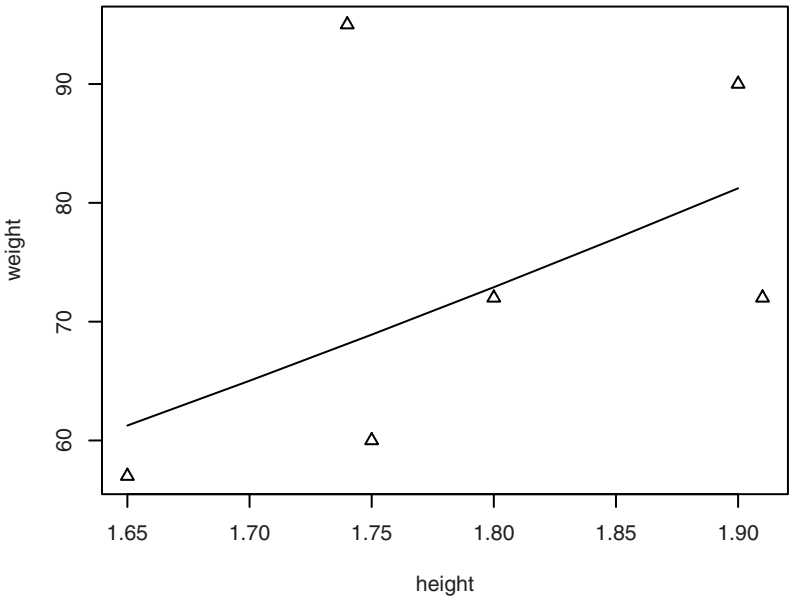


Figure 1.4. Superimposed reference curve, using `lines(...)`.

putting up a graphics window or writing to a file. All R expressions return a value (possibly `NULL`), but sometimes it is "invisible" and not printed.

Expressions typically involve variable references, operators such as `+`, and function calls, as well as some other items that have not been introduced yet.

Expressions work on *objects*. This is an abstract term for anything that can be assigned to a variable. R contains several different types of objects. So far, we have almost exclusively seen numeric vectors, but several other types are introduced in this chapter.

Although objects can be discussed abstractly, it would make a rather boring read without some indication of how to generate them and what to do with them. Conversely, much of the expression syntax makes little sense without knowledge of the objects on which it is intended to work. Therefore, the subsequent sections alternate between introducing new objects and introducing new language elements.

### 1.2.2 Functions and arguments

At this point, you have obtained an impression of the way R works, and we have already used some of the special terminology when talking about the plot *function*, etc. That is exactly the point: Many things in R are done using *function calls*, commands that look like an application of a mathematical function of one or several variables; for example, `log(x)` or `plot(height, weight)`.

The format is that a function name is followed by a set of parentheses containing one or more arguments. For instance, in `plot(height,weight)` the function name is `plot` and the arguments are `height` and `weight`. These are the *actual arguments*, which apply only to the current call. A function also has *formal arguments*, which get connected to actual arguments in the call.

When you write `plot(height, weight)`, R assumes that the first argument corresponds to the $x$-variable and the second one to the $y$-variable. This is known as *positional matching*. This becomes unwieldy if a function has a large number of arguments since you have to supply every one of them and remember their position in the sequence. Fortunately, R has methods to avoid this: Most arguments have sensible defaults and can be omitted in the standard cases, and there are nonpositional ways of specifying them when you need to depart from the default settings.

The `plot` function is in fact an example of a function that has a large selection of arguments in order to be able to modify symbols, line widths, titles, axis type, and so forth. We used the alternative form of specifying arguments when setting the plot symbol to triangles with `plot(height, weight, pch=2)`.

The `pch=2` form is known as a *named actual argument*, whose name can be matched against the formal arguments of the function and thereby allow *keyword matching* of arguments. The keyword `pch` was used to say that the argument is a specification of the plotting character. This type of function argument can be specified in arbitrary order. Thus, you can write `plot(y=weight,x=height)` and get the same plot as with `plot(x=height,y=weight)`.

The two kinds of argument specification — positional and named — can be mixed in the same call.

Even if there are no arguments to a function call, you have to write, for example, `ls()` for displaying the contents of the workspace. A common error is to leave off the parentheses, which instead results in the display of a piece of R code since `ls` entered by itself indicates that you want to see the definition of the function rather than execute it.

The *formal arguments* of a function are part of the function definition. The set of formal arguments to a function, for instance `plot.default` (which is the function that gets called when you pass `plot` an `x` argument for which no special plot method exists), may be seen with

```
> args(plot.default)
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes,
    panel.first = NULL, panel.last = NULL, asp = NA, ...)
```

Notice that most of the arguments have defaults, meaning that if you do not specify (say) the `type` argument, the function will behave as if you had passed `type="p"`. The `NULL` defaults for many of the arguments really serve as indicators that the argument is unspecified, allowing special behaviour to be defined inside the function. For instance, if they are not specified, the `xlab` and `ylab` arguments are constructed from the actual arguments passed as `x` and `y`. (There are some very fine points associated with this, but we do not go further into the topic.)

The triple-dot (`...`) argument indicates that this function will accept additional arguments of unspecified name and number. These are often meant to be passed on to other functions, although some functions treat it specially. For instance, in `data.frame` and `c`, the names of the `...`-arguments become the names of the elements of the result.

### *1.2.3 Vectors*

We have already seen numeric vectors. There are two further types, character vectors and logical vectors.

A *character vector* is a vector of text strings, whose elements are specified and printed in quotes:

```
> c("Huey","Dewey","Louie")
[1] "Huey"  "Dewey" "Louie"
```

It does not matter whether you use single- or double-quote symbols, as long as the left quote is the same as the right quote:

```
> c('Huey','Dewey','Louie')
[1] "Huey"  "Dewey" "Louie"
```

However, you should avoid the *acute accent* key (´), which is present on some keyboards. Double quotes are used throughout this book to prevent mistakes. *Logical vectors* can take the value `TRUE` or `FALSE` (or `NA`; see below). In input, you may use the convenient abbreviations `T` and `F` (if you

are careful not to redefine them). Logical vectors are constructed using the `c` function just like the other vector types:

```
> c(T,T,F,T)
[1]  TRUE  TRUE FALSE  TRUE
```

Actually, you will not often have to specify logical vectors in the manner above. It is much more common to use single logical values to turn an option on or off in a function call. Vectors of more than one value most often result from *relational expressions*:

```
> bmi > 25
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
```

We return to relational expressions and logical operations in the context of conditional selection in Section 1.2.12.

### *1.2.4 Quoting and escape sequences*

Quoted character strings require some special considerations: How, for instance, do you put a quote symbol inside a string? And what about special characters such as newlines? This is done using *escape sequences*. We shall look at those in a moment, but first it will be useful to observe the following.

There is a distinction between a text string and the way it is printed. When, for instance, you give the string `"Huey"`, it is a string of four characters, not six. The quotes are not actually part of the string, they are just there so that the system can tell the difference between a string and a variable name.

If you print a character vector, it usually comes out with quotes added to each element. There is a way to avoid this, namely to use the `cat` function. For instance,

```
> cat(c("Huey","Dewey","Louie"))
Huey Dewey Louie>
```

This prints the strings without quotes, just separated by a space character. There is no newline following the string, so the prompt (>) for the next line of input follows directly at the end of the line. (Notice that when the character vector is printed by `cat` there is no way of telling the difference from the single string `"Huey Dewey Louie"`.)

To get the system prompt onto the next line, you must include a newline character

```
> cat("Huey","Dewey","Louie", "\n")
Huey Dewey Louie
>
```

Here, `\n` is an example of an escape sequence. It actually represents a single character, the linefeed (LF), but is represented as two. The backslash (`\`) is known as the *escape character*. In a similar vein, you can insert quote characters with `\"`, as in

```
> cat("What is \"R\"?\n")
What is "R"?
```

There are also ways to insert other control characters and special glyphs, but it would lead us too far astray to discuss it in full detail. One important thing, though: What about the escape character itself? This, too, must be escaped, so to put a backslash in a string, you must double it. This is important to know when specifying file paths on Windows, see also Section 2.4.1.

### *1.2.5 Missing values*

In practical data analysis, a data point is frequently unavailable (the patient did not show up, an experiment failed, etc.). Statistical software needs ways to deal with this. R allows vectors to contain a special `NA` value. This value is carried through in computations so that operations on `NA` yield `NA` as the result. There are some special issues associated with the handling of missing values; we deal with them as we encounter them (see "missing values" in the index).

### *1.2.6 Functions that create vectors*

Here we introduce three functions, `c`, `seq`, and `rep`, that are used to create vectors in various situations.

