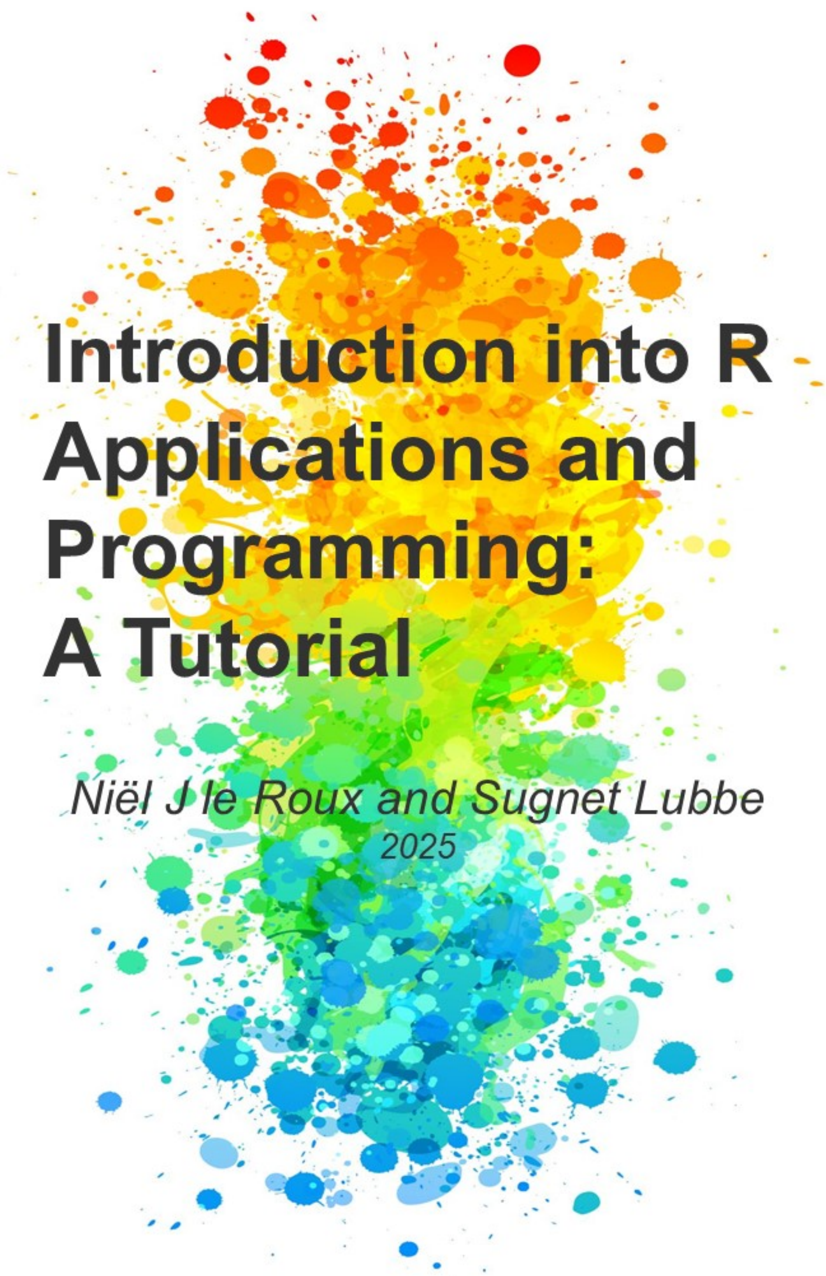
Introduction into R Applications and Programming: A Tutorial

Niël J le Roux and Sugnet Lubbe

2025

Contents

Preface


Introduction into R
Applications and
Programming:
A Tutorial
Niël J le Roux and Sugnet Lubbe
2025

This book is an updated version of (?).

Preface to A Step-by-Step R Tutorial (2013)

The R system is an open-source software project for analyzing data and constructing graphics. It provides a general computer language for performing tasks like organizing data, statistical analyses, simulation studies, model fitting, building of complex graphics and many more.

Central to the R system is the high-level R computer language. Its roots date back to the birth of the computer language S on May 5, 1976 at Bell Labs, Murray Hill, New Jersey (?). In its early days S underwent several revisions and extensions mainly for implementation on the UNIX operating system. Eventually an enhanced version of S was licensed under the name S-PLUS and became available for the Windows operating system under the name S-PLUS for Windows. The earlier versions of R adhered to the principles of functional programming and with the release of version S3 in the middle eighties its building blocks were dynamically generated, self-describing objects. The publication The New S Language (?) provides a detailed description of S3. The next major development of S was the release of Statistical Models in S (?) which involved the merging of the functional style of S with object-oriented programming concepts of classes and methods. However, S3 has only limited formal support for classes and methods. The introduction of S4 objects (?) introduced a new class and method system but retains S3 compatibility. In the meantime several versions of S-PLUS based upon S3 at first and later on S4 were released in the commercial market.

The R language itself was introduced in a paper published by Ross Ihaka and Robert Gentleman of Auckland, New Zealand in 1996 (?). This proposal was to a large extent compatible with S but included features from the Lisp/Scheme family of languages. An important aspect of R was its availability as an open-source system.

Both R and S-PLUS can be considered to be clones of the same underlying S. That means that if you are able to program in the one you can quite easily program in the other but be warned: there are also fundamental differences between the two systems.

In the first two decades of the twenty-first century interest in R has exceeded all possible expectations. Apart from a well-maintained core system with new releases every few months there are currently literally thousands of researchers contributing add-on packages on cutting-edge developments in statistics and data analysis.

This book is a tutorial with a twofold aim; learning the basics of the R system and how to program efficiently in R. It is the result of an introductory course in S-PLUS taught at the University of Stellenbosch since 1995. The initial course was based on the book An Introduction to S and S-Plus (?). Since 2002 increasingly more emphasis was put on R to such an extent that it is currently exclusively devoted to R. This change necessitated the preparation of class notes for a ten-day (eight hours a day) tutorial course in R. The result is A Step-by-Step R Tutorial: An introduction into R applications and programming.

Preface to A Step-by-Step R Tutorial (2021)

Since the first publication of A Step-by-Step R Tutorial: An introduction into R applications and programming the R system has experienced a dramatic evolutionary process. This edition still maintains the twofold aim of the first edition while adapting its contents to the needs of the modernization that has been happening within the R system itself. Deprecated or outdated material has been omitted and new developments included. What follows is a brief description of these changes.

Chapter 1 contains a new section explaining how to use R Markdown for creating PDF and HTML documents from R output. Chapters 2, 3, 4 and 5 see only minor changes. In Chapter 6 changes are made in the data sets used as well as in some exercises being borrowed from later chapters in the first edition. In Chapter 7, 'Writing R Functions', a notable reference is made to the `Rcpp` package for the inclusion of C++ code into R. This package allows compiled code to be included considerably easier and more robust. Vectorized programming and mapping functions are enhanced in Chapter 8 by a discussion of the function `mapply()`. A major addition is a discussion in section 8.14 for writing user-friendly applications using the package `shiny`. This replaces the usage of the function `menu()`. An exercise to create a simple shiny App is also included.

In the first part of Chapter 9, 'Reading data files into R, formatting and printing', methods for reading Microsoft Excel files have been updated; functions like `readRDS()` and `writeRDS()` for transporting R objects are introduced; and the `clipr` package is discussed. A major addition to this chapter is the section devoted to the functionality provided by the `tidyverse` collection of R packages for data manipulation and exploration; `tibbles` are discussed in detail as well as the pipe operator `%>%`, tidy data is illustrated and the data manipulation functions of `dplyr` illustrated in detail.

Chapter 10, 'R graphics: Round II', has been considerably extended by the inclusion of a section on how to specify colours; a rewritten section on quantile plots and inclusion of material previously in Chapter 11. There is now a section on density estimation, which includes a discussion of density histograms and average shifted histograms. In the new section 10.14 the package `ggplot2` is discussed with many examples of its capabilities.

The chapter on 'Modelling in R' (Chapter 11) and the extensive discussion of the Analysis of Variance and Covariance (Chapter 12) in the previous edition have been rewritten completely and consolidated into a new Chapter 11. The final chapter is now Chapter 12, 'Introduction to Optimization'. Apart from a new data set the material is similar to that in Chapter 13 of the previous edition.

Chapter 1

Introducing the R System

1.1 Introduction

This chapter introduces the R system to the new R user. The Windows operating system is emphasized but most of the material covered also applies to other operating systems after allowing for the requirements of the particular operating system in use. Users with some experience with R should quickly glance through this chapter making sure they have mastered all topics covered here before proceeding with the main tutorial starting with Chapter 2.

In the computer age statistics has become inseparable from being able to write computer programs. Therefore, let us start with a reminder of the Fundamental Goal of S:

Conversion of an idea into useful software

The challenge is to pursue this goal keeping in mind the Mission of R (**?**):

... to enable the best and most thorough exploration of data possible

and its Prime Directive (**?**):

... places and obligation on all creators of software to program in such a way that the computations can be understood and trusted.

1.2 Downloading the R system

Website for downloading R.

To download R to your own computer: Navigate to *.../bin/windows/base* and save the file *R-x.y.z.-win.exe* on your computer. Click this file to start the installation procedure and select the defaults unless you have a good reason not to do so. If you select 'Create desktop icon' during the installation phase, an icon similar to the one below should appear on the desktop. Alternatively, you can find R under *All Applications*.

The core R system that is installed includes several *packages*. Apart from these installed packages several thousands of dedicated *contributed packages* are available to be downloaded by users in need of any of them.

1.3 A quick sample R session

Click the R icon created on your desktop to open the *Commands Window* or *Console*. Notice the R prompt `>` waiting for some instruction from the user.

(a) At the R prompt `>` enter `5 - 8`. We will follow the following convention to write instructions:

```
5 - 8
#> [1] -3
```

(b) Repeat (a) but enter only 5 – and see what happens:

```
> 5 -
> +
> +
```

The above `+` is the secondary R prompt. It indicates that an instruction is unfinished. Either respond by completing the instruction or press the Esc key to start all over again from the primary prompt.

(c) Enter

```
xx <- 1:10
```

This instruction creates an R object with name (or label) `xx` containing the vector (1, 2, 3, 4, 5, 6, 7, 8, 10).

(d) Enter

```
yy <- rnorm(n = 20, mean = 50, sd = 15)
```

This instruction creates an R object with name `yy` containing a random sample of 20 values from a normal distribution with a mean of 50 and a standard deviation of 15.

(e) Enter

```
xx
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

The above example shows that when the name of an R object is entered at the prompt, R will respond by displaying the contents of the object.

(f) Obtain a representation of the contents of the object `yy` created in (d).

(g) A program in R is called a *function*. Any function in R is also an R *object* and therefore has a name (or label). It follows from (e) that if the name of a function is entered at the prompt, R will respond by displaying the contents of the function.

How then can an R function be executed i.e. how can an R function be called? Apart from its name an R function has a list of arguments enclosed within parentheses. An R function is called by entering its name followed by a list of arguments enclosed within parentheses. As an example, let us calculate the mean of the object `yy` created above by calling the function `mean`:

```
mean(yy)
#> [1] 50.31071
```

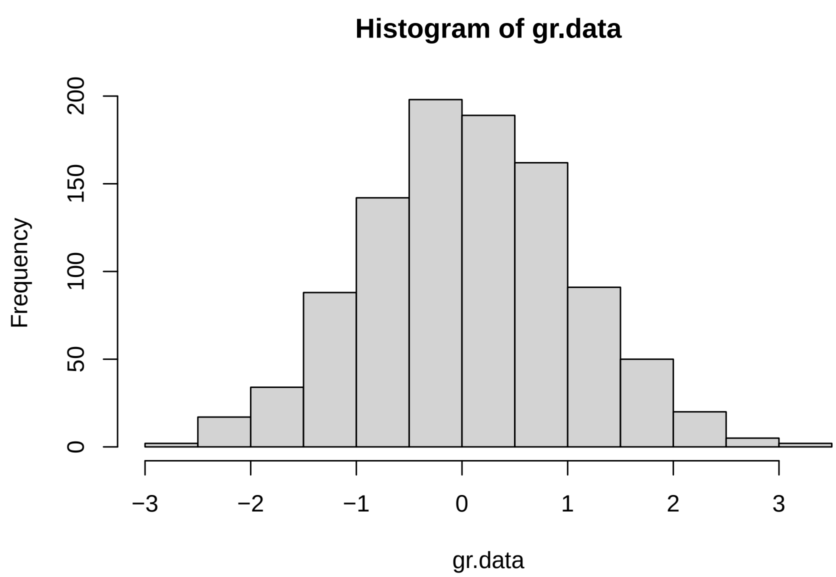
Note that the prompt appear followed by the mean of object `yy`.

(h) Objects created during an R session in the *workspace* are stored in a database *.RData* in the current folder. A listing of all the objects in a database can be obtained by calling the functions `ls()` or `objects()`. Now, first enter, at the R prompt, the instruction `objects` (or `ls`) and then the instruction `objects()` (or `ls()`). Explain what has happened.

(i) Objects can be removed by the following instruction: `rm(name1, name2, ... )`.

(j) Apart from the *console* there are several other types of windows available in R e.g. graphs are displayed in graph windows. To illustrate, enter the following instructions at the R prompt in the console or commands window:

```
gr.data <- rnorm(1000)
hist(gr.data)
```



These instructions have resulted in the opening of a graph window containing the required histogram and the user can switch from the console to the graph window and back again to the console.

(k) The R session can be terminated by closing the window or entering `q()` at the R prompt. Either way the user is prompted to save the *workspace*. If the user chooses not to save, all objects created during the session are lost.

1.4 Working with RStudio

Many users of R prefer working with *RStudio*. *RStudio* is a free and open source integrated development environment for R which works with the standard version of R available from CRAN. It can be downloaded from the RStudio home page to be run from your desktop (*Windows*, *Mac* or *Linux*). Full details about the functionality of *RStudio* are available from its home page. Here, only a brief introduction to *RStudio* is given.

When *RStudio* is installed on your computer the following icon is created on the desktop:

Clicking the above icon open the *RStudio* development environment as shown in Figure 1.1. In order to open any R *workspace* with *RStudio* drag the corresponding *.RData* file to the above *RStudio* icon and drop it as soon as 'Open with RStudio' becomes visible.

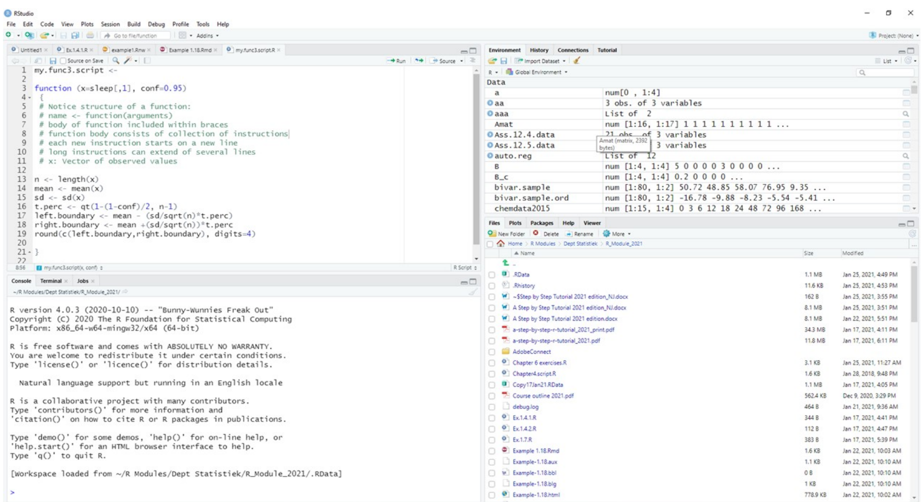


Figure 1.1: The RStudio development environment for R.

The bottom left-hand panel is the familiar R console.

The bottom right-hand panel is used for : (a) a listing of the files in the folder where the *workspace* (*.RData*) for the active project is kept (b) a listing of all installed packages available to be attached to the search path as well as menus for installing and updating packages (c) the graph windows (if any) (d) the Help facilities.

The top left-hand panel can be used for creating and managing script files (see 1.9.1) while the top right-hand panel provides information on the objects in the current folder as well as the history of previous commands given in the console.

1.5 R: an interpretive computer language

Essentially, in an interpretive language instructions are given one by one. Each instruction is then evaluated or interpreted in turn by an internal program called an *interpreter* or *evaluator* and some immediate action is taken. For example, the instruction given in 1.3(a) is evaluated by the R evaluator resulting in the answer `-3` being returned. On the other hand, in 1.3(b) the evaluator found the instruction to be incomplete and therefore asked for more information.

An advantage of an interpretive language is that intermediate results can be obtained quickly without having first to wait for a complete program to finish as is the case with a compiler language. In the latter case a complete program is translated (or compiled) by a program called a compiler. The compiled program can then be converted to a standalone application that can be called by other programs to perform a complete task. In general compiler languages handle computer memory relatively more efficiently and calculations are executed more speedily. Communication with the R evaluator takes place through a set of instructions called *escape sequences*. These escape sequences take the form of a backslash preceding a character. Examples of such escape sequences are:

`\n` new line

`\r` carriage return

`\t` go to next tab stop

`\b` backspace

`\a` bell

`\f` form feed

`\v` vertical tab

A consequence of the above role of the backslash in R is that a single backslash in a filename will not be properly recognized. Therefore, when referring in R to the following file path *"c:\My Documents\myFile.txt"* all backslashes must be entered as double backslashes i.e. `"c:\\My Documents\\myFile.txt"` or as `"c:/My Documents/myFile.txt"`.

1.5.1 Exercise

The `cat()` function can be used to write a text message to the console. Initialize a new R session and investigate the results of the following R instructions:

```
cat("aaa bbb")
cat("aaa bbb \n")
cat("aaa \n bbb \n")
cat("aaa \nbbb \n")
cat("aaa \t\t bbb \n")
cat("aaa\b\b\bbbb \n")
cat("aaa \n\a bbb \a\n")
cat("1\a\n"); cat("2\a\n")
```

What is the purpose of the semi-colon in the line above?

Could you distinguish the two soundings of the bell? Try the following:

```
cat("1\a\n"); Sys.sleep(2); cat("2\a\n")
```

Could you now distinguish the two soundings of the bell?

What is the purpose of the `Sys.sleep()` instruction?

1.5.2 Exercise

Write R code to achieve the following output:

`My name is:`

Bell sounds once.

Your name appears on a new line.

Two distinct sounds of the bell are heard and

`Thank you` is visible on a new line.

The cursor appears on a new line.

1.6 Accessing the Help functionality

(a) Use

```
?mean
```

to obtain help on the usage of the R function `mean()`.

(b) Find out what is the difference between the instructions

```
?mean
```

and

```
??mean
```

(c) What help is available via the instruction

```
help.start()
```

(d) Use

```
?help.search()
```

to find out how to obtain help using the R function `help.search(xx)`. Note: For hep on an operator or reserved word quotes are needed, e.g.