The first of these, `c`, has already been introduced. It is short for "concatenate", joining items end to end, which is exactly what the function does:

```
> c(42,57,12,39,1,3,4)
[1] 42 57 12 39  1  3  4
```

You can also concatenate vectors of more than one element as in

```
> x <- c(1, 2, 3)
> y <- c(10, 20)
```

```
> c(x, y, 5)
[1]  1  2  3 10 20  5
```

However, you do not need to use `c` to create vectors of length 1. People sometimes type, for example, `c(1)`, but it is the same as plain `1`.

It is also possible to assign names to the elements. This modifies the way the vector is printed and is often used for display purposes.

```
> x <- c(red="Huey", blue="Dewey", green="Louie")
> x
    red    blue   green
 "Huey" "Dewey" "Louie"
```

(In this case, it *does* of course make sense to use `c` even for single-element vectors.)

The names can be extracted or set using `names`:

```
> names(x)
[1] "red"   "blue"  "green"
```

All elements of a vector have the same type. If you concatenate vectors of different types, they will be converted to the least "restrictive" type:

```
> c(FALSE, 3)
[1] 0 3
> c(pi, "abc")
[1] "3.14159265358979" "abc"
> c(FALSE, "abc")
[1] "FALSE" "abc"
```

That is, logical values may be converted to 0/1 or `"FALSE"`/`"TRUE"` and numbers converted to their printed representations.

The second function, `seq` ("sequence"), is used for equidistant series of numbers. Writing

```
> seq(4,9)
[1] 4 5 6 7 8 9
```

yields, as shown, the integers from 4 to 9. If you want a sequence in jumps of 2, write

```
> seq(4,10,2)
[1]  4  6  8 10
```

This kind of vector is frequently needed, particularly for graphics. For example, we previously used `c(1.65,1.70,1.75,1.80,1.85,1.90)` to define the $x$-coordinates for a curve, something that could also have been

written `seq(1.65,1.90,0.05)` (the advantage of using `seq` might have been more obvious if the heights had been in steps of 1 cm rather than 5 cm!).

The case with step size equal to 1 can also be written using a special syntax:

```
> 4:9
[1] 4 5 6 7 8 9
```

The above is exactly the same as `seq(4,9)`, only easier to read.

The third function, `rep` ("replicate"), is used to generate repeated values. It is used in two variants, depending on whether the second argument is a vector or a single number:

```
> oops <- c(7,9,13)
> rep(oops,3)
[1]  7  9 13  7  9 13  7  9 13
> rep(oops,1:3)
[1]  7  9  9 13 13 13
```

The first of the function calls above repeats the entire vector `oops` three times. The second call has the number 3 replaced by a vector with the three values (1, 2, 3); these values correspond to the elements of the `oops` vector, indicating that 7 should be repeated once, 9 twice, and 13 three times. The `rep` function is often used for things such as group codes: If it is known that the first 10 observations are men and the last 15 are women, you can use

```
> rep(1:2,c(10,15))
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

to form a vector that for each observation indicates whether it is from a man or a woman.

The special case where there are equally many replications of each value can be obtained using the `each` argument. E.g., `rep(1:2,each=10)` is the same as `rep(1:2,c(10,10))`.

### *1.2.7 Matrices and arrays*

A *matrix* in mathematics is just a two-dimensional array of numbers. Matrices are used for many purposes in theoretical and practical statistics, but it is not assumed that the reader is familiar with matrix algebra, so many special operations on matrices, including matrix multiplication, are skipped. (The document "An Introduction to R", which comes with

the installation, outlines these items quite well.) However, matrices and also higher-dimensional arrays do get used for simpler purposes as well, mainly to hold tables, so an elementary description is in order.

In R, the matrix notion is extended to elements of any type, so you could have, for instance, a matrix of character strings. Matrices and arrays are represented as vectors with dimensions:

```
> x <- 1:12
> dim(x) <- c(3,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The `dim` assignment function sets or changes the *dimension attribute* of `x`, causing R to treat the vector of 12 numbers as a $3 \times 4$ matrix. Notice that the storage is column-major; that is, the elements of the first column are followed by those of the second, etc.

A convenient way to create matrices is to use the `matrix` function:

```
> matrix(1:12,nrow=3,byrow=T)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Notice how the `byrow=T` switch causes the matrix to be filled in a rowwise fashion rather than columnwise.

Useful functions that operate on matrices include `rownames`, `colnames`, and the transposition function `t` (notice the lowercase `t` as opposed to uppercase `T` for `TRUE`), which turns rows into columns and vice versa:

```
> x <- matrix(1:12,nrow=3,byrow=T)
> rownames(x) <- LETTERS[1:3]
> x
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
> t(x)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
```

The character vector `LETTERS` is a built-in variable that contains the capital letters A–Z. Similar useful vectors are `letters`, `month.name`, and `month.abb` with lowercase letters, month names, and abbreviated month names.

You can "glue" vectors together, columnwise or rowwise, using the `cbind` and `rbind` functions.

```
> cbind(A=1:4,B=5:8,C=9:12)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
> rbind(A=1:4,B=5:8,C=9:12)
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
```

We return to table operations in Section 4.5, which discusses tabulation of variables in a data set.

### *1.2.8 Factors*

It is common in statistical data to have categorical variables, indicating some subdivision of data, such as social class, primary diagnosis, tumor stage, Tanner stage of puberty, etc. Typically, these are input using a numeric code.

Such variables should be specified as *factors* in R. This is a data structure that (among other things) makes it possible to assign meaningful names to the categories.

There are analyses where it is essential for R to be able to distinguish between categorical codes and variables whose values have a direct numerical meaning (see Chapter 7).

The terminology is that a factor has a set of *levels* — say four levels for concreteness. Internally, a four-level factor consists of two items: (a) a vector of integers between 1 and 4 and (b) a character vector of length 4 containing strings describing what the four levels are. Let us look at an example:

```
> pain <- c(0,3,2,2,1)
> fpain <- factor(pain,levels=0:3)
> levels(fpain) <- c("none","mild","medium","severe")
```

The first command creates a numeric vector `pain`, encoding the pain levels of five patients. We wish to treat this as a categorical variable, so we create a factor `fpain` from it using the function `factor`. This is called with one argument in addition to `pain`, namely `levels=0:3`, which indicates that the *input* coding uses the values 0–3. The latter can in principle be left out since R by default uses the values in `pain`, suitably sorted, but it is a good habit to retain it; see below. The effect of the final line is that the level names are changed to the four specified character strings.

The result should be apparent from the following:

```
> fpain
[1] none   severe medium medium mild
Levels:  none mild medium severe
> as.numeric(fpain)
[1] 1 4 3 3 2
> levels(fpain)
[1] "none"   "mild"   "medium" "severe"
```

The function `as.numeric` extracts the numerical coding as numbers 1–4 and `levels` extracts the names of the levels. Notice that the original input coding in terms of numbers 0–3 has disappeared; the internal representation of a factor always uses numbers starting at 1.

R also allows you to create a special kind of factor in which the levels are ordered. This is done using the `ordered` function, which works similarly to `factor`. These are potentially useful in that they distinguish nominal and ordinal variables from each other (and arguably `text.pain` above ought to have been an ordered factor). Unfortunately, R defaults to treating the levels as if they were *equidistant* in the modelling code (by generating polynomial contrasts), so it may be better to ignore ordered factors at this stage.

### *1.2.9 Lists*

It is sometimes useful to combine a collection of objects into a larger composite object. This can be done using *lists*.

You can construct a list from its components with the function `list`.

As an example, consider a set of data from Altman (1991, p. 183) concerning pre- and postmenstrual energy intake in a group of women. We can place these data in two vectors as follows:

```
> intake.pre <- c(5260,5470,5640,6180,6390,
+ 6515,6805,7515,7515,8230,8770)
> intake.post <- c(3910,4220,3885,5160,5645,
+ 4680,5265,5975,6790,6900,7335)
```

Notice how input lines can be broken and continue on the next line. If you press the Enter key while an expression is syntactically incomplete, R will assume that the expression continues on the next line and will change its normal > prompt to the *continuation prompt* +. This often happens inadvertently due to a forgotten parenthesis or a similar problem; in such cases, either complete the expression on the next line or press ESC (Windows and Macintosh) or Ctrl-C (Unix). The "Stop" button can also be used under Windows.

To combine these individual vectors into a list, you can say

```
> mylist <- list(before=intake.pre,after=intake.post)
> mylist
$before
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770

$after
 [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335
```

The components of the list are named according to the argument names used in `list`. Named components may be extracted like this:

```
> mylist$before
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
```

Many of R's built-in functions compute more than a single vector of values and return their results in the form of a list.

### *1.2.10 Data frames*

A data frame corresponds to what other statistical packages call a "data matrix" or a "data set". It is a list of vectors and/or factors of the same length that are related "across" such that data in the same position come from the same experimental unit (subject, animal, etc.). In addition, it has a unique set of row names.

You can create data frames from preexisting variables:

```
> d <- data.frame(intake.pre,intake.post)
> d
   intake.pre intake.post
1        5260        3910
2        5470        4220
3        5640        3885
4        6180        5160
5        6390        5645
6        6515        4680
7        6805        5265
```

```
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
```

Notice that these data are paired, that is, the same woman has an intake of 5260 kJ premenstrually and 3910 kJ postmenstrually.

As with lists, components (i.e., individual variables) can be accessed using the `$` notation:

```
> d$intake.pre
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
```

### *1.2.11 Indexing*

If you need a particular element in a vector, for instance the premenstrual energy intake for woman no. 5, you can do

```
> intake.pre[5]
[1] 6390
```

The brackets are used for selection of data, also known as *indexing* or *subsetting*. This also works on the left-hand side of an assignment (so that you can say, for instance, `intake.pre[5] <- 6390`) if you want to modify elements of a vector.

If you want a subvector consisting of data for more than one woman, for instance nos. 3, 5, and 7, you can index with a vector:

```
> intake.pre[c(3,5,7)]
[1] 5640 6390 6805
```

Note that it is necessary to use the `c(...)`-construction to define the vector consisting of the three numbers 3, 5, and 7. `intake.pre[3,5,7]` would mean something completely different. It would specify indexing into a three-dimensional array.

Of course, indexing with a vector also works if the index vector is stored in a variable. This is useful when you need to index several variables in the same way.

```
> v <- c(3,5,7)
> intake.pre[v]
[1] 5640 6390 6805
```

It is also worth noting that to get a sequence of elements, for instance the first five, you can use the `a:b` notation:

```
> intake.pre[1:5]
[1] 5260 5470 5640 6180 6390
```

A neat feature of R is the possibility of negative indexing. You can get all observations *except* nos. 3, 5, and 7 by writing

```
> intake.pre[-c(3,5,7)]
[1] 5260 5470 6180 6515 7515 7515 8230 8770
```

It is not possible to mix positive and negative indices. That would be highly ambiguous.

### *1.2.12 Conditional selection*

We saw in Section 1.2.11 how to extract data using one or several indices. In practice, you often need to extract data that satisfy certain criteria, such as data from the males or the prepubertal or those with chronic diseases, etc. This can be done simply by inserting a relational expression instead of the index,

```
> intake.post[intake.pre > 7000]
[1] 5975 6790 6900 7335
```

yielding the postmenstrual energy intake for the four women who had an energy intake above 7000 kJ premenstrually.

Of course, this kind of expression makes sense only if the variables that go into the relational expression have the same length as the variable being indexed.

The comparison operators available are < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to), and != (not equal to). Notice that a double equal sign is used for testing equality. This is to avoid confusion with the = symbol used to match keywords with function arguments. Also, the != operator is new to some; the ! symbol indicates negation. The same operators are used in the C programming language.

To combine several expressions, you can use the logical operators & (logical "and"), | (logical "or"), and ! (logical "not"). For instance, we find the postmenstrual intake for women with a premenstrual intake between 7000 and 8000 kJ with

```
> intake.post[intake.pre > 7000 & intake.pre <= 8000]
[1] 5975 6790
```

There are also `&&` and `||`, which are used for flow control in R programming. However, their use is beyond what we discuss here.

It may be worth taking a closer look at what actually happens when you use a logical expression as an index. The result of the logical expression is a logical vector as described in Section 1.2.3:

```
> intake.pre > 7000 & intake.pre <= 8000
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  FALSE
[11] FALSE
```

Indexing with a logical vector implies that you pick out the values where the logical vector is `TRUE`, so in the preceding example we got the 8th and 9th values in `intake.post`.

If missing values (`NA`; see Section 1.2.5) appear in an indexing vector, then R will create the corresponding elements in the result but set the values to `NA`.

In addition to the relational and logical operators, there are a series of functions that return a logical value. A particularly important one is `is.na(x)`, which is used to find out which elements of `x` are recorded as missing (`NA`).

Notice that there is a real need for `is.na` because you cannot make comparisons of the form `x==NA`. That simply gives `NA` as the result for any value of `x`. The result of a comparison with an unknown value is unknown!

### *1.2.13 Indexing of data frames*

We have already seen how it is possible to extract variables from a data frame by typing, for example, `d$intake.post`. However, it is also possible to use a notation that uses the matrix-like structure directly:

```
> d <- data.frame(intake.pre,intake.post)
> d[5,1]
[1] 6390
```

gives fifth row, first column (that is, the "pre" measurement for woman no. 5), and

```
> d[5,]
  intake.pre intake.post
5       6390        5645
```

gives *all* measurements for woman no. 5. Notice that the comma in `d[5,]` is required; without the comma, for example `d[2]`, you get the data frame

consisting of the second *column* of `d` (that is, more like `d[,2]`, which is the column itself).

Other indexing techniques also apply. In particular, it can be useful to extract all data for cases that satisfy some criterion, such as women with a premenstrual intake above 7000 kJ:

```
> d[d$intake.pre>7000,]
   intake.pre intake.post
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
```

Here we extracted the rows of the data frame where `intake.pre>7000`. Notice that the row names are those of the original data frame.

If you want to understand the details of this, it may be a little easier if it is divided into smaller steps. It could also have been done like this:

```
> sel <- d$intake.pre>7000
> sel
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[11]  TRUE
> d[sel,]
   intake.pre intake.post
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
```

What happens is that `sel` *(select)* becomes a logical vector with the value `TRUE` for to the four women consuming more than 7000 kJ premenstrually. Indexing as `d[sel,]` yields data from the rows where `sel` is `TRUE` and from all columns because of the empty field after the comma.

It is often convenient to look at the first few cases in a data set. This can be done with indexing, like this:

```
> d[1:2,]
   intake.pre intake.post
1        5260        3910
2        5470        4220
```

This is such a frequent occurrence that a convenience function called `head` exists. By default, it shows the first six lines.

```
> head(d)
   intake.pre intake.post
1        5260        3910
2        5470        4220
```

```
3        5640        3885
4        6180        5160
5        6390        5645
6        6515        4680
```

Similarly, `tail` shows the last part.

### 1.2.14 *Grouped data and data frames*

The natural way of storing grouped data in a data frame is to have the data themselves in one vector and parallel to that have a factor telling which data are from which group. Consider, for instance, the following data set on energy expenditure for lean and obese women.

```
> energy
   expend stature
1    9.21   obese
2    7.53    lean
3    7.48    lean
4    8.08    lean
5    8.09    lean
6   10.15    lean
7    8.40    lean
8   10.88    lean
9    6.13    lean
10   7.90    lean
11  11.51   obese
12  12.79   obese
13   7.05    lean
14  11.85   obese
15   9.97   obese
16   7.48    lean
17   8.79   obese
18   9.69   obese
19   9.68   obese
20   7.58    lean
21   9.19   obese
22   8.11    lean
```

This is a convenient format since it generalizes easily to data classified by multiple criteria. However, sometimes it is desirable to have data in a separate vector for each group. Fortunately, it is easy to extract these from the data frame:

```
> exp.lean <- energy$expend[energy$stature=="lean"]
> exp.obese <- energy$expend[energy$stature=="obese"]
```

Alternatively, you can use the `split` function, which generates a list of vectors according to a grouping.

```
> l <- split(energy$expend, energy$stature)
> l
$lean
 [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88  6.13  7.90  7.05
[11]  7.48  7.58  8.11

$obese
[1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69  9.68  9.19
```

### 1.2.15 *Implicit loops*

The looping constructs of R are described in Section 2.3.1. For the purposes of this book, you can largely ignore their existence. However, there is a group of R functions that it will be useful for you to know about.

A common application of loops is to apply a function to each element of a set of values or vectors and collect the results in a single structure. In R this is abstracted by the functions `lapply` and `sapply`. The former always returns a list (hence the 'l'), whereas the latter tries to simplify (hence the 's') the result to a vector or a matrix if possible. So, to compute the mean of each variable in a data frame of numeric vectors, you can do the following:

```
> lapply(thuesen, mean, na.rm=T)
$blood.glucose
[1] 10.3

$short.velocity
[1] 1.325652

> sapply(thuesen, mean, na.rm=T)
 blood.glucose short.velocity
     10.300000       1.325652
```

Notice how both forms attach meaningful names to the result, which is another good reason to prefer to use these functions rather than explicit loops. The second argument to `lapply/sapply` is the function that should be applied, here `mean`. Any further arguments are passed on to the function; in this case we pass `na.rm=T` to request that missing values be removed (see Section 4.1).

Sometimes you just want to repeat something a number of times but still collect the results as a vector. Obviously, this makes sense only when the repeated computations actually give different results, the common case being simulation studies. This can be done using `sapply`, but there is a simplified version called `replicate`, in which you just have to give a count and the expression to evaluate:

```
> replicate(10,mean(rexp(20)))
 [1] 1.0677019 1.2166898 0.8923216 1.1281207 0.9636017 0.8406877
 [7] 1.3357814 0.8249408 0.9488707 0.5724575
```

A similar function, `apply`, allows you to apply a function to the rows or columns of a matrix (or over indices of a multidimensional array in general) as in

```
> m <- matrix(rnorm(12),4)
> m
           [,1]       [,2]       [,3]
[1,] -2.5710730 0.2524470 -0.16886795
[2,]  0.5509498 1.5430648  0.05359794
[3,]  2.4002722 0.1624704 -1.23407417
[4,]  1.4791103 0.9484525 -0.84670929
> apply(m, 2, min)
[1] -2.5710730  0.1624704 -1.2340742
```

The second argument is the index (or vector of indices) that defines what the function is applied to; in this case we get the columnwise minima.

Also, the function `tapply` allows you to create tables (hence the 't') of the value of a function on subgroups defined by its second argument, which can be a factor or a list of factors. In the latter case a cross-classified table is generated. (The grouping can also be defined by ordinary vectors. They will be converted to factors internally.)

```
> tapply(energy$expend, energy$stature, median)
 lean obese
 7.90  9.69
```

### *1.2.16 Sorting*

It is trivial to sort a vector. Just use the `sort` function. (We use the built-in data set `intake` here; it contains the same data that were used in Section 1.2.9.)

```
> intake$post
 [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335
> sort(intake$post)
 [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900 7335
```

(`intake$pre` could not be used for this example since it is sorted already!)

However, sorting a single vector is not always what is required. Often you need to sort a series of variables according to the values of some *other* variables — blood pressures sorted by sex and age, for instance. For this

purpose, there is a construction that may look somewhat abstract at first but is really very powerful. You first compute an *ordering* of a variable.

```
> order(intake$post)
 [1]  3  1  2  6  4  7  5  8  9 10 11
```

The result is the numbers 1 to 11 (or whatever the length of the vector is), sorted according to the size of the argument to `order` (here `intake$post`). Interpreting the result of `order` is a bit tricky — it should be read as follows: You sort `intake$post` by placing its values in the order no. 3, no. 1, no. 2, no. 6, etc.

The point is that, by indexing with this vector, other variables can be sorted by the same criterion. Note that indexing with a vector containing the numbers from 1 to the number of elements exactly once corresponds to a reordering of the elements.

```
> o <- order(intake$post)
> intake$post[o]
 [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900 7335
> intake$pre[o]
 [1] 5640 5260 5470 6515 6180 6805 6390 7515 7515 8230 8770
```

What has happened here is that `intake$post` has been sorted — just as in `sort(intake$post)` — while `intake$pre` has been sorted by the size of the corresponding `intake$post`.

It is of course also possible to sort the entire data frame `intake`

```
> intake.sorted <- intake[o,]
```

Sorting by several criteria is done simply by having several arguments to `order`; for instance, `order(sex,age)` will give a main division into men and women, and within each sex an ordering by age. The second variable is used when the order cannot be decided from the first variable. Sorting in reverse order can be handled by, for example, changing the sign of the variable.

## 1.3 Exercises

**1.1** How would you check whether two vectors are the same if they may contain missing (`NA`) values? (Use of the `identical` function is considered cheating!)

**1.2** If `x` is a factor with `n` levels and `y` is a length `n` vector, what happens if you compute `y[x]`?

**1.3** Write the logical expression to use to extract girls between 7 and 14 years of age in the `juul` data set.

**1.4** What happens if you change the levels of a factor (with `levels`) and give the same value to two or more levels?

**1.5** On p. 27, `replicate` was used to simulate the distribution of the mean of 20 random numbers from the exponential distribution by repeating the operation 10 times. How would you do the same thing with `sapply`?

# 2

# The R environment

This chapter collects some practical aspects of working with R. It describes issues regarding the structure of the workspace, graphical devices and their parameters, and elementary programming, and includes a fairly extensive, although far from complete, discussion of data entry.

## 2.1 Session management

### *2.1.1 The workspace*

All variables created in R are stored in a common workspace. To see which variables are defined in the workspace, you can use the function `ls` (*ls*t). It should look as follows if you have run all the examples in the preceding chapter:

```
> ls()
 [1] "bmi"           "d"             "exp.lean"
 [4] "exp.obese"     "fpain"         "height"
 [7] "hh"            "intake.post"   "intake.pre"
[10] "intake.sorted" "l"             "m"
[13] "mylist"        "o"             "oops"
[16] "pain"          "sel"           "v"
[19] "weight"        "x"             "xbar"
[22] "y"
```

P. Dalgaard, *Introductory Statistics with R*,
DOI: 10.1007/978-0-387-79054-1_2, 

Remember that you cannot omit the parentheses in `ls()`.

If at some point things begin to look messy, you can delete some of the objects. This is done using `rm` (*rem*ove), so that

```
> rm(height, weight)
```

deletes the variables `height` and `weight`.

The entire workspace can be cleared using `rm(list=ls())` and also via the "Remove all objects" or "Clear Workspace" menu entries in the Windows and Macintosh GUIs. This does not remove variables whose name begins with a dot because they are not listed by `ls()` — you would need `ls(all=T)` for that, but it could be dangerous because such names are used for system purposes.

If you are acquainted with the Unix operating system, for which the S language, which preceded R, was originally written, then you will know that the commands for listing and removing files in Unix are called precisely `ls` and `rm`.

It is possible to save the workspace to a file at any time. If you just write

```
save.image()
```

then it will be saved to a file called `.RData` in your working directory. The Windows version also has this on the File menu. When you exit R, you will be asked whether to save the workspace image; if you accept, the same thing will happen. It is also possible to specify an alternative filename (within quotes). You can also save selected objects with `save`. The `.RData` file is loaded by default when R is started in its directory. Other save files can be loaded into your current workspace using `load`.

### 2.1.2 *Textual output*

It is important to note that the workspace consists only of R objects, not of any of the output that you have generated during a session. If you want to save your output, use "Save to File" from the File menu in Windows or use standard cut-and-paste facilities. You can also use ESS (Emacs Speaks Statistics), which works on all platforms. It is a "mode" for the Emacs editor where you can run your entire session in an Emacs buffer. You can get ESS and installation instructions for it from CRAN (see Appendix A).

An alternative way of diverting output to a file is to use the `sink` function. This is largely a relic from the days of the $80 \times 25$ computer terminal, where cut-and-paste techniques were not available, but it can still be use-

ful at times. In particular, it can be used in batch processing. The way it works is as follows:

```
> sink("myfile")
> ls()
```

No output appears! This is because the output goes into the file `myfile` in the current directory. The system will remain in a state where commands are processed, but the output (apparently) goes into the drain until the normal state of affairs is reestablished by