```
?matrix
```

but

```
?"?"
```

or

```
?"for"
```

1.7 More R basics

(a) R as an *interactive* language allows for fast acquisition of results.

(b) R is a *functional* language in two important senses: In a more technical sense it means the R model of computation relies more on *function evaluation* than by procedural computations and changes of state. The second sense refers to the way how users communicate to R namely almost entirely through *function calls*.

(c) R as an *object-oriented* language refers in a technical sense to the S4 or S5 type of objects with their associated classes and methods as mentioned in the Preface. In a less technical sense it means that everything in R is an object.

(d) R objects will be studied in detail in later chapters. What is important for now, is the following:

- Everything in R is an object.
- There are different types of objects e.g. function objects, data objects, graphics objects, character objects, numeric objects.
- Usually objects are stored in the current folder called the *Global environment*; recognized by R under the name `.GlobalEnv` and available in the file system under the name *.RData*.

- Objects are created from the console by *assignment* through the instruction

```
name <- object
```

or

```
object <- name
```

- In R names are *case sensitive* i.e. peter and Peter are two different objects.
- Objects created by assignment during an R session are stored permanently in the Global environment (working directory) unless the user chooses not to save when terminating an R session.
- Care must be exercised when creating a new object by assignment: if an object with the name my.object already exists in the Global environment and a new object is created by assigning it to the name my.object then the old my.object is over-written and it is replaced by the new object *without any warning*.
- Remember the way the R evaluator operates: if an object name is given at the R prompt the R evaluator responds by displaying the content of the object. Review the difference between the instructions

```
q
```

and

```
q()
```

(e) The symbol `#` marks a comment. Everything following a `#` on a line is ignored by the R evaluator. Check for example the result of the instruction

```
5+8 # +12
#> [1] 13
```

(f) Usage of the symbols `<-`, `=` and `==`. The symbol `<-` is used for assigning the object on its right-hand side to a name (label) on its left-hand side; the equality sign `=` is used for specifying the arguments of functions while the double equality symbol `==` is used for comparison purposes. In earlier versions of R these rules were strictly applied by the R evaluator. However, in recent versions of R the evaluator allows the equality sign also in the case for assigning an object to a name. We believe that reserving the equality sign only for argument specifications in functions leads to more clarity when writing complex functions and therefore we discourage its usage for creating objects by assignment. In this book creating objects by assignment will be exclusively carried out with the assignment symbol `<-`.

(g) The symbol `->` assigns the object on its left-hand side to the name (label) on its right-hand side.

(h) Working with packages: The core installation includes several packages. To see them issue the command `search()` from the R prompt in the console. Notice that the first object in the search list is `.GlobalEnv`. This is followed by other objects. Packages are recognized by the string package followed by a colon and the name of the package. In order for a package to be used the following steps must be followed: if the package has been *installed* previously it needs only to be *loaded* into the search path using the command `library(packagename)` from the R prompt. This will load the package by default in the second position on the search path. If the package has not been installed previously it must first be installed. This is most easily done using the top menu Packages. The command `require(packagename)` appears to be identical to `library(packagename)`. The function `require()` is designed for use inside other functions as it gives a warning, rather than an error, if the package does not exist.

(i) More on the help (`?`) facility: Table 1.1 contains details about help available for some special keywords.

Table 1.1: Some useful keywords available for help queries.

Help query	*Explanation*
`?Arithmetic`	Unary and binary operators to perform arithmetic on numeric and complex vectors
`?Comparison`	Binary operators for comparison of values in vectors
`?Control`	The basic constructs for control of the flow in R instructions
`?dotsMethods`	The use of the special operator `...`
`?Extract`	Operators to extract or replace parts of vectors, matrices, arrays and lists
`?Logic`	Logical operators for operating on logical and numeric vectors
`?.Machine`	Information on the variable `.Machine` holding information on the numerical characteristics of the machine R is running on
`?NumericConstants`	How R parses numeric constants including `Inf`, `NaN`, `NA`
`?options`	Allow the user to set and examine a variety of global options which affect the way in which R computes and displays its results
`?Paren`	Parentheses and braces in R

Help query	*Explanation*
`?Quotes`	Single and double quotation marks. Back quote (backtick) and backslash for starting an escape sequence
`?Reserved`	Description of reserved words in R
`?Special`	Special mathematical functions related to the beta and gamma functions including permutations and combinations
`?Syntax`	Outlines R syntax and gives the precedence of operators

1.8 Regular expressions in R: the basics

It follows from 1.7(d) that care must be taken when objects are assigned to names. Furthermore, the Global environment or any other R database may easily contain hundreds of objects. Therefore, a frequent task is to search for patterns in the names of objects e.g. searching for all object names starting with "Figure" or ending in ".dat". The R function `objects()` or `ls()` has arguments `pos` and `pattern` for specifying the position of a database to search and a pattern of characters appearing in a name (or string), respectively. The pattern argument can be given any *regular expression*. Regular expressions provide a method of expressing patterns in character values and are used to perform various tasks in R. Here we are only considering the task of extracting certain specified objects in a database using the pattern argument of `objects()` or `ls()`.

The syntax of regular expressions follows different rules to the syntax of ordinary R instructions. Moreover its syntax differs depending on the particular implementation a program uses. By default, R uses a set of regular expressions similar to those used by UNIX utilities, but function arguments are available for changing the default e.g. by setting argument `perl = TRUE`.

Regular expressions consist of three components: *single characters*, *character classes* and *modifiers* operating on single characters and character classes.

Character classes are formed by using square brackets surrounding a set of characters to be matched e.g. `[abc123]`, `[a-z]`, `[a-zA-Z]`, `[0-9a-z]`. Note the usage of the dash to indicate a range of values.

The modifiers operating on characters or character classes are summarized in Table 1.2.

Table 1.2: Modifiers for regular expressions.

Modifier	*Operation*
`^`	Expression anchors at beginning of target string
`$`	Expression anchors at end of target string
`.`	Any single character except newline is matched
`|`	Alternative patterns are separated
`( )`	Patterns are grouped together
`*`	Zero or more occurrences of preceding entity are matched
`?`	Zero or one occurrences of preceding entity are matched
`+`	One or more occurrences of preceding entity are matched
`{n}`	Exactly n occurrences of preceding entity are matched
`{n,}`	At least n occurrences of preceding entity are matched
`{n, m}`	At least n and at most m occurrences of preceding entity are matched

Because of their role as modifiers or in forming character classes the following characters must be preceded by a backslash when their literal meaning is needed:

```
[ ] { } ( ) ^ $ . | * + \
```

Note that in R this means that whenever one of the above characters needs to be escaped in a regular expression it must be preceded by double backslashes. Table 1.3 contains some examples of regular expressions.

Table 1.3: Examples of regular expressions.

Regular expression	*Meaning*
`"[a-z][a-z][0-9]"`	Matches a string consisting of two lower case letters followed by a digit
`"[a-z][a-z][0-9]$"`	Matches a string ending in two lower case letters followed by a digit
`"^[a-zA-Z]+\\."`	Matches a string beginning with any number of lower or upper case letters followed by a period
`"(ab){2}(34){2}$"`	Matches a string ending in `abab3434`

1.8.1 Exercise

Initialize an R session

(a) Attach the MASS package in the second (the default) position on the search path by issuing the command

```
library(MASS)
```

(b) Get a listing of all the objects in package MASS by requesting

```
objects(pos=2)
```

(c) Explain the difference between `objects(pos=2, pat=".")` and `objects(pos=2, patt="\\.")`.
(d) Obtain a listing of all objects with names starting with three letters followed by a digit.
(e) Obtain a listing of all objects with names ending with three letters followed by a digit.
(f) Obtain a listing of all objects with names ending in a period followed by exactly three or four letters.

1.9 From single instructions to sets of instructions: introducing R functions

Consider the following problem: the R data set `sleep` contains the extra hours of sleep of 20 patients after a drug treatment. Suppose this data set can be considered a sample from a normal population. A 95% confidence interval is required for the mean extra hours of sleep. It is known that the confidence interval is given by $\left[\bar{\mathbf{x}} - \left(\frac{s}{\sqrt{(n)}}\right) t_{n-1,0.025}; \bar{\mathbf{x}} + \left(\frac{s}{\sqrt{(n)}}\right) t_{n-1,0.025}\right]$. This problem can be solved by entering the following instructions one by one:

```
sleep.data <- sleep[ ,1]
sleep.mean <- mean(sleep.data)
sleep.sd <- sd(sleep.data)
t.perc <- qt(0.975,19)
left.boundary <- sleep.mean - (sleep.sd/sqrt(length(sleep.data)))*t.perc
right.boundary <- sleep.mean + (sleep.sd/sqrt(length(sleep.data)))*t.perc
cat ("[", left.boundary, ";", right.boundary, "]\n")
#> [ 0.5955845 ; 2.484416 ]
```

In situations like the above, the problem can be addressed using a *script file* or writing a *function*. We are going to introduce two methods for writing functions in R:

(i) using a script file and
(ii) using the function `fix()`.

1.9.1 Writing an R function using a script file

(a) From the R top menu select *File; New script*. A script window will open with a simultaneous change in the menu bar.
(b) Type the instructions in the script window.
(c) Select all the typed text and run the script by clicking the run icon (or Ctrl+R).
(d) Note what is shown in the R console window.
(e) Script files are ordinary text files. They can be saved, edited and opened using any text editor.
(f) By convention R script files have the extension xxxx.r.
(g) Next, change the spelling in the last two lines from `right.boundary` to `Right.boundary`. Select all the text and run the script. Check the output appearing on the console.
(h) Script windows can also be used for creating an R function.
(i) Create an R function by changing the text as shown below.

```
conf.int <- function (x = sleep[,1])
{
  x.mean <- mean(x)
  x.sd <- sd(x)
  t.perc <- qt(0.975,19)
  left.boundary <- x.mean - (x.sd/sqrt(length(x)))*t.perc
  right.boundary <- x.mean + (x.sd/sqrt(length(x)))*t.perc
  list (lower = left.boundary, upper = right.boundary)
}
```

(j) Select the text and notice what happens in the R commands window (the console).
(k) Give the instruction `objects()` at the R prompt. What has happened?
(l) You can now run the function from the commands window (the console) by typing:

```
conf.int (x = sleep[,1])
#> $lower
#> [1] 0.5955845
#>
#> $upper
#> [1] 2.484416
```

(l) If you want to create and run the function `conf.int` in a script window then add the instruction `conf.func (x = sleep[,1])` as the last line in the script window. Now, select only this line and run it. Check the R console.

(m) What will happen if a syntax error is made in the script window? Change the code in the script file as follows, deliberately deleting the last closing parenthesis in the last line of the function.

```
conf.int <- function (x = sleep[,1])
{
  x.mean <- mean(x)
  x.sd <- sd(x)
  t.perc <- qt(0.975,19)
  left.boundary <- x.mean - (x.sd/sqrt(length(x)))*t.perc
  right.boundary <- x.mean + (x.sd/sqrt(length(x)))*t.perc
  list (lower = left.boundary, upper = right.boundary
}
conf.int (x = sleep[,1])
```

(n) Select *only the final line* and run it. Check the R console. No problem, the function executed correctly. This is because the code for `conf.int` in the script file was changed, but the updated object was not created by running it in the console.
(o) Select *all the code* in the script and run it. Check the R console. Discuss.

1.9.2 Writing an R function using `fix()`

When using `fix()` the built-in *R text editor* can be used when using script files but in the windows environment notepad or preferably notepad++ or Tinn-R is preferred.

The following instruction is necessary for changing the default editor to be used with `fix()`:

```
options(editor = "notepad")
```

or

```
options(editor = "full path to the relevant exe file")
```

(a) Enter `fix (my.func)` at the R prompt. A text editor will open. Type the instructions as shown below.

```
function (x = sleep[,1])
{
  x.mean <- mean(x)`
  x.sd <- sd(x)
```

```
  t.perc <- qt(0.975,19)
  left.boundary <- x.mean - (x.sd/sqrt(length(x)))*t.perc
  right.boundary <- x.mean + (x.sd/sqrt(length(x)))*t.perc
  list (lower = left.boundary, upper = right.boundary)
}
```

Close the window. Check what happens in the R console.

You can now run the function from the commands window (the console) similar to in 1.9.1(l), but changing the name of the function from `conf.int` to `my.func`.

(b) What will happen if a syntax error is made when using fix? At the R prompt type `fix (my.func)`. Make a deliberate syntax error, e.g. delete the last closing brace. Close the text editor window. What happens in the console? What is to be done to correct the mistake?

(c) Carefully study the message in the R console when a syntax error occurred in a function created by `fix()`:

```
> Error in edit(name, file, title, editor) :
    unexpected 'yyy' occurred on line xx
    use a command like
    x <- edit()
    to recover
```

(d) The following is the correct way to respond to the above message from the R evaluator:

```
my.func <- edit()
```

If you simply use `fix(my.func)` at this point, the R and the editor will revert to the version of the function *before* the previous edit.

WARNING

Before writing a function for solving any problem: make sure the problem is understood exactly; make 100% sure the relevant statistical theory is understood correctly. Failure to do so is careless and dangerous!

1.10 R Projects

The different windows in R are the Data window, Script window, Graph window and Menus and Dialog windows. The current *workspace* in R is `.GlobalEnv`.

The function `getwd()` is used to obtain the path to the current folder's *.Rdata* and *.Rhistory*.

Note: In order to see the files *.Rdata* and *.Rhistory* being displayed as such, it may be necessary to turn off the option "Hide extensions for known file types" in *Windows Explorer*.

It is important to make provision for different *workspaces* associated with different *projects*. In R, different *.Rdata* files in different folders would separate different projects. There is however much to gain in using *Projects* in *RStudio*.

1.10.1 Creating a project in RStudio

From the top menu, select *File, New Project*. Follow the prompts to create a new project, either in an existing folder or creating a new folder for your project, say *MyProject*.

(a) Navigate to the folder *MyProject* in *Windows Explorer*.

(b) Notice a file *MyProject.Rproj* has been created in the folder.

(c) By double-clicking on this file you open the project in *RStudio*. The advantages of opening the project this way are:

- your *workspace* from the file *MyProject.Rdata* is automatically loaded
- by placing any related files like data set in the folder *MyProject* or a subfolder, say *MyProject\data* means that in your code you only have to use relative folder references, i.e. refer to *MyProject\mydata.xlsx* or *MyProject\data\mydata.xlsx* instead of something like *c:\users\myname\Documents\MyProject\data\mydata.xlsx*.
- the major advantage of relative references is that it is not specific to the computer and makes porting between devices possible
- sharing your project with a collaborator will simply entail copying the entire contents of the *MyProject* folder.

1.11 A note on computations by a computer

When writing R functions it is important to keep in mind that the way computations are performed by a computer are not always according to the rules of algebra. Two important occurrences are given below.

- In mathematics the following statement is incorrect: `x = x + k` for $k \neq 0$ but in computer programming the statement `x = x + k` is legitimate and it means `x` is replaced by `x + k`.
- In general, the treatment of integers and real numbers for which R uses floating point representation happens at a fundamental level over which R has no control. Real numbers cannot necessarily be exactly represented in a computer – some can only be approximated. Furthermore, there are limitations to the minimum and maximum numbers that can be represented in a computer. This might lead to what is known as *underflow* or *overflow*. A more detailed discussion appears in a later chapter.

Open an R session and issue the command

```
.Machine
```

for details about the numerical environment of your computer.

1.12 Built-in data sets in R

R contains several built-in data sets collected in the package `datasets`. This package is automatically attached to the search path. Type `?datasets` at the R prompt for details. Apart from these data sets several other data sets from other packages are also used in this book.

1.13 The use of `.First()` and `.Last()`

The function `.First()` is executed at the beginning of every R session. *This only works in R and not in RStudio.*

Instead of having to specify

```
options(editor = "notepad")
```

each time an R session is initialized, create the following function and save in the .Rdata before exiting R.

```
.First <- function() { options(editor = "notepad") }
```

to ensures that Notepad is the text editor during any subsequent session.

Similar to `.First()` the function `.Last()` can be created for execution at the end of an R session.

1.13.1 Security: an example of the usage of `.First()`

The `.First()` facility can be used to prevent access to a R *workspace* by setting a password protection. This can be done as follows:

Create a new *workspace* for running the example on security. In this *workspace* create the following R function

```
password <- function()          # Note the structure of a function
{ cat("Password? \n")
  password <- readline()        # What is the usage of readline()?
  if (password != "PASSWORD")
    q(save="no")                # The meaning of !=  is "not equal to"
  else (cat("You can proceed \n"))
}
```

Now create the function:

```
.First <- function()
{   #  What must you be careful of?
   password()
}
```

- Terminate your R session and open it again.
- Discuss the construction and usage of the above functions.
- Can you break the above security?
- Can you make changes to the above security to make it more safe?

1.14 Options

Study the result of the instruction `> options()` in R.

1.15 Creating PDF and HTML documents from R output: R Markdown

The R package `knitr` is used to obtain reproducible results from R code in the form of *.pdf* or *.html* documents. In addition to `knitr`, *R Markdown* can be used to create *.html*, *.pdf* or even *MS Word* documents. *Markdown* is a so-called markup language with plain-text-formating syntax. An *R Markdown* document is written in markdown and contains chunks of embedded R code. Although the `render()` function in the package `rmarkdown` can be used (similar to the `knit()` function from the package `knitr`), to create the output document from the *R Markdown .Rmd* file, *R Markdown* is typically used in conjunction with *RStudio*. In the top menu, select *File, New File, R Markdown...* to open the *example.Rmd* file providing the user with the structure of an *R Markdown* file. For our illustration, we will select the output format as *.html*.

Edit the *example.Rmd* file to contain the following:

````
---
title: "An Illustration of Some Capabilities of R Markdown"
author: "Niel le Roux and Sugnet Lubbe"
date: "22/01/2021"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## Short description

Code chunks in .Rmd files are delimited with ` ```{r} ` at the top where a chunk
label and any chunk options can appear and  ` ``` ` at the end. In-line R code
chunks are indicated with single ` `r ` on either side.

*****

Here is an example containing several chunks of code. Note that in the first
chunk R code is not shown due to the option `echo = FALSE`. In the remaining
chunks R code is shown due to the option above 'echo = TRUE'.

_Note R code not shown for this chunk._

```{r y, echo=FALSE}
y <- 1
y
```

```{r rnorm}
require(lattice)
set.seed(123)
x <- rnorm(1000, 20, 5)
```

We analyse data drawn from $\mathcal{N}(20,25)$. The mean is
`r round(mean(x),3)`. The following code shows the distribution via a histogram
````

````
```{r histexample}
  hist(x)
```

and the code below via a boxplot.

```{r boxexample}
  boxplot(x)
```

The first element of \texttt{x} is `r x[1]`. Note the usage of ` \texttt{x} `
above.

*two plots side by side (option fig.show='hold')*

```{r side-by-side, fig.show='hold', out.width="50%"}
  par(mar=c(4,4,0.1,0.1), cex.lab=0.95, cex.axis=0.9, mgp=c(2,0.7,0),
      tcl=-0.3, las=1)
  boxplot(x)
  hist(x,main="")
```

```{r linear_model}
  n <- 10
  x <- rnorm(n)
  y <- 2*x + rnorm(n)
  out <- lm(y ~ x)
  summary(out)$coef
```
````

At the top of the text editor, click on *Knit* to create the *.html* document. Note that with the down arrow, options *Knit to PDF* and *Knit to Word* can also be chosen. The output format is also specified in line 5 of the text file with `output: html_document`. Had we chosen *.pdf* as output format, it would be `output: pdf_document`. Typically, *R Markdown* is used for reporting, directly incorporating the R code and output. For more formal documents with Figure and Table caption references, tables of content, etc. the R package `bookdown` should be used. Install the package and replace the output statement with `output:bookdown::pdf_document2`. For more information on the use of bookdown, click here.

1.16 Command line editing

Commands given in an R session are stored together with commands given in previous sessions in a file *.History* in the same folder as the *.RData* file. In an R session previous commands can be retrieved at the R prompt by pressing the *up* and *down* arrow keys. A previous command can then be edited using the *backspace*, *delete*, *home*, *end* keys as well as the shortcuts for *copy* and *paste*.

Chapter 2

Managing objects

After completing the introductory chapter you now know how to

- initialize an R session;
- save your workspace;
- open an existing project;
- execute simple tasks in R to obtain numerical, text or graphical results;
- obtain help.

You know also that everything in R can be considered as some kind of an object. In this chapter the focus is on what properties the different objects have and how to manage objects in the workspace.

2.1 Instructions and objects in R

2.1.1 General

Recall that

- instructions are separated by a semi-colon or start on new lines;
- the `#` symbol marks the rest of the line as comments;
- the default R (primary) prompt is `>`; the secondary default prompt is `+`;
- use of `<-` to create objects. (The equality sign (`=`) will also be accepted. However, avoid this practice and use
  - `=` only for function arguments;

    - `<-` for assignment;
    - `==` for comparison / control structures);

- the use of `->` for assigning left-hand side to the name on right-hand side.
- the use of function `assign()` for assigning names to objects. (to be discussed in detail in Chapter 3)

```
aa <- 1:10
```

Examples Assigning numeric vector to name “aa”. Assignment takes place in global environment.

```
Aa <- seq(from = 1,to = 10,by = 0.01); yy <- c("a","b","c")
c("a","b","c") -> bb
```

Assigning character vector to name “bb”.

```
assign("aa", rnorm(10), pos = 1)
```

Note the use of the argument `pos`, ” ” or ’ ’ are used for characters. Be careful when mixing single quotes and double quotes. See below.

```
c("u",'v',"'w'",""x"",'"y"',''z'') -> cc
#> Error in parse(text = input): <text>:1:19: unexpected symbol
#> 1: c("u",'v',"'w'",""x
#>                       ^
```

```
c("u",'v',"'w'",'"x"','"y"',''z'') -> cc
#> Error in parse(text = input): <text>:1:31: unexpected symbol
#> 1: c("u",'v',"'w'",'"x"','"y"',''z
#>                                   ^
```

```
c("u",'v',"'w'",'"x"','"y"','z') -> cc
cc
#> [1] "u"     "v"     "'w'"   "\"x\"" "\"y\"" "z"
```

- Explain error message above.
- Explain backslash above.

```r
objects()
#> [1] "aa" "Aa" "bb" "cc" "yy"
aa
#>  [1] -1.30237657  1.35234865 -0.32712077  0.32666819
#>  [5]  0.12642658 -0.39484180 -0.72774624  0.09173947
#>  [9]  0.83248656  0.49906948
bb
#> [1] "a" "b" "c"
objects()[3]
#> [1] "bb"
parse(text=objects()[3])
#> expression(bb)
eval(parse(text=objects()[3]))
#> [1] "a" "b" "c"
rm(a,b)
#> Warning in rm(a, b): object 'a' not found
#> Warning in rm(a, b): object 'b' not found
rm(aa,bb)
objects()
#> [1] "Aa" "cc" "yy"
rm("cc")
objects()
#> [1] "Aa" "yy"
```

2.1.2 Objects in R

(a) Everything is an object but there are many different types of objects.

(b) Study and also take note of the following *naming conventions*:

- Allowed are upper or lower case letters, numbers 0 – 9, full stop(s) and underscore(s).
- Must not begin with a number.
- R is case sensitive i.e. `John` and `john` refer to different objects.
- Use full stops (periods) or underscores to break up a name into meaningful words.
- Avoid `c`, `s`, `t`, `C`, `F`, `T`, `diff` as well as other reserved words for naming an object.

(c) The use of the functions `conflicts()` and `find()` when naming objects. The instruction `conflicts (detail = TRUE)` outputs details on whether and where objects with identical names exist on the search path e.g.

```
conflicts(detail=TRUE)
#> $`package:graphics`
#> [1] "plot"
#>
#> $`package:methods`
#> [1] "body<-"    "kronecker"
#>
#> $`package:base`
#> [1] "body<-"    "kronecker" "plot"
```

The instruction `find ("object")` outputs details on whether and where objects with the name object exist on the search path e.g.

```
find("kronecker")
#> [1] "package:methods" "package:base"
```

(d) Objects can possess several attributes e.g.

- mode (The way an object is internally stored)
- length
- names
- dim
- class

Examples

```
a <- 1:10
class(a)
#> [1] "integer"
b <- factor(c("a","b","c"))
class(b)
#> [1] "factor"
b
#> [1] a b c
#> Levels: a b c
mode(a)
#> [1] "numeric"
mode(b)
#> [1] "numeric"
length(a)
#> [1] 10
length(b)
```

```
#> [1] 3
dim(a)
#> NULL
mat <- matrix(1:12,nrow=4)
mat
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
dim(mat)
#> [1] 4 3
mode(mat)
#> [1] "numeric"
logic <- c(TRUE,TRUE,FALSE,TRUE)
mode(logic)
#> [1] "logical"
class(logic)
#> [1] "logical"
```

Levels show that it is a categorical variable (object).

Mode `numeric` tells us that the categorical variable (object) `b` is internally stored as a set of numeric codes.

(e) Special attention is given to the class and mode of integers. An object of type integer is stored internally more effectively than an integer represented in double format.

```
x <- 5
y <- 5L
typeof(x)
#> [1] "double"
typeof(y)
#> [1] "integer"
class(x)
#> [1] "numeric"
class(y)
#> [1] "integer"
mode(x)
#> [1] "numeric"
mode(y)
#> [1] "numeric"
```

(f) Objects in R are *vectors*, *functions* or *lists*. There are no scalars - instead vectors of length one are used. In addition to the above three types, there are several other types of objects.

(g) Objects that are created during a session are permanently stored in the *.RData* file in the folder containing the *workspace* (unless not saved at termination).

(h) Objects that are created within a function exist only for as long as the function is being executed.

(i) Use of `rm()` and `rm(list = ListOfNames)` to remove objects from the *workspace*.

(j) Use of `objects()` or equivalently `ls()` to obtain a list of object names in a data base (by default the *workspace*). Note the optional arguments `pos`, `all.names` and `pattern` to specify which database to be considered and what object names to include.

(k) How can an object be printed to the screen?

(l) *Warning:* If a new object is assigned to a name that already exists in the working directory the old object is overwritten without warning and it cannot be retrieved again.

2.1.3 Data in R

(a) R has several built-in data sets. Use `?datasets` and/or `library(help= "datasets")` for details. Note that the two instructions return different information.

(b) Study the help file of `c()`.

(c) Study the help file of `scan()`.

(d) Study the help files of `read.table()` and `read.csv()`. Care must be taken with data containing characters (text) and categorical variables. Reading data into R will be discussed in detail in Chapter **??**.

2.1.4 Generation of data

Study the operators and functions `:`, `seq()`, `rep()`, `rev()`, `rnorm()`, `runif()` with the following instructions:

```
1:10
8:3
seq(from=1, to=10, length=10)
seq(from=2, to=10, length=5)
```

```
rev(10:1)
rnorm (20, mean=50, sd=5)
runif (10, min=1, max=3)
```

The function `rmvnorm()` for generating multivariate normal samples is in the `mvtnorm` R package. This package must first be loaded by using the instruction

```
library(mvtnorm)
```

Alternatively, for generating multivariate normally data there is also a function `mvrnorm()` in R package `MASS`.

2.2 Introduction to functions in R

We introduced R functions in section 1.9. The basic structure of an R function is as follows:

```
func.name <- function(list of arguments)
{
  # R code
}
```

When the function `func.name()` is called, the code in `{ }` is executed.

The arguments of a function can be inspected by using the command

```
args(name of function)
```

The function `str(x)` provides information on the object `x`. If `x` is a function its output is similar to that of `args()`. Default values are given to function arguments using the construction (`argument name = value`). It is good programming practice to make extensively use of comments to describe arguments and / or what a particular chunk of code does. What is the usage of the following function:

```
cube <- function(a) a^3
```

In the above function the argument a is called a *dummy argument*. What will happen to an object `a` in the working directory?

Functions are called by replacing the *formal arguments* by the *actual arguments*. This can be done *by position* or *by name*. *Hint*: It is less error prone to call functions using named arguments. Create the following function

```r
Demofunc <- function(vec = 1:10, m,k)
 { # Function to subtract a specified constant from
   # each element of a given vector and after subtraction
   # divide each element by a second specified constant.
   # The result of the above transformation is returned.
 (vec - m)/ k
}
```

Execute the following function calls and explain the output

```r
Demofunc(3, 2, 5)
#> [1] 0.2
Demofunc(2,5)
#> Error in Demofunc(2, 5): argument "k" is missing, with no default
Demofunc(m = 2, k = 5)
#>  [1] -0.2  0.0  0.2  0.4  0.6  0.8  1.0  1.2  1.4  1.6
Demofunc(m = 2, k = 5, vec = 1:100)
#>   [1] -0.2  0.0  0.2  0.4  0.6  0.8  1.0  1.2  1.4  1.6  1.8
#>  [12]  2.0  2.2  2.4  2.6  2.8  3.0  3.2  3.4  3.6  3.8  4.0
#>  [23]  4.2  4.4  4.6  4.8  5.0  5.2  5.4  5.6  5.8  6.0  6.2
#>  [34]  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.2  8.4
#>  [45]  8.6  8.8  9.0  9.2  9.4  9.6  9.8 10.0 10.2 10.4 10.6
#>  [56] 10.8 11.0 11.2 11.4 11.6 11.8 12.0 12.2 12.4 12.6 12.8
#>  [67] 13.0 13.2 13.4 13.6 13.8 14.0 14.2 14.4 14.6 14.8 15.0
#>  [78] 15.2 15.4 15.6 15.8 16.0 16.2 16.4 16.6 16.8 17.0 17.2
#>  [89] 17.4 17.6 17.8 18.0 18.2 18.4 18.6 18.8 19.0 19.2 19.4
#> [100] 19.6
```

Note the use of `prompt()` and `package.skeleton()` to provide a new function with a help-file.

The final expression in an R function is automatically returned when the function completes execution.

```r
my.func <- function(a=5)
{  a+2
}
my.func()
#> [1] 7
```

When a function consists of a single line, it can be written more succinctly

```r
my.func <- function(a=5) {  a+2  }
my.func()
#> [1] 7
```

or even without the `{ }`:

```
my.func <- function(a=5) a+2
my.func()
#> [1] 7
```

In general, functions will consist of more lines of code and often multiple outputs are returned. If only a single output object needs to be returned, the object can be created in the last line of the code

```
my.func <- function(a=5)
  {  number <- (a+3)^2
     number/a
  }
my.func()
#> [1] 12.8
```

or with a `return()` statement:

```
my.func <- function(a=5)
  {  number <- (a+3)^2
     return(number/a)
  }
my.func()
#> [1] 12.8
```

In general, all the outputs are combined and returned as a `list`. The final expression in the function creates the list object:

```
my.func <- function(a=5)
  {  number <- (a+3)^2
     list(number/a)
  }
my.func()
#> [[1]]
#> [1] 12.8
```

To return multiple outputs, the list is simply extended as shown below:

```
my.func <- function(a=5)
  {  number <- (a+3)^2
     list(number, number/a)
  }
my.func()
```

```
#> [[1]]
#> [1] 64
#>
#> [[2]]
#> [1] 12.8
```

It is good practice to name the output objects in the list, such as:

```
my.func <- function(a=5)
  {  number <- (a+3)^2
     list(number = number, ratio = number/a)
  }
my.func()
#> $number
#> [1] 64
#>
#> $ratio
#> [1] 12.8
```

Finally, to place the output into an object for further processing, the function is assigned to an object name:

```
my.func <- function(a=5)
  {  number <- (a+3)^2
     list(number = number, ratio = number/a)
  }
out <- my.func()
out
#> $number
#> [1] 64
#>
#> $ratio
#> [1] 12.8
```

2.3 How R finds data

In order to understand how objects are found by R it is necessary to have some understanding of the concepts

- Environment
- Frame
- Search path
- Parent environment

- Inheritance.

The mechanism that R uses to organize objects is based on frames and environments. A *frame* is a collection of named objects and an *environment* consists of a frame together with a pointer or reference to another environment called the *parent environment*. Environments are nested so that the *parent environment* is the environment that directly contains the current *environment*. At the start of an R session a *workspace* is created which always has an associated environment, the *global environment*. The global environment occupies the first position on the *search path* and is accessed by a call to `globalenv()`. Packages and databases can be added to the search path by a call to `attach()` and removed from the search path by a call to `detach()`.

- What is an R *package*? What is the difference between *installing* and *loading* a package?
- Work through the following example:

```
search()
#> [1] ".GlobalEnv"        "package:stats"
#> [3] "package:graphics"  "package:grDevices"
#> [5] "package:utils"     "package:datasets"
#> [7] "package:methods"   "Autoloads"
#> [9] "package:base"
```

To attach the package `MASS`

```
library (MASS)
```

By default `MASS` is attached in the second position in the search path.

```
search()
#>  [1] ".GlobalEnv"        "package:MASS"
#>  [3] "package:stats"     "package:graphics"
#>  [5] "package:grDevices" "package:utils"
#>  [7] "package:datasets"  "package:methods"
#>  [9] "Autoloads"         "package:base"
```

We use `detach` to remove `MASS` from the search path.

```
detach("package:MASS")
search()
#> [1] ".GlobalEnv"        "package:stats"
#> [3] "package:graphics"  "package:grDevices"
```

```
#> [5] "package:utils"     "package:datasets"
#> [7] "package:methods"   "Autoloads"
#> [9] "package:base"
```

To obtain the parent of the global environment

```
parent.env(.GlobalEnv)
#> <environment: package:stats>
#> attr(,"name")
#> [1] "package:stats"
#> attr(,"path")
#> [1] "C:/Program Files/R/R-4.5.1/library/stats"
parent.env(parent.env(.GlobalEnv))
#> <environment: package:graphics>
#> attr(,"name")
#> [1] "package:graphics"
#> attr(,"path")
#> [1] "C:/Program Files/R/R-4.5.1/library/graphics"
parent.env(parent.env(parent.env(.GlobalEnv)))
#> <environment: package:grDevices>
#> attr(,"name")
#> [1] "package:grDevices"
#> attr(,"path")
#> [1] "C:/Program Files/R/R-4.5.1/library/grDevices"
environmentName(parent.env(parent.env(parent.env(.GlobalEnv))))
#> [1] "package:grDevices"
```

When the R evaluator looks for an object and it cannot find the name in the global environment it will search the parent of the global environment. It will carry on the search along the search path until the first occurrence of the name. If the name is not found it will return the message `Error: object 'xx' not found`. The usage of the double colon `::` and the triple colon `:::` is to access the intended object when more than one object with the same name exist on the search path. These two operators use the *namespace* facility of R packages. The *namespace* of a package allow the creator of a package to hide functions and data that are meant only for internal use; it provides a way through the operators `::` and `:::` to an object within a particular package. Thus a namespace prevent functions from breaking down when a user selects a name that clashes with one in the package. The double-colon operator `::` selects objects from a particular namespace. Only functions that are exported from the package can be retrieved in this way. The triple-colon operator `:::` acts like the double-colon operator but also allows access to hidden objects. Packages are often inter-dependent, and loading one may cause others to be automatically loaded. Such automatically loaded packages are not added to the search list.

We note that the *function* call `getAnywhere()`, which searches multiple packages can be used for finding hidden objects. When a function is called, R creates a new (temporary) environment which is enclosed in the current (calling) environment. Objects created in the new environment are not available in the parent environment and dies with it when the function terminates. Objects in the calling environment are available for use in the new environment created when a function is called.

Similarly, when an *expression* is evaluated a hierarchy of environments is created. Search for objects continue up this hierarchy and if necessary to the global environment and from there up onto the search path.

- Study the use of the arguments `pos`, `all.names`, and `pattern` of the function `objects()`.
- Study the behaviour of the functions `conflicts()` and `exists()` in the examples below:

```
conflicts()
#> [1] "body<-"    "kronecker" "plot"
conflicts(detail=TRUE)
#> $`package:graphics`
#> [1] "plot"
#>
#> $`package:methods`
#> [1] "body<-"    "kronecker"
#>
#> $`package:base`
#> [1] "body<-"    "kronecker" "plot"
exists("kronecker")
#> [1] TRUE
exists("kronecker", where = 1)
#> [1] TRUE
exists("kronecker", where = 1, inherits = FALSE)
#> [1] FALSE
exists("kronecker", where = 2)
#> [1] TRUE
exists("kronecker", where = 2, inherits = FALSE)
#> [1] FALSE
exists("kronecker", where = 7, inherits = FALSE)
#> [1] TRUE
exists("kronecker", where = 8, inherits = FALSE)
#> [1] FALSE
exists("kronecker", where = 9, inherits = FALSE)
#> [1] TRUE
```

- Study the above code carefully and then explain what inheritance does.

- The example below leads to the same conclusion as above but is more complicated at this stage. Its behaviour will become clear as we work through the coming chapters.

```
sapply(search(), function(x) exists("kronecker", where = x, inherits=FALSE))
#>        .GlobalEnv     package:stats  package:graphics
#>             FALSE             FALSE             FALSE
#> package:grDevices     package:utils  package:datasets
#>             FALSE             FALSE             FALSE
#>   package:methods         Autoloads      package:base
#>              TRUE             FALSE              TRUE
```

- Direct access to objects down the search path can be achieved with the function `get()`. The function `get()` takes as its first argument the name of an object as a character string. The optional argument `pos` can be used to specify where on the search list to look for the object. As an illustration explain the outcomes of the following function calls:

```
get ("%o%")
#> function (X, Y)
#> outer(X, Y)
#> <bytecode: 0x0000020531c4c990>
#> <environment: namespace:base>
mean <- mean (rnorm (1000))
get (mean)
#> Error in get(mean): invalid first argument
get ("mean")
#> [1] -0.02192818
get ("mean", pos = 1)
#> [1] -0.02192818
get ("mean", pos = 2)
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x0000020529b13530>
#> <environment: namespace:base>
rm (mean)
```

- Instead of attaching databases the function `with()` is often to be preferred. Discuss the usage of `with()` by referring to the instructions:

```
with (beaver1, mean(time))
#> [1] 1312.018
with (beaver2, mean(time))
#> [1] 1446.2
```

2.4 The organisation of data (data structures)

Study the help files of `list()`, `matrix()`, `data.frame()` and `c()` carefully.

A *list* is created with the function `list()`. A list is the basic means of storing a collection of data objects in R when the modes and/or lengths of the objects are different. List elements are accessed using `[[ ]]` or `$` when the objects are named. List objects are named using the construction

```
my.list <- list(name1 = 1:10, name2 = mean)
my.list
#> $name1
#>  [1]  1  2  3  4  5  6  7  8  9 10
#>
#> $name2
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x0000020529b13530>
#> <environment: namespace:base>
```

and elements are retrieved using the instruction

```
my.list[[2]]
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x0000020529b13530>
#> <environment: namespace:base>
my.list$name2
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x0000020529b13530>
#> <environment: namespace:base>
```

A *matrix* in R is a rectangular collection of data, all of the same mode (e.g. numeric, character/text or logical). It is formed with the construction

```
my.matrix <- matrix(1:12, ncol=3, nrow=4, byrow=FALSE)
my.matrix
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
```

Matrix elements are accessed using `my.matrix[i,j]`. The functions `nrow()`, `ncol()`, `dim()`, `dimnames()`, `colnames()` and `rownames()` are useful when working with matrices.

A *dataframe* is also a rectangular collection of data but the columns can be of different modes. It can be regarded as a cross between a list and a matrix. Dataframes are constructed with the function `data.frame()`.

Study the help files of the above functions.

2.5 Time series

Study the usage of the function `ts()`.

2.6 The functions `as.xxx()` and `is.xxx()`

The function `as.xxx()` transforms an object as best as possible to a specified type e.g. `as.matrix(mydata)` transforms the numerical dataframe to a numerical matrix. `is.xxx()` tests if the argument is of a certain type e.g. `is.matrix(mydata)` evaluates to false if `mydata` does not satisfy all the conditions of a matrix.

2.7 Simple manipulations; numbers and vectors

- Explain vector calculations and the recycling principle by referring to the example below.

```
c(1,3,5,9) + c(1,2,3)
#> Warning in c(1, 3, 5, 9) + c(1, 2, 3): longer object length
#> is not a multiple of shorter object length
#> [1]  2  5  8 10
```

- Logical vectors. Explain the behaviour of the instruction below

```
sum (c (TRUE, FALSE, TRUE, TRUE, FALSE))
#> [1] 3
```

- Missing values: `NA` (indicate a missing value in the data), `NaN` (not a number)

```
10/0
#> [1] Inf
0/0
#> [1] NaN
```

- Character vectors: see section 3.5.11
- Subscripting vectors: see section 5.1

2.8 Objects, their modes and attributes

- Vector elements must be of same mode: logical, numeric, complex, character
- Empty object; once created (e.g. `xx <- numeric()`) components may be added (e.g. `xx[5] <- 22`)
- Getting and setting attributes: The functions `attr()` and `attributes()`
- Class of an object and the function `unclass()` for removing class.

2.9 Representation of objects

We have already seen that a representation of an object can be obtained by calling (entering) its name:

```
cars
#>    speed dist
#> 1      4    2
#> 2      4   10
#> 3      7    4
#> 4      7   22
#> 5      8   16
#> 6      9   10
#> 7     10   18
#> 8     10   26
#> 9     10   34
#> 10    11   17
#> 11    11   28
#> 12    12   14
#> 13    12   20
#> 14    12   24
#> 15    12   28
#> 16    13   26
#> 17    13   34
```

```
#> 18     13   34
#> 19     13   46
#> 20     14   26
#> 21     14   36
#> 22     14   60
#> 23     14   80
#> 24     15   20
#> 25     15   26
#> 26     15   54
#> 27     16   32
#> 28     16   40
#> 29     17   32
#> 30     17   40
#> 31     17   50
#> 32     18   42
#> 33     18   56
#> 34     18   76
#> 35     18   84
#> 36     19   36
#> 37     19   46
#> 38     19   68
#> 39     20   32
#> 40     20   48
#> 41     20   52
#> 42     20   56
#> 43     20   64
#> 44     22   66
#> 45     23   54
#> 46     24   70
#> 47     24   92
#> 48     24   93
#> 49     24  120
#> 50     25   85
```

It is often not convenient to have a full representation returned of an object as above. The functions `head()`, `str()` and `summary()` are available for extracting a partial representation of an object:

```
head(cars)
#>   speed dist
#> 1     4    2
#> 2     4   10
#> 3     7    4
#> 4     7   22
#> 5     8   16
```

```
#> 6     9   10
summary(cars)
#>      speed           dist       
#>  Min.   : 4.0   Min.   :  2.00  
#>  1st Qu.:12.0   1st Qu.: 26.00  
#>  Median :15.0   Median : 36.00  
#>  Mean   :15.4   Mean   : 42.98  
#>  3rd Qu.:19.0   3rd Qu.: 56.00  
#>  Max.   :25.0   Max.   :120.00  
str(cars)
#> 'data.frame':    50 obs. of  2 variables:
#>  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
#>  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

There are many more R functions provided for getting information of what an R object represents. Some of these functions like `mode()`, `class()`, `length()`, `levels()`, `is.xxx()` and `as.xxx()` have already been encountered and others will be given in the chapters to come.

```
length(cars)
#> [1] 2
length(as.matrix(cars))
#> [1] 100
dim(cars)
#> [1] 50  2
is.matrix(cars)
#> [1] FALSE
is.data.frame(cars)
#> [1] TRUE
is.list(cars)
#> [1] TRUE
mode(cars)
#> [1] "list"
class(cars)
#> [1] "data.frame"
levels(cars)
#> NULL
```

2.10 Exercise

2.10.1 Exercise

According to the central limit theorem (CLT) the distribution of the sum (or mean) of independently, identically distributed stochastic variables converges to a normal distribution with an increase in the number variables. The binomial distribution can be expressed as the sum of independently, identically distributed Bernoulli stochastic variables and therefore converges in distribution to the normal distribution. The lognormal distribution in contrast cannot be expressed as a sum.