```
> sink()
```

The current working directory can be obtained by `getwd()` and changed by `setwd(mydir)`, where `mydir` is a character string. The initial working directory is system-dependent; for instance, the Windows GUI sets it to the user's home directory, and command line versions use the directory from which you start R.

### 2.1.3 *Scripting*

Beyond a certain level of complexity, you will not want to work with R on a line-by-line basis. For instance, if you have entered an $8 \times 8$ matrix over eight lines and realize that you made a mistake, you will find yourself using the up-arrow key 64 times to reenter it! In such cases, it is better to work with R *scripts*, collections of lines of R code stored either in a file or in computer memory somehow.

One option is to use the `source` function, which is sort of the opposite of `sink`. It takes the input (i.e., the commands from a file) and runs them. Notice, though, that the entire file is syntax-checked before anything is executed. It is often useful to set `echo=T` in the call so that commands are printed along with the output.

Another option is more interactive in nature. You can work with a script editor window, which allows you to submit one or more lines of the script to a running R, which will then behave as if the same lines had been entered at the prompt. The Windows and Macintosh versions of R have simple scripting windows built-in, and a number of text editors also have features for sending commands to R; popular choices on Windows include TINN-R and WinEdt. This is also available as part of ESS (see the preceding section).

The history of commands entered in a session can be saved and reloaded using the `savehistory` and `loadhistory` commands, which are also mapped to menu entries in Windows. Saved histories can be useful as a

starting point for writing scripts; notice also that the `history()` function will show the last commands entered at the console (up to a maximum of 25 lines by default).

### 2.1.4 *Getting help*

R can do a lot more than what a typical beginner can be expected to need or even understand. This book is written so that most of the code you are likely to need in relation to the statistical procedures is described in the text, and the compendium in Appendix C is designed to provide a basic overview. However, it is obviously not possible to cover everything.

R also comes with extensive online help in text form as well as in the form of a series of HTML files that can be read using a Web browser such as Netscape or Internet Explorer. The help pages can be accessed via "help" in the menu bar on Windows and by entering `help.start()` on any platform. You will find that the pages are of a technical nature. Precision and conciseness here take precedence over readability and pedagogy (something one learns to appreciate after exposure to the opposite).

From the command line, you can always enter `help(aggregate)` to get help on the `aggregate` function or use the prefix form `?aggregate`. If the HTML viewer is running, then the help page is shown there. Otherwise it is shown as text either through a pager to the terminal window or in a separate window.

Notice that the HTML version of the help system features a very useful "Search Engine and Keywords" and that the `apropos` function allows you to get a list of command names that contain a given pattern. The function `help.search` is similar but uses fuzzy matching and searches deeper into the help pages, so that it will be able to locate, for example, Kendall's correlation coefficient in `cor.test` if you use `help.search("kendal")`.

Also available with the R distributions is a set of documents in various formats. Of particular interest is "An Introduction to R", originally based on a set of notes for S-PLUS by Bill Venables and David Smith and modified for R by various people. It contains an introduction to the R language and environment in a rather more language-centric fashion than this book. On the Windows platform, you can choose to install PDF documents as part of the installation procedure so that — provided the Adobe Acrobat Reader program is also installed — it can be accessed via the Help menu. An HTML version (without pictures) can be accessed via the browser interface on all platforms.

### 2.1.5 Packages

An R installation contains one or more libraries of packages. Some of these packages are part of the basic installation. Others can be downloaded from CRAN (see Appendix A), which currently hosts over 1000 packages for various purposes. You can even create your own packages.

A library is generally just a folder on your disk. A system library is created when R is installed. In some installations, users may be prohibited from modifying the system library. It is possible to set up private user libraries; see `help(".Library")` for details.

A package can contain functions written in the R language, dynamically loaded libraries of compiled code (written in C or Fortran mostly), and data sets. It generally implements functionality that most users will probably not need to have loaded all the time. A package is loaded into R using the `library` command, so to load the `survival` package you should enter

```
> library(survival)
```

The loaded packages are not considered part of the user workspace. If you terminate your R session and start a new session with the saved workspace, then you will have to load the packages again. For the same reason, it is rarely necessary to remove a package that you have loaded, but it can be done if desired with

```
> detach("package:survival")
```

(see also Section 2.1.7).

### 2.1.6 Built-in data

Many packages, both inside and outside the standard R distribution, come with built-in data sets. Such data sets can be rather large, so it is not a good idea to keep them all in computer memory at all times. A mechanism for on-demand loading is required. In many packages, this works via a mechanism called *lazy loading*, which allows the system to "pretend" that the data are in memory, but in fact they are not loaded until they are referenced for the first time.

With this mechanism, data are "just there". For example, if you type "thuesen", the data frame of that name is displayed. Some packages still require explicit calls to the `data` function. Most often, this loads a data frame with the name that its argument specifies; `data(thuesen)` will, for instance, load the `thuesen` data frame.

What `data` does is to go through the data directories associated with each package (see Section 2.1.5) and look for files whose basename matches the given name. Depending on the file extension, several things can then happen. Files with a `.tab` extension are read using `read.table` (Section 2.4), whereas files with a `.R` extension are executed as source files (and could, in general, do anything!), to give two common examples.

If there is a subdirectory of the current directory called `data`, then it is searched as well. This can be quite a handy way of organizing your personal projects.

### 2.1.7 *attach and detach*

The notation for accessing variables in data frames gets rather heavy if you repeatedly have to write longish commands like

```
plot(thuesen$blood.glucose,thuesen$short.velocity)
```

Fortunately, you can make R look for objects among the variables in a given data frame, for example `thuesen`. You write

```
> attach(thuesen)
```

and then `thuesen`'s data are available without the clumsy `$`-notation:

```
> blood.glucose
 [1] 15.3 10.8  8.1 19.5  7.2  5.3  9.3 11.1  7.5 12.2  6.7  5.2
[13] 19.0 15.1  6.7  8.6  4.2 10.3 12.5 16.1 13.3  4.9  8.8  9.5
```

What happens is that the data frame `thuesen` is placed in the system's *search path*. You can view the search path with `search`:

```
> search()
 [1] ".GlobalEnv"        "thuesen"           "package:ISwR"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

Notice that `thuesen` is placed as no. 2 in the search path. `.GlobalEnv` is the workspace and `package:base` is the system library where all standard functions are defined. `Autoloads` is not described here. `package:stats` and onwards contains the basic statistical routines such as the Wilcoxon test, and the other packages similarly contain various functions and data sets. (The package system is modular, and you can run R with a minimal set of packages for specific uses.) Finally, `package:ISwR` contains the data sets used for this book.

There may be several objects of the same name in different parts of the search path. In that case, R chooses the first one (that is, it searches first in `.GlobalEnv`, then in `thuesen`, and so forth). For this reason, you need to be a little careful with "loose" objects that are defined in the workspace outside a data frame since they will be used before any vectors and factors of the same name in an attached data frame. For the same reason, it is not a good idea to give a data frame the same name as one of the variables inside it. Note also that changing a data frame after attaching it will not affect the variables available since `attach` involves a (virtual) copy operation of the data frame.

It is not possible to attach data frames in front of `.GlobalEnv` or following `package:base`. However, it is possible to attach more than one data frame. New data frames are inserted into position 2 by default, and everything except `.GlobalEnv` moves one step to the right. It is, however, possible to specify that a data frame should be searched before `.GlobalEnv` by using constructions of the form

```
with(thuesen, plot(blood.glucose, short.velocity))
```

In some contexts, R uses a slightly different method when looking for objects. If looking for a variable of a specific type (usually a function), R will skip those of other types. This is what saves you from the worst consequences of accidentally naming a variable (say) `c`, even though there is a system function of the same name.

You can remove a data frame from the search path with `detach`. If no arguments are given, the data frame in position 2 is removed, which is generally what is desired. `.GlobalEnv` and `package:base` cannot be `detach`'ed.

```
> detach()
> search()
 [1] ".GlobalEnv"        "package:ISwR"      "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

### 2.1.8 `subset`, `transform`, and `within`

You can attach a data frame to avoid the cumbersome indexing of every variable inside of it. However, this is less helpful for selecting subsets of data and for creating new data frames with transformed variables. A couple of functions exist to make these operations easier. They are used as follows:

```
> thue2 <- subset(thuesen,blood.glucose<7)
> thue2
   blood.glucose short.velocity
6            5.3           1.49
11           6.7           1.25
12           5.2           1.19
15           6.7           1.52
17           4.2           1.12
22           4.9           1.03
> thue3 <- transform(thuesen,log.gluc=log(blood.glucose))
> thue3
   blood.glucose short.velocity log.gluc
1           15.3           1.76 2.727853
2           10.8           1.34 2.379546
3            8.1           1.27 2.091864
4           19.5           1.47 2.970414
5            7.2           1.27 1.974081
...
22           4.9           1.03 1.589235
23           8.8           1.12 2.174752
24           9.5           1.70 2.251292
```

Notice that the variables used in the expressions for new variables or for subsetting are evaluated with variables taken from the data frame.

`subset` also works on single vectors. This is nearly the same as indexing with a logical vector (such as `short.velocity[blood.glucose<7]`), except that observations with missing values in the selection criterion are excluded.

`subset` also has a `select` argument which can be used to extract variables from the data frame. We shall return to this in Section 10.3.1.

The `transform` function has a couple of drawbacks, the most serious of which is probably that it does not allow chained calculations where some of the new variables depend on the others. The = signs in the syntax are not assignments, but indicate names, which are assigned to the computed vectors in the last step.

An alternative to `transform` is the `within` function, which can be used like this:

```
> thue4 <- within(thuesen,{
+    log.gluc <- log(blood.glucose)
+    m <- mean(log.gluc)
+    centered.log.gluc <- log.gluc - m
+    rm(m)
+ })
> thue4
   blood.glucose short.velocity centered.log.gluc log.gluc
1           15.3           1.76       0.481879807 2.727853
2           10.8           1.34       0.133573113 2.379546
```

```
3            8.1          1.27      -0.154108960 2.091864
4           19.5          1.47       0.724441444 2.970414
5            7.2          1.27      -0.271891996 1.974081
...
22           4.9          1.03      -0.656737817 1.589235
23           8.8          1.12      -0.071221300 2.174752
24           9.5          1.70       0.005318777 2.251292
```

Notice that the second argument is an arbitrary expression (here a *compound* expression, see p. 45). The function is similar to `with`, but instead of just returning the computed value, it collects all new and modified variables into a modified data frame, which is then returned. As shown, variables containing intermediate results can be discarded with `rm`. (It is particularly important to do this if the contents are incompatible with the data frame.)

## 2.2 The graphics subsystem

In Section 1.1.5, we saw how to generate a simple plot and superimpose a curve on it. It is quite common in statistical graphics for you to want to create a plot that is slightly different from the default: Sometimes you will want to add annotation, sometimes you want the axes to be different — labels instead of numbers, irregular placement of tick marks, etc. All these things can be obtained in R. The methods for doing them may feel slightly unusual at first, but offers a very flexible and powerful approach.

In this section, we look deeper into the structure of a typical plot and give some indication of how you can work with plots to achieve your desired results. Beware, though, that this is a large and complex area and it is not within the scope of this book to cover it completely. In fact, we completely ignore important newer tools in the `grid` and `lattice` packages.

### *2.2.1 Plot layout*

In the graphics model that R uses, there is (for a single plot) a figure region containing a central plotting region surrounded by margins. Coordinates inside the plotting region are specified in data units (the kind generally used to label the axes). Coordinates in the margins are specified in *lines of text* as you move in a direction perpendicular to a side of the plotting region but in data units as you move along the side. This is useful since you generally want to put text in the margins of a plot.

A standard $x$–$y$ plot has an $x$ and a $y$ title label generated from the expressions being plotted. You may, however, override these labels and also

add two further titles, a main title above the plot and a subtitle at the very bottom, in the plot call.

```
> x <- runif(50,0,2)
> y <- runif(50,0,2)
> plot(x, y, main="Main title", sub="subtitle",
+      xlab="x-label", ylab="y-label")
```

Inside the plotting region, you can place points and lines that are either specified in the `plot` call or added later with `points` and `lines`. You can also place a text with

```
> text(0.6,0.6,"text at (0.6,0.6)")
> abline(h=.6,v=.6)
```

Here, the `abline` call is just to show how the text is centered on the point $(0.6, 0.6)$. (Normally, `abline` plots the line $y = a + bx$ when given `a` and `b` as arguments, but it can also be used to draw horizontal and vertical lines as shown.)

The margin coordinates are used by the `mtext` function. They can be demonstrated as follows:

```
> for (side in 1:4) mtext(-1:4,side=side,at=.7,line=-1:4)
> mtext(paste("side",1:4), side=1:4, line=-1,font=2)
```

The `for` loop (see Section 2.3.1) places the numbers $-1$ to 4 on corresponding lines in each of the four margins at an off-center position of 0.7 measured in user coordinates. The subsequent call places a label on each side, giving the side number. The argument `font=2` means that a boldface font is used. Notice in Figure 2.1 that not all the margins are wide enough to hold all the numbers and that it is possible to use negative line numbers to place text within the plotting region.

### 2.2.2 *Building a plot from pieces*

High-level plots are composed of elements, each of which can also be drawn separately. The separate drawing commands often allow finer control of the element, so a standard strategy to achieve a given effect is first to draw the plot without that element and add the element subsequently. As an extreme case, the following command will plot absolutely nothing:

```
> plot(x, y, type="n", xlab="", ylab="", axes=F)
```

Here `type="n"` causes the points not to be drawn. `axes=F` suppresses the axes and the box around the plot, and the $x$ and $y$ title labels are set to empty strings.

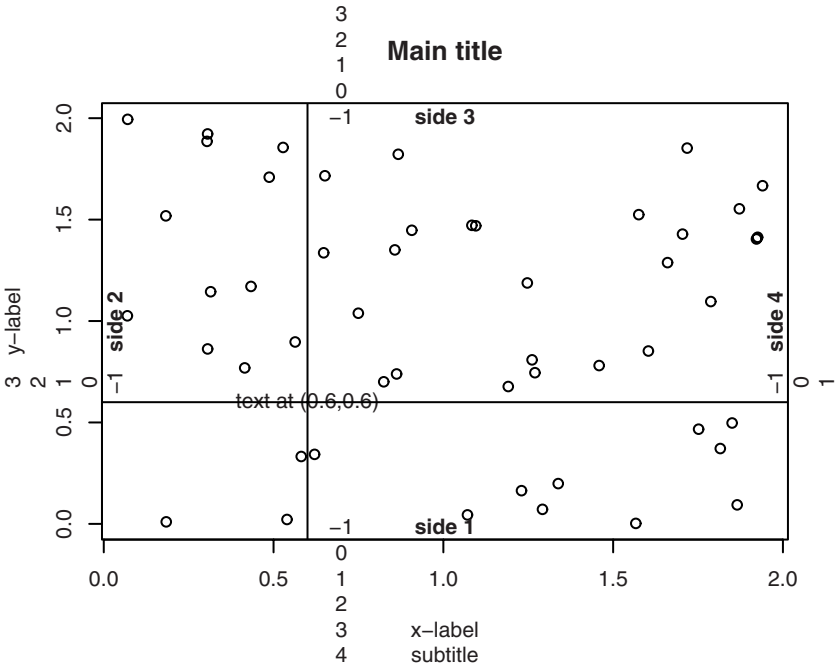


Figure 2.1. The layout of a standard plot.

However, the fact that nothing is plotted does not mean that nothing happened. The command sets up the plotting region and coordinate systems just as if it had actually plotted the data. To add the plot elements, evaluate the following:

```
> points(x,y)
> axis(1)
> axis(2,at=seq(0.2,1.8,0.2))
> box()
> title(main="Main title", sub="subtitle",
+     xlab="x-label", ylab="y-label")
```

Notice how the second `axis` call specifies an alternative set of tick marks (and labels). This is a common technique used to create special axes on a plot and might also be used to create nonequidistant axes as well as axes with nonnumeric labelling.

Plotting with `type="n"` is sometimes a useful technique because it has the side effect of dimensioning the plot area. For instance, to create a plot with different colours for different groups, you could first plot all data with `type="n"`, ensuring that the plot region is large enough, and then

add the points for each group using `points`. (Passing a vector argument for `col` is more expedient in this particular case.)

### 2.2.3 *Using `par`*

The `par` function allows incredibly fine control over the details of a plot, although it can be quite confusing to the beginner (and even to experienced users at times). The best strategy for learning it may well be simply to try and pick up a few useful tricks at a time and once in a while try to solve a particular problem by poring over the help page.

Some of the parameters, but not all, can also be set via arguments to plotting functions, which also have some arguments that cannot be set by `par`. When a parameter can be set by both methods, the difference is generally that if something is set via `par`, then it stays set subsequently.

The `par` settings allow you to control line width and type, character size and font, colour, style of axis calculation, size of the plot and figure regions, clipping, etc. It is possible to divide a figure into several subfigures by using the `mfrow` and `mfcol` parameters.

For instance, the default margin sizes are just over 5, 4, 4, and 2 lines. You might set `par(mar=c(4,4,2,2)+0.1)` before plotting. This shaves one line off the bottom margin and two lines off the top margin of the plot, which will reduce the amount of unused whitespace when there is no main title or subtitle. If you look carefully, you will in fact notice that Figure 2.1 has a somewhat smaller plotting region than the other plots in this book. This is because the other plots have been made with reduced margins for typesetting reasons.

However, it is quite pointless to describe the graphics parameters completely at this point. Instead, we return to them as they are used for specific plots.

### 2.2.4 *Combining plots*

Some special considerations arise when you wish to put several elements together in the same plot. Consider overlaying a histogram with a normal density (see Sections 4.2 and 4.4.1 for information on histograms and Section 3.5.1 for density). The following is close, but only nearly good enough (figure not shown).