Make use of the function `rbinom()` to generate a sample of size 10 from a binomial distribution modelling 20 coin flips with a probability of 0.4 for returning "heads". Use the function `hist()` to graph the results. Repeat with sample sizes 50, 100, 1000, 10000 and 100000. Repeat the whole study with a success probability of 0.5, 0.3, 0.1 and 0.05. Discuss your findings.

Now repeat the same exercise using (a) the lognormal distribution with the function `rlnorm()` and (b) the uniform distribution over the interval $[10; 25]$ with the function `runif(min = 10, max = 25)`. Comment on your findings.

2.10.2 Exercise

Assume that a random sample of size n is available from a certain distribution. A bootstrap sample is obtained by sampling with replacement a sample of size n from the given sample. One of the uses of the bootstrap is to obtain an estimate of the standard error of a statistic. For example, a bootstrap estimate of the standard error of $\bar{X}$ can be obtained as follows:

- Generate independently of each other B bootstrap samples.
- Calculate the mean of the B bootstrap samples, i.e. calculate $\bar{x}_1^*, \bar{x}_2^*, \ldots, \bar{x}_B^*$.
- Calculate $\bar{\bar{x}} = \frac{1}{B}\sum_{i=1}^{B} \bar{x}_i^*$.
- Calculate $\hat{se}(b) = \sqrt{\frac{1}{B-1}\sum_{i=1}^{B} (\bar{x}_i^* - \bar{\bar{x}})^2}$.

(a) Generate a random sample of size 25 from a $normal(100; 255)$ distribution.

(b) Use R to obtain graphical representations and statistics of the characteristics of the sample.

(c) Program the necessary instructions in R to obtain bootstrap estimates of the standard error of the sample mean as well as the sample median. Use 50, 100, 500 and 1000 for B (the number of bootstrap repetitions). How do your answers compare with what is theoretically expected?

(d) Program the necessary R instructions to obtain graphical representations of the bootstrap distribution in (c).

2.10.3 Exercise

Generate a random sample of size 50 from a multivariate normal distribution with mean vector $(118, 396, 118, 400)$ and a covariance matrix so that the variances of the variables are given by 778, 1810, 580 and 2535 respectively. Variables 1 and 2 have a covariance of -642.5 and variables 3 and 4 have a covariance of -670. The other variables are uncorrelated. Store the sample as a matrix object and then program the necessary R instructions to calculate the sample covariance matrix and sample mean vector.

2.10.4 Exercise

Execute the instruction `set.seed(101023)`.

Next, obtain 400 random $normal(0; 1)$ values and arrange them in a matrix with 20 rows and 20 columns. Finally, write an R function to calculate and return (i) the sum of all the elements in the matrix, (ii) the eigenvalues of the matrix, (iii) the inverse of the matrix as well as (iv) the rank of the matrix making use of the eigenvalues. *Hint*: Read the help of the functions `eigen()` and `solve()`.)

Chapter 3

R operators and functions

After completing Chapters 1 and 2 it is assumed that the following are now familiar:

- How to communicate with R;
- How to manage workspaces;
- How to perform simple tasks using R.

In this chapter we take a closer look at the behaviour of some of the most common

- R operators
- R functions.

3.1 Arithmetic operators

(a) Study the use of the operators in Table 3.1.

Table 3.1: Arithmetic operators.

Operator	*Function*	*Operator*	*Function*
`+`	Addition	`^`	Exponentiation
`-`	Subtraction	`%/%`	Integer divide
`*`	Multiplication	`%%`	Modulus
`/`	Division	`:`	Sequence
`%*%`	Matrix multiplication	`-`	Unary minus

Note that the arithmetic operators are also functions. That this is so follows by studying the following examples:

```
3+7
#> [1] 10
"+"(3,7)
#> [1] 10
17 %% 3
#> [1] 2
"%%"(17,3)
#> [1] 2
```

(b) Rules for operator expressions with vector arguments.

Study the results of the following R instructions.

```
cars [,2] * 12 * 25.4 / 1000
#>  [1]  0.6096  3.0480  1.2192  6.7056  4.8768  3.0480  5.4864
#>  [8]  7.9248 10.3632  5.1816  8.5344  4.2672  6.0960  7.3152
#> [15]  8.5344  7.9248 10.3632 10.3632 14.0208  7.9248 10.9728
#> [22] 18.2880 24.3840  6.0960  7.9248 16.4592  9.7536 12.1920
#> [29]  9.7536 12.1920 15.2400 12.8016 17.0688 23.1648 25.6032
#> [36] 10.9728 14.0208 20.7264  9.7536 14.6304 15.8496 17.0688
#> [43] 19.5072 20.1168 16.4592 21.3360 28.0416 28.3464 36.5760
#> [50] 25.9080
7%/%3
#> [1] 2
7%%3
#> [1] 1
matrix(1,nrow=4,ncol=4) * matrix(3,nrow=4,ncol=4)
#>      [,1] [,2] [,3] [,4]
#> [1,]    3    3    3    3
#> [2,]    3    3    3    3
#> [3,]    3    3    3    3
#> [4,]    3    3    3    3
matrix(1,nrow=4,ncol=4) %*% matrix(3,nrow=4,ncol=4)
#>      [,1] [,2] [,3] [,4]
#> [1,]   12   12   12   12
#> [2,]   12   12   12   12
#> [3,]   12   12   12   12
#> [4,]   12   12   12   12
```

Explain the following instructions and output from R:

```
1:12 + 1:3
#>  [1]  2  4  6  5  7  9  8 10 12 11 13 15
1:10 + 1:2
#>  [1]  2  4  4  6  6  8  8 10 10 12
1:10 + 1:3
#> Warning in 1:10 + 1:3: longer object length is not a
#> multiple of shorter object length
#>  [1]  2  4  6  5  7  9  8 10 12 11
```

In the above examples it is illustrated that R uses *vectorized arithmetic* i.e. it operates on vectors as wholes. Sometimes the *recycling principle* is applied with or without a warning. It is a good R programming habit to make use of vectorizing calculations where possible. The effect of the recycling principle must be kept in mind since it might lead to unwanted results.

(c) Missing values, infinity and "not a number".

A missing value in R is denoted by NA. The result of a computation involving NAs is always NA e.g.

```
mean(c(1,3,NA,12,5))
#> [1] NA
0/0
#> [1] NaN
5/0
#> [1] Inf
-5/0
#> [1] -Inf
5/(-0)
#> [1] -Inf
```

The result of a computation that cannot be represented as a number e.g. `0/0` is denoted by `NaN`. Note: some computational results are differently reported by R as the corresponding algebraic equivalents, `5/0` in R is given by `Inf` while algebraically it is undefined.

(d) Scientific notation

R uses decimal notation as well as scientific notation for arithmetic calculations. Scientific notation is not to be confused with *exp*().

```
60000000
#> [1] 6e+07
1/6000000
#> [1] 1.666667e-07
exp(15)
#> [1] 3269017
exp(-15)
#> [1] 3.059023e-07
```

(e) How are numbers represented in a computer's memory? What are the implications of this?

Computers use ON/OFF (or 1/0) switches for encoding information. A single switch is called a *bit* and a group of eight bits is called a *byte*. A single integer is represented exactly in a computer by a fixed number of bytes i.e. 32 or 64 bits. There are several schemes according to which integers are represented by bits in a computer. This representation in a computer takes place at a level where R has no control over it but R stores information about the computing environment in an object `.Machine`. The element `.Machine$integer.max` returns the largest integer that can be represented in the computer on which R is running e.g.

```
.Machine$integer.max
#> [1] 2147483647
```

Although the above method of representing integers by strings of bits provides a very efficient way of storing integers in a computer R usually treats integers similar to real numbers by using *floating point representation*. In binary floating point notation a number x is written as a sequence of zeros and ones (the *mantissa*) times two with an exponent say m: $x = b_0b_1b_2... \times 2^m$ where $b_0 = 1$ except when $x = 0$.

In practice there is only a limited number of b's available and the exponent is also limited therefore, in general, not all real numbers can be represented exactly in a computer – they can at most be approximated. The smallest number x such that $1 + x$ can be distinguished from 1 in a computer is called *machine epsilon*. In R this can be obtained from `.Machine$double.eps` e.g.

```
.Machine$double.eps
#> [1] 2.220446e-16
```

Although floating point representation allows computation with very small (in magnitude) and very large numbers the above limitations can lead to *underflow* or *overflow* which can have disastrous consequences in practice. Writing good code in R must take the above seriously into account.

3.2 Logical operators

Logical operators result in `TRUE`, `FALSE` or `NA`. Study the use of the logical operators in Table 3.2. *Warning*: While it is perfectly legitimate to write

```
x[x == -1] <- 0
x[x == 1] <- 0
```

it is incorrect to specify

```
x[x == NA] <- 0
x[x = = NaN] <- 0
```

The correct code in the latter case is

```
x[is.na(x)] <- 0
x[is.nan(x)] <- 0
```

What are the consequences of the above code? Also take note of the functions `any()` and `all()`. These two functions are useful when combining logical objects. Give the necessary instructions to carry out the following tasks:

(a) Check if any of the states in the `state.x77` data set have populations with an illiteracy rate that is not larger than 1.6 and a Murder rate of more than 10.0.
(b) Check if there is at least one state with income greater than \$5000 and life expectancy less than 70.0 years.
(c) Check if all states with an income of more than \$5000 has an illiteracy of below 2.0.

What is meant by a control logical operator?

Table 3.2: Logical operators.

Operator	*Function*
`>`	Greater than
`<`	Less than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equality
`&`	Elementwise and
`|`	Elementwise or
`&&`	Control and

Operator	*Function*
`||`	Control or
`!`	Unary not
`!=`	Not equal to

(d) Carry out the instructions:

```
mata <- matrix(1:4, ncol = 2)
matb <- matrix(c(10, 20, 30, 40), ncol = 2)
mata
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
matb
#>      [,1] [,2]
#> [1,]   10   30
#> [2,]   20   40
mata>1 & matb>1
#>       [,1] [,2]
#> [1,] FALSE TRUE
#> [2,]  TRUE TRUE
mata>1 | matb>1
#>      [,1] [,2]
#> [1,] TRUE TRUE
#> [2,] TRUE TRUE
mata>1 && matb>1
#> Error in mata > 1 && matb > 1: 'length = 4' in coercion to 'logical(1)'
mata>1 || matb>1
#> Error in mata > 1 || matb > 1: 'length = 4' in coercion to 'logical(1)'
```

Comment on the above.

(e) What is the result of `sum(c(TRUE, !FALSE, FALSE, TRUE, TRUE))`?
(f) What is the result of `sum(c(TRUE, !FALSE, FALSE, NA, TRUE))` ?

Explain

3.3 The operators `<-`, `<<-` and `~`

Before considering the use of these operators answer the following:

(a) What will happen to an object `aa` in the working directory if within a function the following assignment is made `aa <- 20`?

(b) Now, study the help file of `<<-` and then answer (a) if the operator `<-` has been replaced with the operator `<<-`. *Warning*: use `<<-` very carefully.

(c) The tilde operator is used in modelling functions, e.g. `lm (length ~ age)`.

3.4 Operator precedence

Study the precedence rules as summarized in Table 3.4.1. The rules followed are shown in Table 3.3 from top to bottom and left to right. Note the use of

- parentheses `( )` for function arguments and changing precedence,
- braces `{ }` for demarcating blocks of instructions
- and brackets `[ ]` for subscripting.

The correct way of extracting the fifth element of a sequence like 1:20 is

```
(1:20)[5]
#> [1] 5
```

Table 3.3: Precedence rules.

Operator	*What it does*
`$`	List and dataframe subscripting
`[]`, `[[]]`	Vector and matrix subscripting; list subscripting
`^`	Exponentiation
`%*%`, `%/%`, `%%`	Matrix multiplication; integer divide; modulus
`*`, `/`	Multiplication and division
`+`, `-`	Addition and subtraction
`<`, `>`, `<=`, `>=`, `==`, `!=`	Logical comparisons
`!`	Unary not
`&`, `|`, `&&`, `||`	Logical and; logical or; control and; control or
`<-`, `<<-`	Assignment

Explain the result of the following R instructions:

```
20 / 4 * 12 ^2 - 6 + 1
#> [1] 715
(20 / 4) * (12 ^2) + (-6 + 14)
#> [1] 728
20 / 4 * 12 ^(2 - 6 + 14)
#> [1] 309586821120
20 / 4 * (12 ^2 - 6 + 14)
#> [1] 760
```

3.5 Some mathematical functions

3.5.1 General mathematical functions

`abs()`, `exp()`, `log(x, base = exp(1))`, `log10()`, `gamma()`, `sign()`, `sqrt()`

3.5.2 Trigonometric functions

See Table 3.4.

Table 3.4: Trigonometric functions.

Operator	*Function*		*Operator*
`cos()`	cosine	`acos()`	arc cosine
`sin()`	sine	`asin()`	arc sine
`tan()`	tangent	`atan()`	arc tangent
`cosh()`	hyperbolic cosine	`acosh()`	arc hyperbolic cosine
`sinh()`	hyperbolic sine	`asinh()`	arc hyperbolic sine
`tanh()`	hyperbolic tangent	`atanh()`	arc hyperbolic tangent

3.5.3 Complex numbers

`Arg()`, `Conj()`, `Mod()`, `Re()`, `Im()`

3.5.4 Functions for rounding and truncating

`round()`, `ceiling()`, `floor()`, `trunc()`

Study the help files of the above functions. Check all arguments.

3.5.5 Functions for matrices

Study Table 3.5 in detail.

Two other functions that play an important role in matrix calculations are the functions `rbind()` and `cbind()` for concatenating matrices row-wise or column-wise. Also revise the functions `matrix()`, `dim()`, `dimnames()`, `colnames()`, `rownames()` as well as `scan()` and `read.table()`.

Table 3.5: Functions for matrices.

Function	*What it does*
`chol()`	Cholesky decomposition
`crossprod()`	Matrix crossproduct
`diag()`	Create identity matrix, diagonal matrix or extract diagonal elements depending on its argument
`eigen()`	Finding eigenvectors and eigenvalues
`kronecker()`	Computing the kronecker product of two matrices
`outer()`	Outer product of two vectors
`scale()`	Centring and scaling a data matrix
`solve()`	Finding the inverse of a nonsingular matrix
`svd()`	Singular value decomposition of a rectangular matrix
`qr()`	QR orthogonalization
`t()`	Transpose of a matrix

(a) The function `chol()` performs a Cholesky decomposition of the square, symmetric, positive definite matrix $\mathbf{A} = \mathbf{U}'\mathbf{U}$ where $\mathbf{U}$ is an upper triangular matrix.

(b) The function `crossprod (A, B)` returns the matrix $\mathbf{A}'\mathbf{B}$.

(c) The function `diag(arg)` performs various actions depending on its argument: if `arg` is a positive integer `diag(arg)` returns an identity matrix of the given size; if `arg` is a vector `diag(arg)` returns a diagonal matrix with diagonal elements the respective elements of the given vector; if `arg` is a matrix then `diag(arg)` returns a vector containing the diagonal elements of the given matrix.

(d) What is the difference between `diag(A)` and `diag(diag(A))` where `A` is a square matrix?

(e) The function `eigen()` operates on a square matrix and returns a list with named elements `values` and `vectors` containing respectively, the eigenvalues and eigenvectors. Study the help file of `eigen()` carefully.

(f) The function `kronecker()` returns the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ of matrices $\mathbf{A}$ and $\mathbf{B}$.

(g) The function `outer (x, y, f)` operates on two vectors $x : n \times 1$ and $y : p \times 1$ to return a matrix of size $n \times p$ with ijth element the result of applying the function `f` on `x[i]` and `y[j]`. The default for `f` is `*`.

(h) The function `scale()` has three arguments: a matrix as first argument; a second argument `center` and a third argument `scale`. If `center = FALSE`, no centring of the columns of the matrix argument is performed, if set to `TRUE` (the default), the mean value of each column is subtracted from the respective columns, if given a vector of values these values are subtracted from the respective columns. If `scale = FALSE`, no scaling of the columns of the matrix argument is performed, if set to `TRUE` (the default) each column is divided by its standard deviation, if given a vector of values then each column is divided by the corresponding value.

(i) The function `solve (A, b)` is used for solving the equation $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, where $\mathbf{b}$ can be either a vector or a matrix with $\mathbf{A}$ being a square matrix. If argument `b` is missing it is taken to be the identity matrix so that the inverse of argument `A` is returned.

(j) The function `svd()` returns the singular value decomposition of its matrix argument $\mathbf{A} = \mathbf{UDV}'$. It returns a list with three components: `u` the orthogonal or orthonormal matrix $\mathbf{U}$; `d` the vector containing the ordered singular values of the rectangular matrix $\mathbf{A}$; `v` the orthogonal or orthonormal matrix $\mathbf{V}$.

(k) The function `qr()` performs a QR decomposition of any arbitrary matrix $\mathbf{M} = \mathbf{QR}$ with $\mathbf{Q}$ and orthogonal matrix and $\mathbf{R}$ an upper triangular matrix. Study the help file of `qr()` for full details and usages of the function. Note that the matrices $\mathbf{Q}$ and $\mathbf{R}$ can be obtained directly by calling `qr.Q(qr())` and `qr.R(qr())`, respectively.

(l) What is the meaning of each of the following instructions?

`rbind(a,b)`; `rbind(1,x)`; `rbind(a = 1:5,b = 10:14,c=20:24)`; `cbind( a= 1:5, b=10:14, c=20:24)`

(m) Write a function to calculate the determinant of a square matrix. Name this function `det.own()` in order to distinguish it from the built in R function `det()`.

(n) When the user is satisfied with a function, it is often necessary to have it available for all R projects. It is useful to assign all such functions to the same data base or folder. Use the function `assign (x, object, pos = , envir = )` to store the function `det.own()` in your own R functions folder. The argument `x` in `assign()` is a character string for assigning a name to the object. The function `remove (list of objects names, pos = , envir = )` can be used to remove objects from your own or any other database. *Hint*: First create a file and then use `attach()` to add it to the R search path.

```
save(file= " C:\\MyFunctions").
```

Study how `save()` works.

```
attach("C:\\MyFunctions", pos=2).
```

Study how `attach()` works.

```
assign("det.own", det.own, pos=2).
```

Study how `assign()` works.

```
save(list=objects(2), file = "C:\\MyFunctions")
```

Explain the use of the argument `list=objects(2)`. To summarize: The construction `NAME <- object` is a simple way to assign an object to a name. This form of assignment always takes place in the global environment (the workspace). Assignment can also be performed using the functions `save()` and `assign()` as illustrated above. The latter form of assignment is more complicated but the assignment is not restricted to the global environment.

(o) The result of the function `gamma(x)` is $(x-1)!$ if x is a non-negative whole number. Now write a function `fact()` to calculate $x!$. This function must make provision for 0! as well as for a negative number or a fraction that is read in by mistake. *Hint*: First study the usage of the if statement by requesting help `?Control`, recall Table 1.1. Store this function in your folder of R functions. How will you go about to make `fact()` and `det.own()` available for any R project?

(p) The function `lgamma(x)` returns the logarithms of $\Gamma(x)$. Write a function to calculate the value of $f(n) = \frac{\Gamma(\frac{n-1}{2})}{\Gamma(\frac{1}{2})\Gamma(\frac{n-2}{2})}$. Calculate the value of $f(n)$ for $n = -10, 10, 100, 500, 1000$.

3.5.6 Sorting functions

Note the use of the functions `sort()`, `order()` and `rank()`. First construct `MatX` using the functions `scan()` and `matrix()`. Explain in detail what `order()` does by sorting all the columns of `MatX` according to the values in the first column of the matrix.

$$MatX = \begin{bmatrix} 4 & 80 & 12 \\ 5 & 70 & 70 \\ 6 & 30 & 19 \\ 2 & 40 & 80 \\ 4 & 90 & 40 \\ 1 & 60 & 50 \\ 7 & 10 & 20 \\ 3 & 30 & 200 \end{bmatrix}$$

3.5.7 Some functions for data manipulation

Study the functions in Table 3.6.

Table 3.6: Functions for data manipulation.

Function	*What it does*
`append()`	Combine vectors; more flexibility than `c()`
`c()`	Create vectors
`duplicated()`	Extract duplicated values
`match()`	Match values in pairs of vectors
`pmatch()`	Partial matching
`replace()`	Replace specified values in vectors
`unique()`	Extract unique values

(a) Insert the vector (101, 102, 103, 104, 105) into the vector (10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20) after its fifth element by utilising the argument `after` of the function `append()`.