```
> x <- rnorm(100)
> hist(x,freq=F)
> curve(dnorm(x),add=T)
```

The `freq=F` argument to `hist` ensures that the histogram is in terms of densities rather than absolute counts. The `curve` function graphs an expression (in terms of `x`) and its `add=T` allows it to overplot an existing plot. So things are generally set up correctly, but sometimes the top of the density function gets chopped off. The reason is of course that the height of the normal density played no role in the setting of the *y*-axis for the histogram. It will not help to reverse the order and draw the curve first and add the histogram because then the highest bars might get clipped.

The solution is first to get hold of the magnitude of the *y* values for both plot elements and make the plot big enough to hold both (Figure 2.2):

```
> h <- hist(x, plot=F)
> ylim <- range(0, h$density, dnorm(0))
> hist(x, freq=F, ylim=ylim)
> curve(dnorm(x), add=T)
```

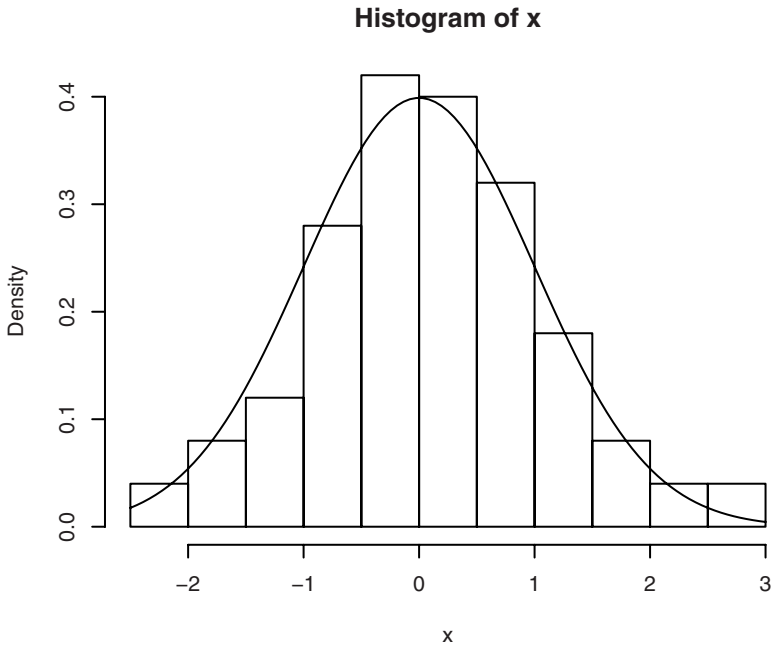


Figure 2.2. Histogram with normal density overlaid.

When called with `plot=F`, `hist` will not plot anything, but it will return a structure containing the bar heights on the density scale. This and the fact that the maximum of `dnorm(x)` is `dnorm(0)` allows us to calculate a range covering both the bars and the normal density. The zero in

the `range` call ensures that the bottom of the bars will be in range, too. The range of $y$ values is then passed to the `hist` function via the `ylim` argument.

## 2.3 R programming

It is possible to write your own R functions. In fact, this is a major aspect and attraction of working with the system in the long run. This book largely avoids the issue in favour of covering a larger set of basic statistical procedures that can be executed from the command line. However, to give you a feel for what can be done, consider the following function, which wraps the code from the example of Section 2.2.4 so that you can just say `hist.with.normal(rnorm(200))`. It has been slightly extended so that it now uses the empirical mean and standard deviation of the data instead of just 0 and 1.

```
> hist.with.normal <- function(x, xlab=deparse(substitute(x)),...)
+ {
+     h <- hist(x, plot=F, ...)
+     s <- sd(x)
+     m <- mean(x)
+     ylim <- range(0,h$density,dnorm(0,sd=s))
+     hist(x, freq=F, ylim=ylim, xlab=xlab, ...)
+     curve(dnorm(x,m,s), add=T)
+ }
```

Notice the use of a default argument for `xlab`. If `xlab` is not specified, then it is obtained from this expression, which evaluates to a character form of the expression given for `x`; that is, if you pass `rnorm(100)` for `x`, then the $x$ label becomes "rnorm(100)". Notice also the use of a `...` argument, which collects any additional arguments and passes them on to `hist` in the two calls.

You can learn more about programming in R by studying the built-in functions, starting with simple ones like `log10` or `weighted.mean`.

### *2.3.1 Flow control*

Until now, we have seen components of the R language that cause evaluation of single expressions. However, R is a true programming language that allows conditional execution and looping constructs as well. Consider, for instance, the following code. (The code implements a version of Newton's method for calculating the square root of `y`.)

```
> y <- 12345
> x <- y/2
> while (abs(x*x-y) > 1e-10) x <- (x + y/x)/2
> x
[1] 111.1081
> x^2
[1] 12345
```

Notice the `while(condition) expression` construction, which says that the expression should be evaluated as long as the condition is `TRUE`. The test occurs at the top of the loop, so the expression might never be evaluated.

A variation of the same algorithm with the test at the bottom of the loop can be written with a `repeat` construction:

```
> x <- y/2
> repeat{
+     x <- (x + y/x)/2
+     if (abs(x*x-y) < 1e-10) break
+ }
> x
[1] 111.1081
```

This also illustrates three other flow control structures: (a) a *compound expression*, several expressions held together between curly braces; (b) an `if` construction for conditional execution; and (c) a `break` expression, which causes the enclosing loop to exit.

Incidentally, the loop could allow for `y` being a vector simply by changing the termination condition to

```
if (all(abs(x*x - y) < 1e-10)) break
```

This would iterate excessively for some elements, but the vectorized arithmetic would likely more than make up for that.

However, the most frequently used looping construct is `for`, which loops over a fixed set of values as in the following example, which plots a set of power curves on the unit interval.

```
> x <- seq(0, 1,.05)
> plot(x, x, ylab="y", type="l")
> for ( j in 2:8 ) lines(x, x^j)
```

Notice the *loop variable* `j`, which in turn takes the values of the given sequence when used in the `lines` call.

### 2.3.2 Classes and generic functions

Object-oriented programming is about creating coherent systems of data and methods that work upon them. One purpose is to simplify programs by accommodating the fact that you will have conceptually similar methods for different types of data, even though the implementations will have to be different. A prototype example is the print method: It makes sense to print many kinds of data objects, but the print layout will depend on what the data object is. You will generally have a *class* of data objects and a *print method* for that class. There are several object-oriented languages implementing these ideas in different ways.

Most of the basic parts of R use the same object system as S version 3. An alternative object system similar to that of S version 4 has been developed in recent years. The new system has several advantages over the old one, but we shall restrict attention to the latter. The S3 object system is a simple system in which an object has a `class` attribute, which is simply a character vector. One example of this is that all the return values of the classical tests such as `t.test` have class `"htest"`, indicating that they are the result of a hypothesis test. When these objects are printed, it is done by `print.htest`, which creates the nice layout (see Chapter 5 for examples). However, from a programmatic viewpoint, these objects are just lists, and you can, for instance, extract the *p*-value by writing

```
> t.test(bmi, mu=22.5)$p.value
[1] 0.7442183
```

The function `print` is a *generic function*, one that acts differently depending on its argument. These generally look like this:

```
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
```

What `UseMethod("print")` means is that R should pass control to a function named according to the object class (`print.htest` for objects of class `"htest"`, etc.) or, if this is not found, to `print.default`. To see all the methods available for `print`, type `methods(print)` (there are 138 of them in R 2.6.2, so the output is not shown here).

## 2.4 Data entry

Data sets do not have to be very large before it becomes impractical to type them in with `c(...)`. Most of the examples in this book use data sets in-

cluded in the ISwR package, made available to you by library(ISwR). However, as soon as you wish to apply the methods to your own data, you will have to deal with data file formats and the specification thereof.

In this section we discuss how to read data files and how to use the data editor module in R. The text has some bias toward Windows systems, mainly because of some special issues that need to be mentioned for that platform.

### 2.4.1 *Reading from a text file*

The most convenient way of reading data into R is via the function called read.table. It requires that data be in "ASCII format"; that is, a "flat file" as created with Windows' NotePad or any plain-text editor. The result of read.table is a data frame, and it expects to find data in a corresponding layout where each line in the file contains all data from one subject (or rat or ...) in a specific order, separated by blanks or, optionally, some other separator. The first line of the file can contain a header giving the names of the variables, a practice that is highly recommended.

Table 11.6 in Altman (1991) contains an example on ventricular circumferential shortening velocity versus fasting blood glucose by Thuesen et al. We used those data to illustrate subsetting and use them again in the chapter on correlation and regression. They are among the built-in data sets in the ISwR package and available as the data frame thuesen, but the point here is to show how to read them from a plain-text file.

Assume that the data are contained in the file thuesen.txt, which looks as follows:

```
blood.glucose   short.velocity
15.3            1.76
10.8            1.34
8.1             1.27
19.5            1.47
7.2             1.27
5.3             1.49
9.3             1.31
11.1            1.09
7.5             1.18
12.2            1.22
6.7             1.25
5.2             1.19
19.0            1.95
15.1            1.28
6.7             1.52
8.6             NA
4.2             1.12
```

```
10.3            1.37
12.5            1.19
16.1            1.05
13.3            1.32
4.9             1.03
8.8             1.12
9.5             1.70
```

To enter the data into the file, you could start up Windows' NotePad or any other plain-text editor, such as those discussed in Section 2.1.3. Unix/Linux users should just use a standard editor, such as `emacs` or `vi`. If you must, you can even use a word processing program with a little care.

You should simply type in the data as shown. Notice that the columns are separated by an arbitrary number of blanks and that `NA` represents a missing value.

At the end, you should save the data to a text file. Notice that word processors require special actions in order to save as text. Their normal save format is difficult to read from other programs.

Assuming further that the file is in the `ISwR` folder on the `N:` drive, the data can be read using

```
> thuesen2 <- read.table("N:/ISwR/thuesen.txt",header=T)
```

Notice `header=T` specifying that the first line is a header containing the names of variables contained in the file. Also note that you use forward slashes (`/`), not backslashes (`\`), in the filename, even on a Windows system.

The reason for avoiding backslashes in Windows filenames is that the symbol is used as an escape character (see Section 1.2.4) and therefore needs to be doubled. You could have used `N:\\ISwR\\thuesen.txt`.

The result is a data frame, which is assigned to the variable `thuesen2` and looks as follows:

```
> thuesen2
   blood.glucose short.velocity
1           15.3           1.76
2           10.8           1.34
3            8.1           1.27
4           19.5           1.47
5            7.2           1.27
6            5.3           1.49
7            9.3           1.31
8           11.1           1.09
9            7.5           1.18
10          12.2           1.22
```

```
11           6.7           1.25
12           5.2           1.19
13          19.0           1.95
14          15.1           1.28
15           6.7           1.52
16           8.6             NA
17           4.2           1.12
18          10.3           1.37
19          12.5           1.19
20          16.1           1.05
21          13.3           1.32
22           4.9           1.03
23           8.8           1.12
24           9.5           1.70
```

To read in factor variables (see Section 1.2.8), the easiest way may be to encode them using a textual representation. The `read.table` function autodetects whether a vector is text or numeric and converts it to a factor in the former case (but makes no attempt to recognize numerically coded factors). For instance, the `secretin` built-in data set is read from a file that begins like this:

```
   gluc person time repl time20plus time.comb
1    92      A  pre    a        pre       pre
2    93      A  pre    b        pre       pre
3    84      A   20    a        20+        20
4    88      A   20    b        20+        20
5    88      A   30    a        20+       30+
6    90      A   30    b        20+       30+
7    86      A   60    a        20+       30+
8    89      A   60    b        20+       30+
9    87      A   90    a        20+       30+
10   90      A   90    b        20+       30+
11   85      B  pre    a        pre       pre
12   85      B  pre    b        pre       pre
13   74      B   20    a        20+        20
....
```

This file can be read directly by `read.table` with no arguments other than the filename. It will recognize the case where the first line is one item shorter than the rest and will interpret that layout to imply that the first line contains a header and the first value on all subsequent lines is a row label — that is, exactly the layout generated when printing a data frame.

Reading factors like this may be convenient, but there is a drawback: The level order is alphabetic, so for instance

```
> levels(secretin$time)
[1] "20"  "30"  "60"  "90"  "pre"
```

If this is not what you want, then you may have to manipulate the factor levels; see Section 10.1.2.

A technical note: The files referenced above are contained in the `ISwR` package in the subdirectory (folder) `rawdata`. Exactly where the file is located on your system will depend on where the `ISwR` package was installed. You can find this out as follows:

```
> system.file("rawdata", "thuesen.txt", package="ISwR")
[1] "/home/pd/Rlibrary/ISwR/rawdata/thuesen.txt"
```

### 2.4.2 *Further details on `read.table`*

The `read.table` function is a very flexible tool that is controlled by many options. We shall not attempt a full description here but just give some indication of what it can do.

#### *File format details*

We have already seen the use of `header=T`. A couple of other options control the detailed format of the input file:

**Field separator.** This can be specified using `sep`. Notice that when this is used, as opposed to the default use of whitespace, there must be exactly one separator between data fields. Two consecutive separators will imply that there is a missing value in between. Conversely, it is necessary to use specific codes to represent missing values in the default format and also to use some form of quoting for strings that contain embedded spaces.

**NA strings.** You can specify which strings represent missing values via `na.strings`. There can be several different strings, although not different strings for different columns. For print files from the SAS program, you would use `na.strings="."`.

**Quotes and comments.** By default, R-style quotes can be used to delimit character strings, and parts of files following the comment character `#` are ignored. These features can be modified or removed via the `quote` and `comment.char` arguments.

**Unequal field count.** It is normally considered an error if not all lines contain the same number of values (the first line can be one item short, as described above for the `secretin` data). The `fill` and `flush` arguments can be used in case lines vary in length.

### *Delimited file types*

Applications such as spreadsheets and databases produce text files in formats that require multiple options to be adjusted. For such purposes, there exist "precooked" variants of `read.table`. Two of these are intended to handle CSV files and are called `read.csv` and `read.csv2`. The former assumes that fields are separated by a comma, and the latter assumes that they are separated by semicolons but use a comma as the decimal point (this format is often generated in European locales). Both formats have `header=T` as the default. Further variants are `read.delim` and `read.delim2` for reading delimited files (by default, Tab-delimited files).

### *Conversion of input*

It can be desirable to override the default conversion mechanisms in `read.table`. By default, nonnumeric input is converted to factors, but it does not always make sense. For instance, names and addresses typically should not be converted. This can be modified either for all columns using `stringsAsFactors` or on a per-item basis using `as.is`.

Automatic conversion is often convenient, but it is inefficient in terms of computer time and storage; in order to read a numeric column, `read.table` first reads it as character data, checks whether all elements can be converted to numeric, and only then performs the conversion. The `colClasses` argument allows you to bypass the mechanism by explicitly specifying which columns are of which class (the standard classes `"character"`, `"numeric"`, etc., get special treatment). You can also skip unwanted columns by specifying `"NULL"` as the class.

## *2.4.3 The data editor*

R lets you edit data frames using a spreadsheet-like interface. The interface is a bit rough but quite useful for small data sets.

To edit a data frame, you can use the `edit` function:

```
> aq <- edit(airquality)
```

This brings up a spreadsheet-like editor with a column for each variable in the data frame. The `airquality` data set is built into R; see `help(airquality)` for its contents. Inside the editor, you can move around with the mouse or the cursor keys and edit the current cell by typing in data. The type of variable can be switched between real (numeric) and character (factor) by clicking on the column header, and the name of

the variable can be changed similarly. Note that there is (as of R 2.6.2) no way to delete rows and columns and that new data can be entered only at the end.

When you close the data editor, the edited data frame is assigned to `aq`. The original `airquality` is left intact. Alternatively, if you do not mind overwriting the original data frame, you can use

```
> fix(aq)
```

This is equivalent to `aq <- edit(aq)`.

To enter data into a blank data frame, use

```
> dd <- data.frame()
> fix(dd)
```

An alternative would be `dd <- edit(data.frame())`, which works fine except that beginners tend to reexecute the command when they need to edit `dd`, which of course destroys all data. It is necessary in either case to start with an empty data frame since by default `edit` expects you to want to edit a user-defined function and would bring up a text editor if you started it as `edit()`.

### *2.4.4 Interfacing to other programs*

Sometimes you will want to move data between R and other statistical packages or spreadsheets. A simple fallback approach is to request that the package in question export data as a text file of some sort and use `read.table`, `read.csv`, `read.csv2`, `read.delim`, or `read.delim2`, as previously described.

The `foreign` package is one of the packages labelled "recommended" and should therefore be available with binary distributions of R. It contains routines to read files in several formats, including those from SPSS (`.sav` format), SAS (export libraries), Epi-Info (`.rec`), Stata, Systat, Minitab, and some S-PLUS version 3 dump files.

Unix/Linux users sometimes find themselves with data sets written on Windows machines. The `foreign` package will work there as well for those formats that it supports. Notice that ordinary SAS data sets are not among the supported formats. These have to be converted to export libraries on the originating system. Data that have been entered into Microsoft Excel spreadsheets are most conveniently extracted using a compatible application such as OOo (OpenOffice.org).

An expedient technique is to read from the system clipboard. Say, highlight a rectangular region in a spreadsheet, press Ctrl-C (if on Windows), and inside R use

```
read.table("clipboard", header=T)
```

This does require a little caution, though. It may result in loss of accuracy since you only transfer the data as they appear on the screen. This is mostly a concern if you have data to many significant digits.

For data stored in databases, there exist a number of interface packages on CRAN. Of particular interest on Windows and with some Unix databases is the RODBC package because you can set up ODBC ("Open Database Connectivity") connections to data stored by common applications, including Excel and Access. Some Unix databases (e.g., PostgreSQL) also allow ODBC connections.

For up-to-date information on these matters, consult the "R Data Import/Export" manual that comes with the system.

## 2.5 Exercises

**2.1** Describe how to insert a value between two elements of a vector at a given position by using the `append` function (use the help system to find out). Without `append`, how would you do it?

**2.2** Write the built-in data set `thuesen` to a Tab-separated text file with `write.table`. View it with a text editor (depending on your system). Change the `NA` value to `.` (period), and read the changed file back into R with a suitable command. Also try importing the data into other applications of your choice and exporting them to a new file after editing. You may have to remove row names to make this work.