(b) The function `replace()` requires three arguments `x`, `list` and `vals`. The values in `x` with indices given in `list` is replaced by the successive values in `vals` making use of the recycling principle if needed. Explain this by replacing in the vector (10, 2, 7, 20, 5, 8, 9, 20, 9, 1,1 15), the values 10, 20 and 15 with zeros.

(c) Find the unique values in the vector (10, 2, 7, 20, 5, 8, 9, 20, 9, 1, 15).

(d) Find the duplicated values in the vector (10, 2, 7, 20, 5, 8, 9, 20, 9, 1, 15, 20, 20, 15).

(e) Explain the usage of `match()` by considering the difference between

```
match (c(10,2,7,20,5,8,9,20,9,1,15), c(10,20,15))
#>  [1]  1 NA NA  2 NA NA NA  2 NA NA  3
match (c(10,20,15), c(10,2,7,20,5,8,9,20,9,1,15))
#> [1]  1  4 11
```

(f) Illustrate the difference between `match()` and `pmatch()` by considering the names of the days of the week.

3.5.8 Basic statistical functions

Study the functions in detail in Table 3.7.

Table 3.7: Basic statistical functions.

Function	*What it does*	*Comments*
`cor()`	Correlation	One or two arguments
`cumsum()`	Cumulative sum of elements of a vector	
`mean()`	Arithmetic mean	Optional argument `trim =`
`median()`	Median	Accepts variable number of arguments
`min()`	Minimum value	Accepts variable number of arguments
`max()`	Maximum value	Accepts variable number of arguments
`prod()`	Product of elements of a vector	Accepts variable number of arguments
`cumprod()`	Cumulative product of elements of a vector	
`quantile()`	Returns specified quantiles	
`range()`	Minimum and maximum of a vector	Accepts variable number of arguments
`sample()`	Random sample	With or without replacement
`sum()`	Arithmetic sum	Also used for counting
`var()`	Variance and covariance; uses n-1 as denominator	Accepts vectors or matrices
`sd()`	Standard deviation; uses n-1 as denominator	Accept a vector as argument

Note also the functions `pmax()` and `pmin()`.

(a) Find the average Life Expectancy of the states in the `state.x77` data set.
(b) Find the 5% trimmed mean for Illiteracy of the states in the `state.x77` data set.
(c) Find the correlation between the Illiteracy and the Income of the states in the `state.x77` data set.
(d) Find the covariance matrix of all the variables in the `state.x77` data set.
(e) Find the range for Murder in the `state.x77` data set.
(f) Obtain the details of a random sample of 10 states in the `state.x77` data set.
(g) Obtain two independent random permutations of the numbers $1, 2, \ldots, 10$.
(h) Write a function for computing the coefficient of kurtosis for a random sample. Test your function on the Frost variable in the `state.x77` data set.
(i) Write a function for computing the coefficient of skewness for a random sample. Test your function on the Murder variable in the `state.x77` data set.

(j) Write a function to compute the harmonic mean of a numeric vector. Test your function on the Life Expectancy of the states in the `state.x77` data set. Compare your answer to your answer in (a).

3.5.9 Probability distributions in R

First, execute the R-instruction

```
help.search("distribution")
```

to obtain a list of available statistical distributions in R. Each distribution has an identifying name preceded by one of the letters *d*, *p*, *q* or *r*. In the case of an F-distribution, for example, the identifier is just the letter `f` and for a normal distribution the identifier is `norm`. Preceding the distribution's identifier by one of the letters `d`, `p`, `q` or `r` returns a density value, a probability, a quantile or a random sample for the specified distribution (probability density function or probability mass function). See Figure 3.1 for an explanation.

3.5.10 Functions for categorical variables

Apart from being *numeric* or *logical*, data in R can also be *categorical* (*factor* in R) or character strings. Study in detail the functions operating on factor data in Table 3.8.

(a) Use `cut()` to create an object `areagrp` to divide the `state.x77` data set into three groups representing the states with area within the intervals $(0, 10000]$,$(10000, 100000]$ and $(100000, Inf]$, respectively. *Hint*: First study the arguments of `cut()`.

(b) Repeat (a) with argument `labels = ??` to specify each state as being *Small*, *Medium* or *Large* with respect to its area.

(c) Use `unclass()` to obtain the numeric codes associated with each level of `areagrp`.

(d) Repeat (a) to obtain `areagrp2` containing five equally spaced categories.

(e) Repeat (a) to obtain `areagrp3` containing five groups with each containing 20% of the data.

(f) Use `cut()` to create an object `illitgrp` to divide the `state.x77` data set into five groups representing the states with illiteracy within the interval $[0, 0.50)$, $[0.50, 1.00)$, $[1.00, 1.50)$, $[1.50, 2.00)$ and $[2.00, 5.00)$, respectively.

(g) Obtain a two-way table of the `state.x77` data set according to `areagrp` and `illitgrp`.

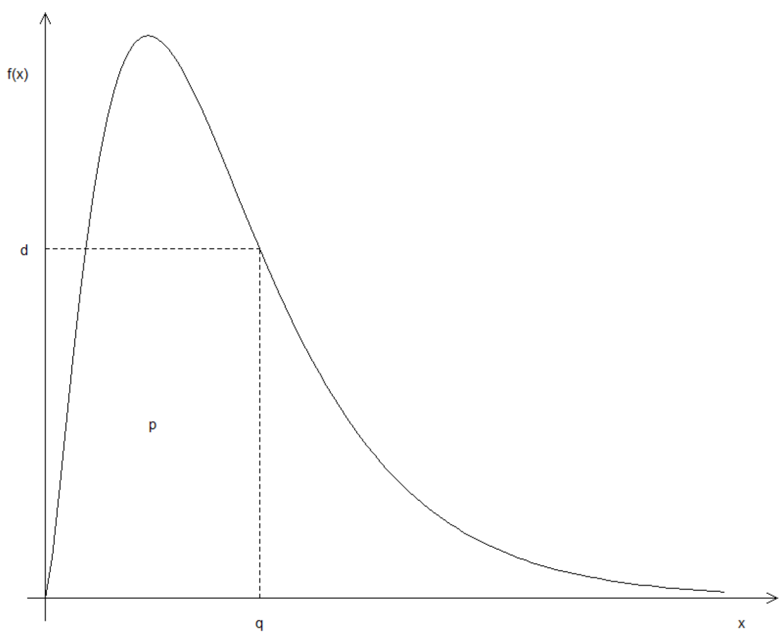


Figure 3.1: Meaning of the letters d, p and q when preceding an R distribution identifier.

Table 3.8: Basic functions for categorical variables.

Function	*What it does*
`cut()`	Creates categories out of a continuous variable
`factor()`	Encodes a vector as a ***nominal*** categorical variable
`ordered()`	Encodes a vector as a ***ordinal*** categorical variable when argument ordered is set to TRUE
`levels()`	Displays or sets the levels of a factor variable
`pretty()`	Creates convenient break points for a categorical variable
`split()`	Breaks up an array according to the value of a categorical variable
`table()`	Counts the number of observations cross-classified by categories
`unclass()`	Returns the numeric codes for representing the levels of a factor variable

3.5.11 Functions for character manipulation

Study the functions in Table 3.9 in detail.

Table 3.9: Basic functions for character manipulation.

Function	*What it does*
`abbreviate()`	Generates abbreviations of character values
`cat()`	Display,messages and/or values on screen or send to file
`grep()`	Search for patterns in characters
`nchar()`	Number of characters in a string
`paste()`	Combine values into character strings
`strsplit()`	Split the elements of a character vector × into substrings
`substring()`	Extracts parts of character strings

(a) What is the returned value of `grep ("ia", state.name)`?

(b) Discuss the usage of `grep ("ia", state.name)`.

(c) Discuss the output of `objects (pos = grep("stats", search()))`.

(d) Use `paste()` to create variable names: var1, var2, ..., var100.

(e) Repeat (d) to create variable names: var_1, var_2, ..., var_100.

(f) Discuss the output of:

```
substring (paste (letters, collapse = ""),
             1:nchar (paste (letters, collapse="")),
             1:nchar (paste (letters, collapse="")))
```

(g) From the Help menu, select Manuals (in PDF) and open the Introduction to R document. Obtain a copy of the first two paragraphs of the Preface on page 1 of this book in the R commands window. Use this copy to calculate the number of words as well as the total number of characters (including spaces between words) in the passage.

We are going to use several of the functions in Table 3.9 to perform this task in steps. Proceed as follows in R after copying the relevant passage to the clipboard:

```
TextPar <- scan(file = "clipboard", what = "")
```

To obtain a vector containing each of the words as a separate element.

```
TextPar <- paste (TextPar, collapse = " ")
```

To convert `TextPar` into a vector containing one element consisting of all the words concatenated and separated by spaces into a single character string. Add the correct line breaks ("\n") in `TextPar` using e.g. `fix()`.

```
TextPar <- strsplit(x = TextPar, split = '\n')
```

```
mode(TextPar)
[1] "list"

mode(unlist(TextPar))
[1] "character"
```

```
TextPar <- unlist(TextPar)
```

To change `TextPar` into a character vector.

```
nchar(TextPar)
length(TextPar)
```

3.6 Differentiation and integration

3.6.1 Symbolic differentiation

Study the help files of `D()` and `deriv()`.

3.6.2 Integration

Study the help file of `integrate()`.

3.6.3 Exercise

(1) It is known from elementary statistics that approximately 68% of data from a normal distribution with a mean of zero and a standard deviation of unity will have an absolute value less than unity. Use the `sum()` and `rnorm()` functions to find the proportion of n random $normal(0, 1)$ variables whose absolute value is less than 1.0. Repeat with different values for n to investigate how widely the results vary.

(2) Define: conditional inverse and generalized (Moore-Penrose) inverse for matrix $\mathbf{X} : p \times q$ and make provision for $p = q$, $p > q$ and $p < q$. First, show how the svd of $\mathbf{X}$ can be used to obtain a conditional inverse, $\mathbf{X}^c$ for $\mathbf{X}$. Now use the above information to write an R function for calculating $\mathbf{X}^c$ for any given $\mathbf{X}$. The function must provide a test to check if the calculated conditional inverse is indeed a conditional inverse. Illustrate the usage of your function.

(3) Give the necessary instructions to:

(i) read into R an external text data file consisting of 10 sample observations with each consisting of one character variable and two numerical variables.
(ii) read into R a large external text data file consisting of 50 numerical variables but unknown number of records. Each record in this data file takes up 5 lines. The variables in the R object must have the names X1, …, X50.

(4) Discuss the meaning of the following R instructions:

(i) `y <- x[!is.na(x)]`
(ii) `z <- (x + y)[!is.na(x) & x >0]`
(iii) `a <- x[-(1:5)]`
(iv) `x[is.na(x)] <- 0`

Chapter 4

Introducing traditional R graphics

A basic knowledge of R graphics is needed before directing attention to the art of writing programs (functions) in R. Therefore, in this chapter a brief overview is given of the basics of traditional R graphics. In a later chapter, after studying the principles of R programming, a second round of R graphics will follow.

4.1 General

Study the graphical parameters by requesting

```
?par
```

In Figure 4.1 the main components of a graph window are illustrated. Study this figure in detail. The *Plot Region* together with the *Margins* is called the *Figure Region*.

(a) What is the difference between high-level and low-level plotting instructions?

(b) Take note especially how the functions `windows()`, `win.graph()` or `x11()` are used as well as the different options available for these functions.

(c) The instruction `dev.new()` allows opening a new graph window in a platform-independent way.

(d) In this chapter some high-level plotting instructions are studied. Each of these instructions results in a (new) graph window with a complete graph drawn. The command `graphics.off()` deletes all open graphic devices.

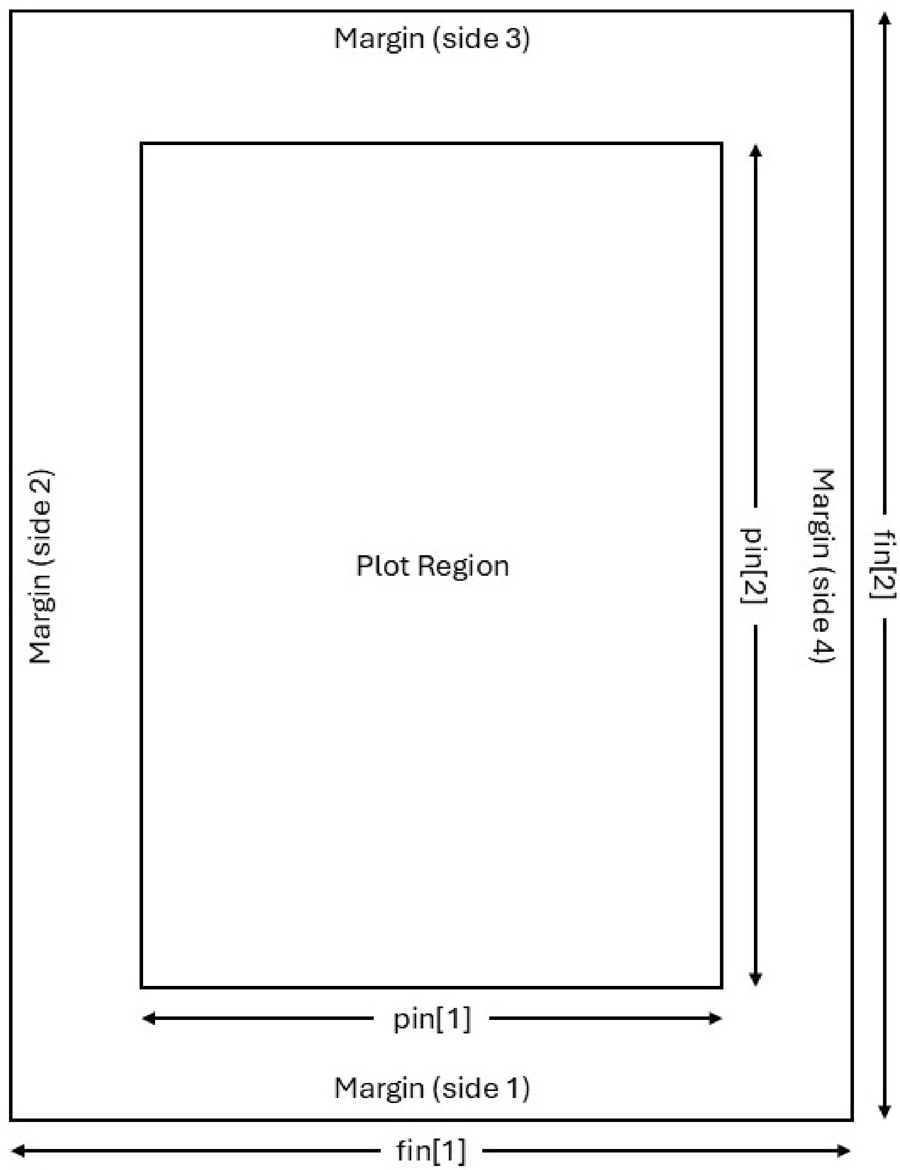


Figure 4.1: The main components of a graph window and the parameters for controlling their sizes. The parameter mai is a numerical vector of the form c(bottom, left, top, right) specifying the margins in inches while the parameter mar has a similar form specifying the respective margins as the number of lines. The default of mar is c(5, 4, 4, 2) + 0.1.

(e) Study the use of `par()`, `par(mfrow =)` and `par(mfcol =)`. Study the use of `par(new = TRUE)` to plot more than one figure on the same set of axes.

(f) Study how the functions `graphics.off()` and `dev.off()` work.

4.2 High-level plotting instructions

(a) Construct a barplot of the illiteracy of the states according to the `areagrp` (as defined in section 3.5.10) in the `state.x77` dataframe. *Hint*: The function `tapply()` operates on a vector given as its first argument. Its second argument groups the first argument into groups so that the function given in its third argument can be applied to each of these groups. Study the following command:

```
barplot (tapply (state.x77[, "Illiteracy"], areagrp, mean),
         names=levels(areagrp), ylab = "Illiteracy", xlab = "Area of State",
         main = "Barplot of Mean Illiteracy")
```

(b) Construct, for the `state.x77` data set, box plots of illiteracy broken down by the income of the states. First use `cut()` to form three categories of state income:

```
state.income <- cut (state.x77[ , "Income"], c(0, 4000, 5000, Inf),
                     labels=c("$4000 or less", "$4001-$5000", "more than $5001"))
```

Then use `boxplot()` together with `split()` to produce the desired graph:

```
boxplot (split (state.x77[ , "Income"], state.income))
```

Add labels for the axes as well as a title for the figure.

(c) Repeat the previous example but use argument `notch = TRUE`.

(d) Attach the package `akima`. What is the usage of the function `interp()`? Discuss by constructing the following contour plot:

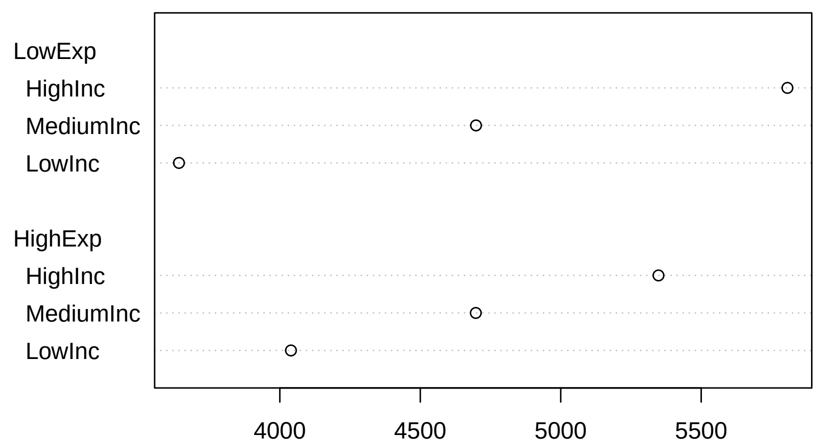
```
contour (interp (state.center$x, state.center$y,  state.x77[,"Frost"]))
```

(e) What is a *coplot*? Discuss after giving the following instruction and referring to the role of the tilde (~) operator.

```
coplot (state.x77[,"Illiteracy"] ~ state.x77[,"Area"] | state.x77[,"Income"])
```

(f) A *dotchart* is constructed with function `dotchart()`. First some preparations are necessary:

```
incgroup <- cut(state.x77[,"Income"],  3,
                labels = c("LowInc", "MediumInc", "HighInc"))
lifgroup <- cut(state.x77[,"Life Exp"], 2,
                labels = c("LowExp", "HighExp"))
table.out <- tapply(state.x77 [,"Income"], list(lifgroup,incgroup), mean)
table.out
#>             LowInc MediumInc HighInc
#> LowExp  3640.917  4698.417    5807
#> HighExp 4039.600  4697.667    5348
dotchart (table.out,
          levels (factor (col (table.out),
                      labels = levels (incgroup)))[col(table.out)],
          factor(row(table.out), labels = levels(lifgroup)))
```



Complete the graph by adding a label to the x-axis and a heading for the graph.

(g) Use function `faces()` available in package `aplpack` to construct Chernoff faces for the Western states in the data set `state.x77`. *Hint*: The Western states appear in rows 3, 5, 12, 26, 28, 37, 44, 47 and 50. Explain what is represented by each of the facial features. First set argument `face.type = 0` and then `face.type = 1`.

(h) Obtain a histogram of the life expectancy in the states of `state.x77`.

(i) Execute the command

```
pairs (state.x77)
```

Interpret the graph.

(j) Three-dimensional graphs are constructed with function `persp()`.

```
pts <- seq(from = -pi, to = pi, len = 20)
z <- outer(X = pts, Y = pts, function(x,y) sin(x)*cos(y))
persp(x = pts, y = pts, z, theta = 10, phi = 60, ticktype = 'detailed')
```

Discuss the meaning of each of the above instructions. Experiment with different values for arguments `theta` and `phi`.

(k) Obtain a pie chart of the object `areagrp` defined in section 3.5.10. *Hint*: function `table()` may be useful here.

(l) A cluster plot (dendrogram) can be constructed with function `plclust()` as follows:

```
west.rows <- c(3, 5, 12, 26, 28, 37, 44, 47, 50)
distmat.west <- dist (scale (state.x77[west.rows,]))
plot(hclust(distmat.west), labels = rownames(state.x77)[west.rows])
```

Interpret the above instructions and the resulting plot.

(m) Use the function `plot()` to plot $sin(\theta)$ as θ varies from $-\pi$ to π.

(n) Could you explain the different graphs resulting from the two calls in (l) and (m) to the `plot()` function above?

(o) Obtain the empirical distribution function of variable `Life Exp` in the `state.x77` data set by using the functions `cut()`, `ecdf()` and `plot()`.

(p) Check the normality of variable `Income` in the `state.x77` data set by using function `qqnorm()`.

(q) Obtain a `qqplot` of the income of small states versus the income of large states in the data set `state.x77` where small and large are defined as below or above the median income, respectively.

```
state.size <- cut (state.x77[,"Area"],
                    c(0, median (state.x77[,"Area"]), max (state.x77[,"Area"])))
state.income <- split (state.x77[,"Income"], state.size)
qqplot(state.income[[1]], state.income[[2]], xlab="Income for small states",
       ylab="income for large states")
```

(r) Use function `ts.plot()` to construct a time series plot of the sunspots data set.

4.3 Interactive communication with graphs

(a) Study the help files of the functions `text()`, `identify()` and `locator()`.

(b) Illustrate the usage of `identify()` on a scatterplot of variables `Illiteracy` and `Life Exp` in the `state.x77` data set:

```
plot (x = state.x77[,'Life Exp'], y = state.x77[,'Income'])
```

To create the scatterplot, then call

```
identify (x = state.x77[,'Life Exp'], y = state.x77[,'Income'],
          seq (along = rownames(state.x77)), n = 5)
```

Notice the change in the cursor; the cursor changes to a cross when moved over the graph. Hover the cursor over a point to identify and click left mouse button. Repeat $n = 5$ times. Explain the result. Next, create the scatterplot once more and then call

```
identify (x = state.x77[,'Life Exp'],  y = state.x77[,'Income'],
          labels = rownames(state.x77)[seq (along =
                                        rownames(state.x77))] , n = 5)
```

Explain what has happened.

(c) Illustrate the usage of `locator()` by:

(i) Joining 5 user defined points on a graph interactively with straight lines.

```
plot (x = state.x77[,'Life Exp'], y = state.x77[,'Income'])
locator(5, type = "l")
```

Use mouse and select the five points on the graph. What happened on the graph? What happened in the commands window?

(ii) Writing text interactively at a specified position on an existing graph.

```
plot (x = state.x77[,'Life Exp'], y = state.x77[,'Income'])
text (locator (n = 1, type = "n"), label = "State with the highest income")
```

4.4 3D graphics: package rgl

Write and execute the following function.

```
rgl.example <- function (size = 0.1, col = "green", alpha.3d = 0.6)
{ require(rgl)
  datmat <- matrix (rnorm (30), ncol = 3)
  open3d()
  spheres3d (datmat,radius = size, color = col, alpha = alpha.3d)
  axes3d(col = "black")
  device.ID <- rgl.cur()
  answer <- readline ("Save 3D graph as a .png file? Y/N\n")
  while (!(answer == "Y" | answer == "y" | answer == "N" | answer == "n"))
    answer <- readline("Save 3D graph as a .png file? Y/N\n")
  if (answer == "Y" | answer == "y")
    repeat
    { file.name <- readline ("Provide file name including full
                              path NOT in quotes and SINGLE
                              back slashes!\n")
      file.name <- paste (file.name, ".png", sep = "")
      snapshot3d (file = file.name)
      rgl.set (device.ID)
      answer2 <- readline("Save another 3D graph as a .png file? Y/N \n")
      if (answer2 == "Y" | answer2 == "y") next else break
    }
  else rgl.set (device.ID)
}
```

Study the above code and constructions in detail.

4.5 Exercise

1. Obtain a graph of a $normal(100, 25)$ probability density function (p.d.f.).
2. Plot on the same set of axes

   (i) a central $beta(9, 5)$ p.d.f.;

(ii) a non-central $beta(95)$ p.d.f. with non-centrality parameter = 15 and
(iii) a non-central $beta(9, 5)$ p.d.f. with non-centrality parameter = 40.

Add a suitable legend to the plot.

3. Use `persp()` to obtain a graph of any user specified bivariate function. The challenge is that the function specification must appear as the main title of the graph. In order to address this problem we need information about the arguments of `persp()`:

```
args (persp)
#> function (x, ...)
#> NULL
```

This is not very helpful so we try

```
methods (persp)
#> [1] persp.default*
#> see '?methods' for accessing help and source code
args (persp.default)
#> Error: object 'persp.default' not found
```

The reason for this error message follows from the above as that `persp.default` is not visible. The immediate visibility of a function is regulated by a package builder through the package's namespace mechanism. Only object names that are exported are immediately visible; object names that are not exported are marked with an asterisk and are not visible. The functions `argsAnywhere()` and `getAnywhere()` are available to get information on asterisked object names:

```
argsAnywhere (persp.default)
#> function (x = seq(0, 1, length.out = nrow(z)), y = seq(0, 1,
#>     length.out = ncol(z)), z, xlim = range(x), ylim = range(y),
#>     zlim = range(z, na.rm = TRUE), xlab = NULL, ylab = NULL,
#>     zlab = NULL, main = NULL, sub = NULL, theta = 0, phi = 15,
#>     r = sqrt(3), d = 1, scale = TRUE, expand = 1, col = "white",
#>     border = NULL, ltheta = -135, lphi = 0, shade = NA, box = TRUE,
#>     axes = TRUE, nticks = 5, ticktype = "simple", ...)
#> NULL
```

We notice that we can make use of the argument main in a call to `persp()` to provide our perspective plot with a title. However, main accepts only character strings and not mathematical expressions. Furthermore, we have seen in the `persp()` example in section 4.2 that the values for the argument `z` are conveniently found by a call to `outer()` using its argument `FUN`. However `FUN` requires a function. So we need the means to convert expressions into character strings and vice versa to convert character strings into expressions.

The following pairs of functions allow these conversions to be made:

Character strings (" ") → expressions: `parse()` and `eval()`

Expressions (unquoted) → character strings (" "): `deparse()` and `substitute()`

```
pts <- seq (from = -3, to = 3, len = 50)
fun1 <- "2 * pi * exp(-(x^2 + y^2)/2)"
fun2 <- parse (text = paste ("function(x,y)", fun1))
```

Explain carefully what `parse()` is doing.

```
zz <- outer (pts, pts, eval(fun2))
```

Explain carefully what `eval()` is doing.

```
persp (x = pts, y = pts, z = zz, theta = 0, phi = 15, ticktype = "detailed",
       main = paste("Persp plot of `"fun2,"`",sep=""))
```

Explain carefully the role of `paste()`.

4. Use the `volcano` data to:

   (i) Obtain a perspective plot using `persp()`.

   (ii) Obtain an RGL plot of the `volcano` data.

Chapter 5

Subscripting

Vectorized arithmetic and subscripting are two cornerstones of R programming. Review section 4.2 for several examples where subscripting has been used. In this chapter subscripting is studied in detail. Specifically, the following two related topics are studied:

- Extracting parts of an object by using *subscripting*.
- The combination and rearranging of data within data structures like matrices, dataframes and lists.

5.1 Subscripting with vectors

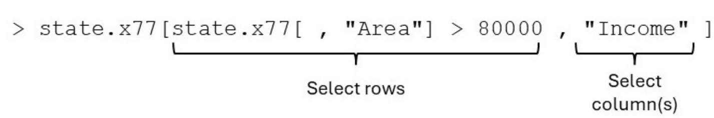
The different types of subscripting with vectors are summarized in Table 5.1:

Table 5.1: Different types of subscripting vectors.

Type	*Effect*	*Example*
empty	Extract all values	`x[ ]`
integer, positive	Extract all values specified by the subscript	`x[c(2:5,8,12) ]`
integer, negative	Extract all values except those specified by the subscript	`x[-c(2:5,8,12) ]`
logical	Extract those values for which subscript is TRUE	`x[x > 5 ]`
character	Extract those values whose names attributes correspond to those specified by the subscript	`x[c("a","d") ]`

Logical subscripting provides a very powerful operation in R. A logical subscript is a vector of `TRUE`s and `FALSE`s that must be of the same length as the object being subscripted e.g.

```
state.x77[ , "Area"] > 80000
#>        Alabama         Alaska        Arizona       Arkansas
#>          FALSE           TRUE           TRUE          FALSE
#>     California       Colorado    Connecticut       Delaware
#>           TRUE           TRUE          FALSE          FALSE
#>        Florida        Georgia         Hawaii          Idaho
#>          FALSE          FALSE          FALSE           TRUE
#>       Illinois        Indiana           Iowa         Kansas
#>          FALSE          FALSE          FALSE           TRUE
#>       Kentucky      Louisiana          Maine       Maryland
#>          FALSE          FALSE          FALSE          FALSE
#>  Massachusetts       Michigan      Minnesota    Mississippi
#>          FALSE          FALSE          FALSE          FALSE
#>       Missouri        Montana       Nebraska         Nevada
#>          FALSE           TRUE          FALSE           TRUE
#>  New Hampshire     New Jersey     New Mexico       New York
#>          FALSE          FALSE           TRUE          FALSE
#> North Carolina   North Dakota           Ohio       Oklahoma
#>          FALSE          FALSE          FALSE          FALSE
#>         Oregon   Pennsylvania   Rhode Island South Carolina
#>           TRUE          FALSE          FALSE          FALSE
#>   South Dakota      Tennessee          Texas           Utah
#>          FALSE          FALSE           TRUE           TRUE
#>        Vermont       Virginia     Washington  West Virginia
#>          FALSE          FALSE          FALSE          FALSE
#>      Wisconsin        Wyoming
#>          FALSE           TRUE
```




```
x <- c(10, 15, 12, NA, 18, 20)
is.na (x)
#> [1] FALSE FALSE FALSE  TRUE FALSE FALSE
x[is.na (x)]
#> [1] NA
x[!is.na (x)]
#> [1] 10 15 12 18 20
mean (x)
```

```
#> [1] NA
mean (x[!is.na (x)])
#> [1] 15
mean (na.omit (x))
#> [1] 15
```

Logical subscripting allows finding the indices of those elements in a vector that meet a certain condition e.g.

```
(1:length (rownames (state.x77)))[state.x77[ ,"Income"] > 5000]
#> [1]  2  5  7 13 20 28 30 34
```

and to find the corresponding names of the states

```
rownames(state.x77)[
  (1:length (rownames(state.x77)))[state.x77[ ,"Income"] > 5000]]
#> [1] "Alaska"       "California"   "Connecticut"
#> [4] "Illinois"     "Maryland"     "Nevada"
#> [7] "New Jersey"   "North Dakota"
```

In addition to extracting elements, the above subscripting operations can also be used to modify selected elements of a vector e.g. changing NA-values to zero:

```
x
#> [1] 10 15 12 NA 18 20
x[is.na (x)] <- 0
x
#> [1] 10 15 12  0 18 20
```

When the right-hand side of the assignment above is a scalar value, each of the selected values will be changed to the specified scalar value; if the right-hand side is a vector, the selecting values will be changed in order, *recycling* the values if more values were selected on the left-hand side than were available on the right-hand side.

5.2 Subscripting with matrices

Element and submatrix extraction of matrices are discussed below.

(a) Revise the use of `matrix()`, `names()`, `dim()` and `dimnames()`.

(b) A matrix in R is an *array* with two indices. Arrays of order two and higher can be constructed with the function `dim()` or `array()`.

Let, for example, **a** be a vector consisting of 150 elements. The instruction

```
dim(a) <- c(3, 5, 10)
```

or the instruction

```
a <- array (a, dim = c(3, 5, 10))
```

constructs a $3 \times 5 \times 10$ array.

- Matrices can therefore be formed as above, but the function `matrix()` is usually easier to use.
- The elements of a p-dimensional array can also be extracted using the one-index or two-index method as described below.

(c) The subscripting methods described in section 5.1 can also be applied to both the first or second dimension of a matrix where the first dimension refers to the rows and the second dimension to the columns of the matrix.

(d) Note that the elements of a matrix can be referred to by the two-index method above or by a one index method. When the one index method is used it is assumed that the matrix has first been strung out *column*-wise into a vector.

```
testmat.a <- matrix (c (17, 40, 20, 34, 21, 12, 14, 57,
                        78, 37, 29, 64), nrow = 4)
testmat.a
#>      [,1] [,2] [,3]
#> [1,]   17   21   78
#> [2,]   40   12   37
#> [3,]   20   14   29
#> [4,]   34   57   64
testmat.b <- matrix (c (17, 40, 20, 34, 21, 12, 14, 57,
                        78, 37, 29, 64), nrow = 4, byrow = TRUE)
testmat.b
#>      [,1] [,2] [,3]
#> [1,]   17   40   20
#> [2,]   34   21   12
#> [3,]   14   57   78
#> [4,]   37   29   64
```

Comment on the difference between `testmat.a` and `testmat.b`.

```
testmat.a[2,3]    # Two index matrix reference
#> [1] 37
testmat.a[10]    # One index matrix reference
#> [1] 37
```

(e) Write a function to convert a one-index to a two-index matrix reference. Give an example of the usage of your function.

(f) Write a function to convert a two-index to a one-index matrix reference. Give an example of the usage of your function.

(g) Consider the following example to form submatrices:

```
testmat <- matrix(1:50, nrow = 10, byrow = TRUE)
testmat[1:2, c (3, 5)]
#>      [,1] [,2]
#> [1,]    3    5
#> [2,]    8   10
testmat[1:2, 3]
#> [1] 3 8
testmat[1:2, 3, drop=FALSE]
#>      [,1]
#> [1,]    3
#> [2,]    8
```

(h) Notice the difference between `testmat [1:2, 3]` and `testmat [1:2, 3, drop = FALSE]`. The first command results in the output to be given in the form of a vector while the optional `drop = FALSE` in the second command retains the matrix structure of the output. This distinction can have serious consequences when a procedure expects a matrix argument and not a vector.

(i) Notice also that the output of both `testmat[1:2,3]` and `testmat[3, 1:2]` has a similar form: R makes no distinction between column vectors and row vectors; all one-dimensional collections of numbers are treated identically.

(j) Apart from using vectors as subscripts to a matrix, a matrix can also be used as a subscript to a matrix. There are two cases:

   (A) a numeric subscripting matrix and
   (B) a logical subscripting matrix.

Case A

Here the subscripting numeric matrix must have exactly two columns: the first provide row indices and the second column indices.

(i) If used on the right-hand side of an expression the result of a *case A* subscripting is a vector containing the values specified by the subscripting matrix.

(ii) If used on the left-hand side of an assignment a numeric matrix first selects those elements specified by its row and column indices; then these values are replaced one by one with the objects specified by the right-hand side of the assignment.

Here is an example of *case A* subscripting with the subscript matrix on the right-hand side of the assignment:

```
xmat <- matrix (1:25, nrow = 5)
xmat
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    6   11   16   21
#> [2,]    2    7   12   17   22
#> [3,]    3    8   13   18   23
#> [4,]    4    9   14   19   24
#> [5,]    5   10   15   20   25
superdiag.index <- matrix (c (1:4, 2:5), ncol = 2, byrow = FALSE)
superdiag.values <- xmat[superdiag.index]
superdiag.values
#> [1]  6 12 18 24
```

Case A subscripting with the numeric subscript matrix on the left-hand side of the assignment:

```
subscript.mat <- matrix (c(1:3, 1:3, rep(1,3), rep(2,3)), ncol=2)
subscript.mat
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    1
#> [3,]    3    1
#> [4,]    1    2
#> [5,]    2    2
#> [6,]    3    2
xx <- matrix(NA, nrow=3,ncol=2)
xx
#>      [,1] [,2]
```

```
#> [1,]   NA   NA
#> [2,]   NA   NA
#> [3,]   NA   NA
xx[subscript.mat] <- c(10,12,14,100,120,140)
xx
#>      [,1] [,2]
#> [1,]   10  100
#> [2,]   12  120
#> [3,]   14  140
```

Case B

The logical subscripting matrix must be in size exactly similar to that matrix it is subscripting and will select those values corresponding to a `TRUE` in the subscripting matrix.

Case B with logical subscripting matrix at right-hand side of assignment:

```
testmat
#>       [,1] [,2] [,3] [,4] [,5]
#>  [1,]    1    2    3    4    5
#>  [2,]    6    7    8    9   10
#>  [3,]   11   12   13   14   15
#>  [4,]   16   17   18   19   20
#>  [5,]   21   22   23   24   25
#>  [6,]   26   27   28   29   30
#>  [7,]   31   32   33   34   35
#>  [8,]   36   37   38   39   40
#>  [9,]   41   42   43   44   45
#> [10,]   46   47   48   49   50
aa <- testmat[testmat < 12]
aa
#>  [1]  1  6 11  2  7  3  8  4  9  5 10
```

Note that the selected elements are placed column-wise in a vector.

Case B with logical subscripting matrix at left-hand side of assignment:

```
testmat[testmat < 12] <- 12
testmat
#>       [,1] [,2] [,3] [,4] [,5]
#>  [1,]   12   12   12   12   12
#>  [2,]   12   12   12   12   12
#>  [3,]   12   12   13   14   15
#>  [4,]   16   17   18   19   20
```

```
#>  [5,]   21   22   23   24   25
#>  [6,]   26   27   28   29   30
#>  [7,]   31   32   33   34   35
#>  [8,]   36   37   38   39   40
#>  [9,]   41   42   43   44   45
#> [10,]   46   47   48   49   50
```

In order to restrict assignment to a subset of a matrix two sets of subscripts are needed. See example below:

```
testmat <- matrix(1:50, nrow=10, byrow=TRUE)
testmat[, c(1,3)][testmat[,c(1,3)] <12] <- 12
testmat
#>       [,1] [,2] [,3] [,4] [,5]
#>  [1,]   12    2   12    4    5
#>  [2,]   12    7   12    9   10
#>  [3,]   12   12   13   14   15
#>  [4,]   16   17   18   19   20
#>  [5,]   21   22   23   24   25
#>  [6,]   26   27   28   29   30
#>  [7,]   31   32   33   34   35
#>  [8,]   36   37   38   39   40
#>  [9,]   41   42   43   44   45
#> [10,]   46   47   48   49   50
```

Study the use of functions `row()` and `col()` in constructing logical matrices.

5.3 Extracting elements of lists

(a) Note the use of `list()` to collect objects into a list while elements are extracted with `$`

- the function `names()`,
- the single square brackets `[ ]` and
- the double square brackets `[[ ]]`.

(b) Study the following example carefully:

```
my.list <- list(el1 = 1:5,
                el2 = c("a", "b", "c"),
                el3 = matrix(1:16, ncol = 4),
                el4 = c(12, 17, 23, 9))
my.list
#> $el1
#> [1] 1 2 3 4 5
#>
#> $el2
#> [1] "a" "b" "c"
#>
#> $el3
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    5    9   13
#> [2,]    2    6   10   14
#> [3,]    3    7   11   15
#> [4,]    4    8   12   16
#>
#> $el4
#> [1] 12 17 23  9
my.list$el2
#> [1] "a" "b" "c"
mode (my.list$el2)
#> [1] "character"
my.list[el2]
#> Error: object 'el2' not found
my.list["el2"]
#> $el2
#> [1] "a" "b" "c"
mode (my.list["el2"])
#> [1] "list"
my.list[["el2"]]
#> [1] "a" "b" "c"
mode (my.list[["el2"]])
#> [1] "character"
```

Note: The above example shows that using the single pair of square brackets for subscripting a list always result in a list object to be returned. This is often the cause of an error message. See the example below.

```
my.list[1]
#> $el1
#> [1] 1 2 3 4 5
mode (my.list[1])
#> [1] "list"
```

```
my.list[[1]]
#> [1] 1 2 3 4 5
mode (my.list[[1]])
#> [1] "numeric"
my.list[3][2,4]
#> Error in my.list[3][2, 4]: incorrect number of dimensions
my.list[[3]][2,4]
#> [1] 14
my.list$el3[2,4]
#> [1] 14
mean (my.list[4])
#> Warning in mean.default(my.list[4]): argument is not
#> numeric or logical: returning NA
#> [1] NA
mean (my.list[[4]])
#> [1] 15.25
mean (my.list$el4)
#> [1] 15.25
```

Explain the differences and similarities between the symbols `[ ]`, `[[ ]]` and `$` when subscripting lists.

5.4 Extracting elements from dataframes

(a) Note the use of data.frame() for creating dataframes. A dataframe has a rectangular structure similar to a matrix but differs from a matrix in that its columns are not restricted to contain the same type of data. Each of its columns must contain the same sort of data but some columns can be numerical while others are factors for example.

(b) Explain the difference between the objects created by the following two instructions:

```
my.matrix <- matrix (c (17, 40, 20, 34, 21, 12, 14, 57,
                        78, 37, 29, 64), nrow = 4, ncol = 3)
my.dataframe <- data.frame ( c(17, 40, 20, 34, 21, 12, 14, 57,
                               78, 37, 29, 64), nrow = 4, ncol = 3)
```

(c) Note the following

```
class(my.matrix)
#> [1] "matrix" "array"
class(my.dataframe)
```

```
#> [1] "data.frame"
is.list(data.frame)
#> [1] FALSE
mode(my.matrix)
#> [1] "numeric"
mode(data.frame)
#> [1] "function"
```

(d) A sample of the behaviour of dataframes

```
my.dataframe.2 <- data.frame (C1 = c('a', 'b', 'c', 'd'),
                              C2 = c(5, 9, 23, 17),
                              C3 = c(TRUE, TRUE, FALSE, TRUE))
my.dataframe.2
#>   C1 C2    C3
#> 1  a  5  TRUE
#> 2  b  9  TRUE
#> 3  c 23 FALSE
#> 4  d 17  TRUE
my.dataframe.2[ ,1:2]
#>   C1 C2
#> 1  a  5
#> 2  b  9
#> 3  c 23
#> 4  d 17
```

Dataframe behaves like a matrix

```
my.dataframe.2$C1
#> [1] "a" "b" "c" "d"
```

Dataframe behaves like a list

```
as.matrix(my.dataframe.2)
#>      C1  C2   C3
#> [1,] "a" " 5" "TRUE"
#> [2,] "b" " 9" "TRUE"
#> [3,] "c" "23" "FALSE"
#> [4,] "d" "17" "TRUE"
```

Explain what has happened above.

(e) The above examples show that a dataframe can be considered as a cross between a matrix and a list. Therefore, subscripting of dataframes generally can be performed using the basic techniques available for matrices and lists.

(f) An alternative technique is to extract the elements of a list by using the functions `attach()` and `names()`. This technique is especially of importance in statistical modelling. What is a potential danger of this technique when attaching dataframes? This danger can be avoided by using `with()`. Is this also true when modelling is performed?

(g) Review section 2.3. Study the help file of the function `with()`. What important usage has `with()`?

5.5 Combining vectors, matrices, lists and dataframes

(a) What is the result of the command

```
my.list <- vector ("list", k)?
```

(b) Recall the function `c()` for creating vectors. When `c()` is used to combine a numeric vector and a character vector the result is a vector of mode "character". Similarly, using `c()` to combine a vector with a list results in a list.

(c) If `list()` is used to combine two or more vectors or lists the result is a list of all the objects.

(d) The function `unlist()` can be used to convert all the elements of a list into a single vector.

```
my.list
#> $el1
#> [1] 1 2 3 4 5
#>
#> $el2
#> [1] "a" "b" "c"
#>
#> $el3
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    5    9   13
#> [2,]    2    6   10   14
#> [3,]    3    7   11   15
#> [4,]    4    8   12   16
#>
#> $el4
#> [1] 12 17 23  9
unlist(my.list)
```

```
#>  el11  el12  el13  el14  el15  el21  el22  el23  el31  el32
#>   "1"   "2"   "3"   "4"   "5"   "a"   "b"   "c"   "1"   "2"
#>  el33  el34  el35  el36  el37  el38  el39 el310 el311 el312
#>   "3"   "4"   "5"   "6"   "7"   "8"   "9"  "10"  "11"  "12"
#> el313 el314 el315 el316  el41  el42  el43  el44
#>  "13"  "14"  "15"  "16"  "12"  "17"  "23"   "9"
```

Explain the above output.

(e) Review the functions `cbind()`, `rbind()`, `append()`, `data.frame()`, `dim()`, `dimnames()`, `names()`, `colnames()`, `rownames()`, `nrow()` and `ncol()`.

5.6 Rearranging the elements in a matrix

Study the usage of the functions `matrix()`, `t()` and `diag()`. These functions are useful to form submatrices of a matrix or to rearrange matrix elements. Note again the argument `byrow =` of `matrix()`.

5.7 Exercise

1. Write an R function to check if a given matrix is symmetric.

2. Write an R function to extract (i) the row(s) and (ii) the columns containing the maximum value in the matrix. Note that provision must be made that the maximum value can occur in more than one row (column). Furthermore, both the indices and actual values of the rows (columns) must be returned. Illustrate the usage of your function with a suitable example.

3. Describe the variables in the built-in data set `LifeCycleSavings`. Is this data set in the form of a matrix or a dataframe?

4. Use subscripting to find the largest proportion of over 75 in those countries with a dpi of less than 1000 in the `LifeCycleSavings` data set. Also determine the country(ies) having this pop75 value.

5. Consider the `LifeCycleSavings` data set.

   (i) Use subscripting to find the mean aggregate savings for countries with a percentage of the population younger than 15 at least 10 times the percentage of the population over 75.

(ii) Also find the mean aggregate savings for countries where the above ratio is less than 10.
(iii) Use function `t.test()` to test if mean aggregate savings are different for the above two groups.
(iv) Use notched box plots for an approximate test.

(v) First, carefully study the output obtained in (iii) and (iv). Then interpret/discuss this output in detail.

6. Consider the `state.x77` data set and the variable `state.region`. Find the state with the minimum income in each of the regions defined in state.region.

Chapter 6

Revision tasks

In general, the purpose of writing a program in R is to address some practical problem directly or indirectly. To prepare the student for seriously writing R functions (programs) this chapter consists of a mixture of revision tasks. While some of these tasks are straight forward others need more thought and preparation before starting with the writing of R code. In Section 6.1 some guidelines are considered for writing R code to address a practical problem.

6.1 Guidelines for problem solving by writing R code

(a) Make sure the problem is clearly understood. You cannot write good code for something that is not correctly grasped.

(b) Break complex problems into simpler components. Formulate these simpler components in terms of specific questions to be answered.

(c) Think in terms of the way R operates e.g. vectorized arithmetic, recycling principle, operating on objects as wholes/units, subscripting, R data structures . . .

(d) Spend time to prepare your data.

(e) Ask yourself the question what information do you need before attempting to write code for coming up with an answer. Then, what facilities are provided in R to get the necessary information and once the information is available what manipulations are needed to code useful output.

(f) Write dedicated code for answering the specific questions in (b).

(g) Do not neglect the debugging/optimizing phase of code that succeeds in providing a first round answer.

6.2 Exercise

1. Use R to obtain a five-point summary of the variable `dpi` in the `LifeCycleSavings` data set. Illustrate the difference between the working of `fivenum()` and `quantile()`. *Hint*: See `boxplot.stats()` for the definition of hinges.

2. Display the pdf of a $normal(100, 15)$ distribution graphically. The area under the density bounded by the 70th and 90th percentiles must appear in red.

3. Use R to obtain the following graphical representations:

   (i) The pdf as well as the cdf of a $F(15, 10)$ and a $F(10, 15)$ stochastic variable. These graphs must be on one graph window with the same set of axes for both F-distributions and be supplied with suitable titles. Furthermore, they must be line graphs that contain no other plotting characters except lines.

   (ii) Obtain representations as line graphs of the inverses of the above cdfs on a single separate graph page.

4. First set the seed to 172389 and then generate a random sample of size 500 from a $normal(100, 20)$ distribution. Give the necessary R instructions to determine the class frequencies in the class intervals "Smaller than 50", "50 to 75–", "75 to 90–", "90 to 100", "100+ to 110", "Larger than 110".

5. Generate a random sample of size 80 from a bivariate normal distribution with mean vector $(50, 100)$. The variances of the two variables are 900 and 2500 respectively with a correlation 0.90. Store the sample in an R matrix object and obtain a scatterplot in the form of

   (i) a point diagram and
   (ii) a line graph of the sample.

6. Define the harmonic mean for a vector of observations. What conditions must be satisfied by the observations?

   (i) Write your own function for calculating a harmonic mean and use it to calculate the harmonic mean of variable `dpi` in the `LifeCycleSavings` data set.

   (ii) Calculate the ordinary mean of variable `dpi` in the `LifeCycleSavings` data set. Compare the answer with the answer in (a). Which answer would you use in practice? Motivate.

7. Fisher's linear discriminant function in the case of two groups is defined as follows:

$LDF = (\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_2)'\mathbf{S}^{-1}\mathbf{x}$ where $\mathbf{S} = [(n_1 - 1)\mathbf{S}_1 + (n_2 - 1)\mathbf{S}_2]/(n_1 + n_2 - 2)$ with $\bar{\mathbf{x}}_i$ and $\mathbf{S}_i$ the vector of means and the covariance matrix of the ith group (sample), respectively.

The corresponding classification function is written as $CF = (\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_2)'\mathbf{S}^{-1}\mathbf{x} - \frac{1}{2}(\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_2)'\mathbf{S}^{-1}(\bar{\mathbf{x}}_1 + \bar{\mathbf{x}}_2)$. The expression $(\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_2)'\mathbf{S}^{-1}$ is referred to as the discriminant coefficients.

In agreement with section 6.1 make sure what an LDF and a CF entail. The `crabs` data set in package `MASS` consists of 200 rows and 8 columns, describing 5 morphological measurements on 50 crabs each of two colour forms and both sexes, of the species *Leptograpsus variegatus* collected at Fremantle, Western Australia.

(i) Obtain the covariance matrix for each of the two species of crabs.

(ii) Obtain the vector of means for each of the two species of crabs.

(iii) Use standard R functions operating on matrices to write a function or code that calculates the discriminant coefficients for the given linear discriminant function.

(iv) Write a function that determines the linear discriminant function and return

- the discriminant coefficients;
- The CF for each observation.

(v) Repeat the discriminant analysis above, discriminating between male and female crabs, ignoring differences in species.

(vi) Compare your results to using the `lda()` function in the package `MASS` with the command

```
predict (lda (sex ~ FL + RW + CL + CW + BD, data=crabs))$class
```

8. Consider the matrix $\mathbf{A} : n \times m$. What is understood by the column space $V(\mathbf{A})$ and the orthogonal complement $V^{\perp}(\mathbf{A})$? The R function `svd()` can be used to obtain an orthogonal basis for $V(\mathbf{A})$ when the rank of $\mathbf{A}$ is k. We also want to determine an orthogonal basis for $V^{\perp}(\mathbf{A})$. How can the function `svd()` be used to simultaneously find a basis for $V(\mathbf{A})$ and for $V^{\perp}(\mathbf{A})$?

The above propositions can be proved as follows: Assume that $n \geq m$ and that an orthonormal basis for $V(\mathbf{A})$ as well as for $V^{\perp}(\mathbf{A})$ must be found. Append $n-m$ zero vectors of size n to the matrix $\mathbf{A}$. Write $\mathbf{A}^0$ for the appended matrix and perform the function `svd()` on $\mathbf{A}^0$. It follows that $\mathbf{A}^0 = \mathbf{UDV}'$ so that $\mathbf{A}^0\mathbf{V} = \mathbf{UD}$, i.e.

$$\begin{bmatrix}\mathbf{A}^0\mathbf{v}_{(1)} & \mathbf{A}^0\mathbf{v}_{(2)} & \dots & \mathbf{A}^0\mathbf{v}_{(n)}\end{bmatrix} = \begin{bmatrix}d_1\mathbf{u}_{(1)} & d_2\mathbf{u}_{(2)} & \dots & d_n\mathbf{u}_{(n)}\end{bmatrix}.$$

Now $\mathbf{A}^0\mathbf{v}_{(i)} \in V(\mathbf{A}^0) = V(\mathbf{A})$. (*Motivate in detail.*) It follows that $\mathbf{u}_{(i)} \in V(\mathbf{A}), i = 1, 2, \dots, k$. (*Motivate in detail.*) Therefore the columns of $\mathbf{U}$ that correspond to the non-zero ds form an orthonormal basis for $V(\mathbf{A})$ while the columns of $\mathbf{U}$ that correspond to the zero ds form an orthonormal basis for the orthogonal complement of $V(\mathbf{A})$. Motivate the last statement in detail.

9. Based on the results in (8) above, write an R function that returns $rank(\mathbf{A})$, an orthogonal basis for $V(\mathbf{A})$ and an orthogonal basis for $V^{\perp}(\mathbf{A})$. Test your function on the matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 4 \\ 3 & 2 & 7 \\ -1 & -5 & 2 \\ 2 & 7 & -1 \end{bmatrix}$$

10. In many graphical displays whose purpose it is to represent distances in two dimensions, it is essential that the scales of the axes are geometrically accurate. This is called the aspect ratio of the graph and the R graphics parameter `par` is used for controlling the aspect ratio of graphics in R. The default value of `par` generally does not ensure that the scales of the horizontal and vertical axes are geometrically accurate. For ensuring geometrically accurate scales the setting `asp = 1` must be explicitly specified e.g. `plot(x =, y =, asp = 1)`.

We are going to investigate the effect of the aspect ratio on graphs by writing our own function for drawing a circle. In agreement with section 6.1 we will start our project by reviewing some basic concepts regarding coordinates for graphical purposes. Figure 6.1 summarizes how to reference a point in geometric space by using (a) Cartesian coordinates and (b) polar coordinates.

(i) Consider the following function for drawing a circle with a specified radius and centred at the origin:

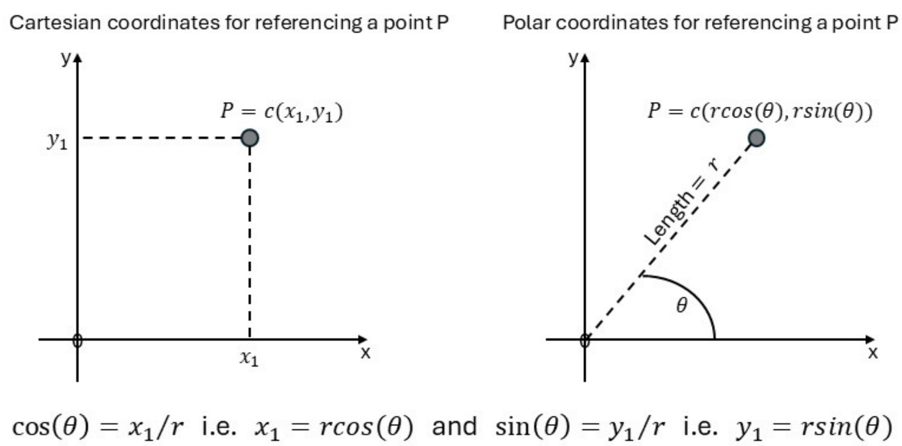


Figure 6.1: Cartesian and polar coordinates for referencing a point on a graph.

```
my.circle <- function (r = 1, xrange = -2:2, yrange = -2:2)
{ plot (x = xrange, y = yrange, type = 'n', xlab = '', ylab = '',
        xaxt = 'n', yaxt = 'n')
  theta <- seq(from = 0, to = 2 * pi, by = 0.01)
  # Notice the use of radians.
  lines (x = r*cos(theta), y = r*sin(theta))
  abline(h = 0)
  abline(v = 0)
}
```

Run the above function and consider the graph window. Increase and decrease the size of the graph window by dragging its edges. Does the figure look like a circle?

(ii) Next, add the argument `asp = 1` to the call to `plot` in `my.circle`. Run the changed function; change the size of the graph window. What happens?

(iii) What changes are necessary for producing a circle centred at any point in a geometrical space? Make the necessary changes in `my.circle()` for constructing a circle centred at any user specified point on a graph.

11. What is understood by a p-dimensional ellipsoid?

    (i) Give a mathematical expression in matrix notation that describes an ellipsoid in p dimensions.

(ii) Describe the axes of the ellipsoid in terms of eigenvalues and eigenvectors.

(iii) Let $p = 2$. Simplify the expression for the ellipse concerned in terms of scalar quantities.

(iv) Use `plot()` and write an R-function to draw an ellipse. Make provision for the centre point to be at $(0, 0)$ as well as at an arbitrary (x_1, x_2) point; for no correlation between the two variables as well as for positive and negative correlation.

(v) Use your function written in (iv) to illustrate differences between plot (using the default value of argument `asp`) and plot with `asp=1`.

12. During experimental design it is often useful to predict the value of the dependent variable at every combination of the levels of the factor variables. Write an R function for this task that makes provision for any number of factor arguments and that also provides a dataframe with the factors as the columns and every combination of levels as the rows. Every levels-combination can only appear once. The function must be user friendly and must test if a given independent variable is a factor variable. *Hint*: Study the help file of `expand.grid()`.

13. Consider the following game. You are given a computer screen containing a rectangle filled at random with evenly spaced letters. Repetitions of the same letter are allowed. The challenge to the user is to sequentially select the first n letters of the alphabet as quickly as possible. The user must read each line from left to right and from top to bottom. Going backwards is not allowed. The time to complete the task is taken as well as whether the rules have been obeyed. Program an R version of this game.

Chapter 7

Writing functions in R

Although we have already written various functions in R, in this chapter the writing of R functions will be approached systematically.

7.1 General

A good way to learn about functions or to write a new function is to look at existing ones. As an example consider that we would like to write a function to implement a novel plotting procedure. So we start by taking a look at the existing `plot` function.

```
plot
#> function (x, y, ...)
#> UseMethod("plot")
#> <bytecode: 0x000002039c1332c8>
#> <environment: namespace:base>
```

This is not very helpful so we give the instruction:

```
methods(plot)
#>  [1] plot.acf*           plot.data.frame*
#>  [3] plot.decomposed.ts* plot.default
#>  [5] plot.dendrogram*    plot.density*
#>  [7] plot.ecdf           plot.factor*
#>  [9] plot.formula*       plot.function
#> [11] plot.hclust*        plot.histogram*
#> [13] plot.HoltWinters*   plot.isoreg*
#> [15] plot.lm*            plot.medpolish*
```

```
#> [17] plot.mlm*          plot.ppr*
#> [19] plot.prcomp*       plot.princomp*
#> [21] plot.profile*      plot.profile.nls*
#> [23] plot.raster*       plot.spec*
#> [25] plot.stepfun       plot.stl*
#> [27] plot.table*        plot.ts
#> [29] plot.tskernel*     plot.TukeyHSD*
#> see '?methods' for accessing help and source code
```

If we decide to take a look at `plot.default` we can do so by

```
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, xgap.axis = NA, ygap.axis = NA,
#>     ...)
#> {
#>     localAxis <- function(..., col, bg, pch, cex, lty, lwd) Axis(...)
#>     localBox <- function(..., col, bg, pch, cex, lty, lwd) box(...)
#>     localWindow <- function(..., col, bg, pch, cex, lty, lwd) plot.window(...)
#>     localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)
#>     xlabel <- if (!missing(x))
#>         deparse1(substitute(x))
#>     ylabel <- if (!missing(y))
#>         deparse1(substitute(y))
#>     xy <- xy.coords(x, y, xlabel, ylabel, log)
#>     if (is.null(xlab))
#>         xlab <- xy$xlab
#>     if (is.null(ylab))
#>         ylab <- xy$ylab
#>     if (is.null(xlim))
#>         xlim <- range(xy$x[is.finite(xy$x)])
#>     if (is.null(ylim))
#>         ylim <- range(xy$y[is.finite(xy$y)])
#>     dev.hold()
#>     on.exit(dev.flush())
#>     plot.new()
#>     localWindow(xlim, ylim, log, asp, ...)
#>     panel.first
#>     plot.xy(xy, type, ...)
#>     panel.last
#>     if (axes) {
#>         localAxis(if (is.null(y))
#>             xy$x
```

```
#>         else x, side = 1, gap.axis = xgap.axis, ...)
#>         localAxis(if (is.null(y))
#>             x
#>         else y, side = 2, gap.axis = ygap.axis, ...)
#>     }
#>     if (frame.plot)
#>         localBox(...)
#>     if (ann)
#>         localTitle(main = main, sub = sub, xlab = xlab, ylab = ylab,
#>             ...)
#>     invisible()
#> }
#> <bytecode: 0x000002039cb3f028>
#> <environment: namespace:graphics>
```

Since our new plotting method is aimed at categorical data we decide rather to take a look at `plot.factor`. But this is an asterisked function and hence is not visible:

```
plot.factor
#> Error: object 'plot.factor' not found
```

Asterisked functions can be inspected using the following method:

```
getAnywhere(plot.factor)
#> A single object matching 'plot.factor' was found
#> It was found in the following places
#>   registered S3 method for plot from namespace graphics
#>   namespace:graphics
#> with value
#>
#> function (x, y, legend.text = NULL, ...)
#> {
#>     if (missing(y) || is.factor(y)) {
#>         dargs <- list(...)
#>         axisnames <- dargs$axes %||% if (!is.null(dargs$xaxt))
#>             dargs$xaxt != "n"
#>         else TRUE
#>     }
#>     if (missing(y)) {
#>         barplot(table(x), axisnames = axisnames, ...)
#>     }
#>     else if (is.factor(y)) {
#>         if (is.null(legend.text))
```

```
#>             spineplot(x, y, ...)
#>         else {
#>             args <- c(list(x = x, y = y), list(...))
#>             args$yaxlabels <- legend.text
#>             do.call("spineplot", args)
#>         }
#>     }
#>     else if (is.numeric(y))
#>         boxplot(y ~ x, ...)
#>     else NextMethod("plot")
#> }
#> <bytecode: 0x000002039bd9f3e0>
#> <environment: namespace:graphics>
```

(a) How are default values assigned to arguments of functions?

(b) What is the default behaviour of `plot.factor()`?

(c) What tasks can be achieved with `pmatch()` and what is understood by partial matching? What will happen if `plot.factor()` is called with (i) `legend.text = 'AA=Agecat'`; (ii) `leg = 'AA=Agecat'`? Explain.

(d) Discuss the usage of `missing()`.

(e) Give an example of the usage of the function `stop(message= " ")`.

(f) Give an example of the usage of the function `warning(message= " ")`.

(g) What is the usage of the function `warnings()`?

(h) Why can functions be called without specifying any arguments e.g. `q()`?

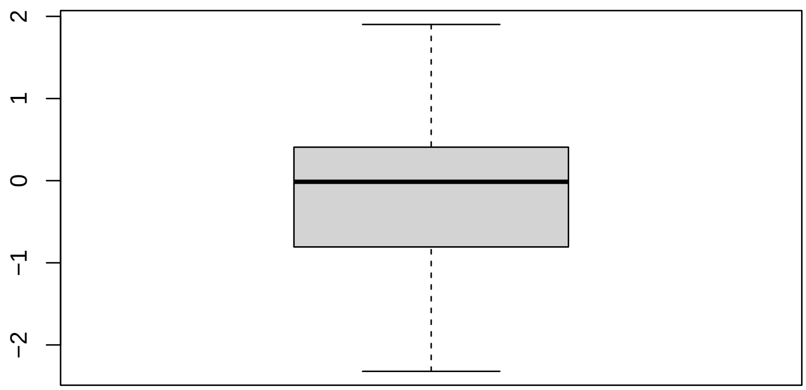
(i) If the body of a function consists only of a single instruction it is not necessary to enclose it with braces.

(j) The convention is to use the last evaluated statement as a function's return value. If several objects are to be returned gather them in a list.

(k) The function `return()` with a single object or a list of objects is useful to interrupt a function at some intermediate stage and return an object or a list of objects at that particular stage. This is usually done when a function is under development.

(l) Sometimes there is no meaningful value to return e.g. when a function is written primarily to produce some plot. In cases like this the function `invisible()` can be used as the last statement of the function. As an example of the usage of `invisible()` give the following instructions:

```r
boxplot(rnorm(100), plot = TRUE)
```




```r
boxplot(rnorm(100), plot = FALSE)
#> $stats
#>             [,1]
#> [1,] -2.13673866
#> [2,] -0.47854052
#> [3,]  0.07906728
#> [4,]  0.79507583
#> [5,]  2.08596468
#>
#> $n
#> [1] 100
#>
#> $conf
#>            [,1]
#> [1,] -0.1221641
#> [2,]  0.2802987
#>
#> $out
#> [1]  3.271010 -2.674717
#>
#> $group
#> [1] 1 1
```

```
#>
#> $names
#> [1] "1"
```

Now look at the end of function `boxplot.default()` to see how `invisible()` has been implemented.

(m) Libraries (packages) of R functions. Attaching and detaching libraries to the search path. (Revise Chapter 1)

(n) Creating a new function using scripts or `fix()`. (Revise Chapter 1)

(o) Editing an existing function using scripts or `fix()`. (Revise Chapter 1)

(p) Note that when writing a function a line can be interrupted at any place and be continued on a next line. *Warning: Be careful not to put the break point where it marks the completion of an executable statement.* Explain.

7.2 Writing a new function

Determining the indices of elements in a vector or matrix that meet a certain condition: the function `where()`

(a) Write the following function:

```
where <- function(x, cond)
{ # Argument cond must evaluate to a logical value
     if(!is.matrix(x))
       seq(along = x)[cond]
     else matrix(c(row(x)[cond], col(x)[cond]), ncol = 2)
}
```

(b) Inspect the *airquality* data set using the command `str(airquality)`.

(c) Use the `where()` function to find the indices of (i) the `NA`s, (ii) the maximum value and (iii) the minimum value in the airquality data set.

(d) Repeat (b) using the built-in function `which()`.

7.3 Checking for object name clashes

(a) What happens if an R object is given the same name as an existing object?

(b) Discuss the usages of the functions `apropos()`, `conflicts()`, `find()` and `match()` for the naming of objects.

(c) Remember that when a function is called the R evaluator first looks in the *global environment* for a function with this name and subsequently in each of the attached packages or date bases in the order shown by `search()`. The evaluator generally stops searching when the name is found for the first time. If two attached packages have functions with the same name one of them will *mask* the object in the other. For example, the function `gam()` exists in two packages: `gam` and `mgcv`. If both were attached the command

```
library (mgcv)
#> Loading required package: nlme
#> This is mgcv 1.9-3. For overview type 'help("mgcv-package")'.
library (gam)
#> Loading required package: splines
#> Loading required package: foreach
#> Loaded gam 1.22-6
#>
#> Attaching package: 'gam'
#> The following objects are masked from 'package:mgcv':
#>
#>     gam, gam.control, gam.fit, s
find("gam")
#> [1] "package:gam"  "package:mgcv"
```

will return both version.

(d) The operator `::` can be used to access the intended version of `gam()` by using the call `mgcv::gam()` or `gam::gam()`.

(e) When writing R packages the *namespace* of the package provides another mechanism for ensuring that the correct version of a function is used. Note in this regard that the operator `:::` can be used to access objects that are not exported.

7.4 Returning multiple values

7.4.1 Exercise

Write an R function that returns the mean, median, variance, minimum, maximum and coefficient of variation of a numeric vector of sample data. The different components must be accessible by name. Test your function with the value of `rnorm(1000)`. *Hint*: Use the construct `list (mean = ..., median = ..., ...)`.

7.5 Local variables and evaluation environments

(a) Where is an object stored that is created by a script or `fix()`?

(b) Where are local objects (objects that are created during the execution of a function) stored?

(c) Explain how the *evaluation environment* works.

(d) What is understood by the *global environment*?

(e) Study the R help-file w.r.t. the operator `<<-`. When is it useful to use this operator? What are the dangers inherent to this operator?

(f) What is understood by the scope of an expression or function?

The symbols which occur in the body of a function can be divided into three classes: *formal parameters*, *local variables* and *free variables*. The formal parameters of a function are those appearing within the parentheses denoting the argument list of the function. Their values are determined by the process of *binding* the actual function arguments to the formal parameters. Local variables are created by the evaluation of expressions in the body of the functions. Variables which are neither formal parameters nor local variables are called free variables. Free variables become local variables when they are assigned to. Consider the following function definition.

```
fun <- function(datvec) {
          mean <- mean(datvec)
          print(mean)
          plot(datvec)
          plot(Traffic)
       }
```

In this function, `datvec` is a formal parameter, the object `mean` on the left-hand of the assignment symbol is a local variable (not to be confused with the function `mean()` on the right-hand side of the assignment symbol) while `Traffic` is a free variable. In R the free variable bindings are resolved by first looking in the *environment* in which the function was created. This is called *lexical scope*.

If the following function call is made from the prompt in the working directory `fun(1:25)` the formal parameter `datvec` within the body of the function is assigned the value `1:25` (the actual argument) and its mean is assigned to the local object `mean`. If the free parameter `Traffic` is found in the *global* *environment* or in a data base on the search path the required graph will be created else an error message will be sent to the console. Perform the above call.

7.6 Cleaning up

(a) Study how the function `on.exit()` is used. This function can be used to reset options that are changed during an R-session back to their original values when the session is ended or a function terminates with an error message. It is also convenient for removal of temporary files.

(b) Study the uses of the functions `.First()` and `.Last()`.

(c) Write a function that automatically opens a graph window with a square plot region when an R-session is started.

7.7 Variable number of arguments: argument `...`

(a) Consider the following situation: You want to write a function for a complex task. At a particular stage a graph of some intermediate results is to be constructed. This requires the calling function to contain a call to the `hist()` function. Here is an example of a chunk of code for executing this task:

```
complexfun <- function(datmat,colgraph)
    { datmat <- scale(datmat)
       # Several lines of complex code here
      hist(datmat, col = colgraph)              }
```

A call like `complexfun(rnorm(1000), 'yellow')` can now be executed for the desired result. The problem is that the hist function has several arguments that you would like to be able to access by passing suitable actual values to them through the calling function `complexfun`. Instead of having to resort to provide a complete set of arguments in the argument list of `complexfun` R provides a neat way of addressing this situation: The argument `...` which acts like any other formal argument except that it can represent a variable number of arguments. To see how the argument `...` works change the above function to:

```
complexfun2 <- function(datmat, ... )
 { datmat <- scale(datmat)
       # Several lines of complex code here
   hist(datmat, ... )    }
```

Arguments represented by argument ... in the argument list of hist are passed to hist through the argument ... appearing in the arguments list of function `complexfun2`:

```
complexfun2(datmat = rnorm(1000), col = 'yellow',
        probability = TRUE, xlim = c(-5,5))
```

(b) Write a function that will retrieve the maximum length of any of an unspecified number of arguments of a specified mode. This is another example of the use of the ... argument:

```
maxlen <- function (mode.use="numeric", ...)
  { my.list <- list(...)
    out <- 0
    for(x in my.list)
      print (mode(x)) #if(mode(x) == mode.use) out <- max(out,length(x))
    out
  }
```

Note that the named argument must be specified as such in the function call:

```
maxlen(1:10, 1:15, 1:3, letters)
#> [1] "numeric"
#> [1] "numeric"
#> [1] "character"
#> [1] 0
maxlen(mode.use="numeric", 1:10, 1:15, 1:3, letters)
#> [1] "numeric"
#> [1] "numeric"
#> [1] "numeric"
#> [1] "character"
#> [1] 0
maxlen(1:10, 1:15, 1:3, letters, mode.use="character")
#> [1] "numeric"
#> [1] "numeric"
#> [1] "numeric"
#> [1] "character"
#> [1] 0
maxlen(mode.use="character", 1:10, 1:15, 1:3, letters)
#> [1] "numeric"
#> [1] "numeric"
#> [1] "numeric"
#> [1] "character"
#> [1] 0
```

7.8 Retrieving names of arguments: functions `deparse()` and `substitute()`

There are many practical situations requiring the conversion of mathematical expressions into character strings (text) or, conversely, requiring the conversion of text into mathematical expressions. The tools (functions) provided in R for achieving such conversions are summarized in Figure 7.1.

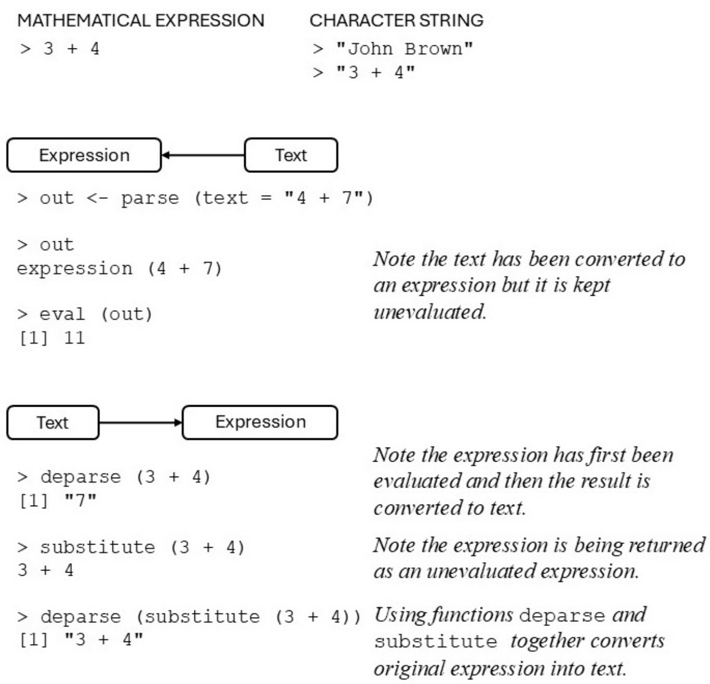


Figure 7.1: Converting text into mathematical expression or mathematical expressions into text.

- Task: write an R function that will plot two vectors using as axis labels the names of the objects passed as arguments to the function.

It follows from Figure 7.1 that the function `substitute()` takes an expression as argument and returns it unevaluated. In order to evaluate the return value of `substitute()` the function `eval()` must be used. The function `deparse()` takes as argument an unevaluated expression and converts it into a character string. Now we are ready to write the following function:

```
labplot <- function (x,y)
{ xname <- deparse(substitute(x))
 yname <- deparse(substitute(y))
 plot(x,y, xlab=xname, ylab=yname, main = paste("Plot of",
        yname,"versus", xname))
}
```

(a) Study and illustrate the usage of function `labplot()`.

(b) From Figure 7.1 it also follows that the function `parse()` does the opposite of `deparse()` by converting a character string into an unevaluated expression. The latter unevaluated expression can be evaluated when needed using `eval()`.

7.9 Operators

Execute the following instruction

```
objects('package:base')[1:31]
#>  [1] "-"                    "-.Date"
#>  [3] "-.POSIXt"             "!"
#>  [5] "!.hexmode"            "!.octmode"
#>  [7] "!="                   "$"
#>  [9] "$.DLLInfo"            "$.package_version"
#> [11] "$<-"                  "$<-.data.frame"
#> [13] "$<-.POSIXlt"          "%%"
#> [15] "%*%"                  "%/%"
#> [17] "%||%"                 "%in%"
#> [19] "%o%"                  "%x%"
#> [21] "&"                    "&&"
#> [23] "&.hexmode"            "&.octmode"
#> [25] "("                    "*"
#> [27] "*.difftime"           "/"
#> [29] "/.difftime"           ":"
#> [31] "::"
```

in order to obtain some examples of operators available in R.

(a) *Operators are special R functions.* Discuss this statement. In what respects do operators differ from ordinary R functions?

(b) Write an operator `%E%` to determine the Euclidean distance between two vectors and give an example of its usage. *Hint*: when creating operators with `fix()` or using scripts the name must be given as a character string e.g. `fix("%E%")`.

7.10 Replacement functions

Execute the following instruction

```
objects('package:base')[300:400]
#>   [1] "c.factor"
#>   [2] "c.noquote"
#>   [3] "c.numeric_version"
#>   [4] "c.POSIXct"
#>   [5] "c.POSIXlt"
#>   [6] "c.warnings"
#>   [7] "call"
#>   [8] "callCC"
#>   [9] "capabilities"
#>  [10] "casefold"
#>  [11] "cat"
#>  [12] "cbind"
#>  [13] "cbind.data.frame"
#>  [14] "ceiling"
#>  [15] "char.expand"
#>  [16] "character"
#>  [17] "charmatch"
#>  [18] "charToRaw"
#>  [19] "chartr"
#>  [20] "chkDots"
#>  [21] "chol"
#>  [22] "chol.default"
#>  [23] "chol2inv"
#>  [24] "choose"
#>  [25] "chooseOpsMethod"
#>  [26] "chooseOpsMethod.default"
#>  [27] "class"
#>  [28] "class<-"
#>  [29] "clearPushBack"
#>  [30] "close"
#>  [31] "close.connection"
#>  [32] "close.srcfile"
#>  [33] "close.srcfilealias"
#>  [34] "closeAllConnections"
#>  [35] "col"
#>  [36] "colMeans"
#>  [37] "colnames"
#>  [38] "colnames<-"
#>  [39] "colSums"
#>  [40] "commandArgs"
```

```
#>  [41] "comment"
#>  [42] "comment<-"
#>  [43] "complex"
#>  [44] "computeRestarts"
#>  [45] "conditionCall"
#>  [46] "conditionCall.condition"
#>  [47] "conditionMessage"
#>  [48] "conditionMessage.condition"
#>  [49] "conflictRules"
#>  [50] "conflicts"
#>  [51] "Conj"
#>  [52] "contributors"
#>  [53] "cos"
#>  [54] "cosh"
#>  [55] "cospi"
#>  [56] "crossprod"
#>  [57] "Cstack_info"
#>  [58] "cummax"
#>  [59] "cummin"
#>  [60] "cumprod"
#>  [61] "cumsum"
#>  [62] "curlGetHeaders"
#>  [63] "cut"
#>  [64] "cut.Date"
#>  [65] "cut.default"
#>  [66] "cut.POSIXt"
#>  [67] "data.class"
#>  [68] "data.frame"
#>  [69] "data.matrix"
#>  [70] "date"
#>  [71] "debug"
#>  [72] "debuggingState"
#>  [73] "debugonce"
#>  [74] "declare"
#>  [75] "default.stringsAsFactors"
#>  [76] "delayedAssign"
#>  [77] "deparse"
#>  [78] "deparse1"
#>  [79] "det"
#>  [80] "detach"
#>  [81] "determinant"
#>  [82] "determinant.matrix"
#>  [83] "dget"
#>  [84] "diag"
#>  [85] "diag<-"
```

```
#>  [86] "diff"
#>  [87] "diff.Date"
#>  [88] "diff.default"
#>  [89] "diff.difftime"
#>  [90] "diff.POSIXt"
#>  [91] "difftime"
#>  [92] "digamma"
#>  [93] "dim"
#>  [94] "dim.data.frame"
#>  [95] "dim<-"
#>  [96] "dimnames"
#>  [97] "dimnames.data.frame"
#>  [98] "dimnames<-"
#>  [99] "dimnames<-.data.frame"
#> [100] "dir"
#> [101] "dir.create"
```

and notice that some object names appear in pairs with the name of one member of the pair ending in `<-`. Examples are `dim<-`, `levels<-`, `diag<-`, `names<-`, `rownames<-`, `colnames<-` and `dimnames<-`. Functions having names ending in `<-` are called *replacement* functions. A replacement function appears on the left-hand side of the assignment symbol using the name without the `<-` to replace contents of the objects appearing in its argument list by the contents of the object appearing at the right-hand side of the assignment symbol e.g.:

```
X <- matrix (1:12, ncol = 3, dimnames =
               list (paste0 ("Row", 1:4), paste0 ("X", 1:3)))
a <- rownames(X) # Function rownames in action.
rownames(X) <- 1:nrow(X) # Replacement function 'rownames<-' in action.
```

How can the object `diag<-` be inspected and is it different from the object `diag`? Compare the result of the following function calls:

```
getAnywhere('diag')
#> 2 differing objects matching 'diag' were found
#> in the following places
#>   package:base
#>   namespace:Matrix
#>   namespace:base
#> Use [] to view one of them
getAnywhere('diag<-')
#> 2 differing objects matching 'diag<-' were found
#> in the following places
#>   package:base
```

```
#>   namespace:Matrix
#>   namespace:base
#> Use [] to view one of them
```

In what respects do replacement functions differ from other functions?

In order to write a replacement function the following rules must be met:

(i) the function name must end in `<-`

(ii) the function must return the complete object with suitable changes made

(iii) the final argument of the function corresponding to the replacement data on the right-hand side of the assignment, must be named `value`

(iv) usually a companion function exists having the same name without the `<-`.

As an example, write a replacement function `undefined()` that will replace missing values in a data object with the values on its right-hand side:

```
"undefined<-" <- function (x, codes = numeric(), value)
  { if (length(codes) > 0) x[x %in% codes] <- NA
    x[is.na(x)] <- value
    x
  }
```

The above function can be created or edited using `fix("undefined<-")`. Illustrate the usage of `undefined()`.

7.11 Default values and lazy evaluation

(a) The function `match.arg()` is useful for selecting a default value from one of a set of possible values. Consider the following example:

```
choice <- function(method=c("PCA","CVA","CA","NONLIN"))
   { match.arg(method)  }
choice()
#> [1] "PCA"
choice("CVA")
#> [1] "CVA"
choice("xx")
#> Error in match.arg(method): 'arg' should be one of "PCA", "CVA", "CA", "NONLIN"
```

(b) Functions in the R language are governed by a principle known as *lazy evaluation* which means that a default value is not evaluated until it is actually needed within the function body. As a result of lazy evaluation it might happen in a function call that some default values are never evaluated.

7.12 The dynamic loading of external routines

Compiled code can run in some instances much faster than corresponding code in R. The functions `.C()` and `.Fortran()` allow users to make use of programs written in *C* or *Fortran* in their R functions. How this is done is illustrated below. Study this example carefully and consult the help files for more details when needed.

First an R function is created to compute the matrix product of two matrices:

```
matmult <- function (A,B)
 { if(ncol(A) != nrow(B)) stop("A and B not conformable with
                        respect to matrix multiplication \n")
   n <- nrow(A)
   q <- ncol(B)
   Cmat <- matrix(NA, nrow=n, ncol=q)
   for(i in 1:n)
      { for(j in 1:q) Cmat[i,j] <- sum(A[i,] * B[,j])
      }
  Cmat
  }
```

Next a *Fortran* subroutine is written for performing matrix multiplication. The *Fortran* code for this subroutine is given below:

```
      SUBROUTINE MATM (A1, A2B1, B2, A, B, OUT)
C     This subroutine performs matrix multiplication.
C     This should be improved with optimized code (such as
C     from Linpack, etc.)
      IMPLICIT NONE
      INTEGER A1, A2B1, B2
      DOUBLE PRECISION A(A1,A2B1), B(A2B1,B2), OUT(A1,B2)
C     DUMMIES
      INTEGER I, J, K
      DO 300,J=1,B2
        DO 200,I=1,A1
          OUT(I,J)=0
          DO 100,K=1,A2B1
```