Hints about **R**

version 2.03, Feruary 2022

Georgi Boshnakov

Contents

1 Introduction

R is a powerful system for data analysis. To install it, go to the **R** Project site (just Google `R` and the first link will probably be spot on) and choose the installer for your operating system. From the **R** Project site you can learn about various extentions, such as graphical user interfaces (GUIs) and integrated development environments (IDEs). If you are going to use **R** regularly, it is highly recommended to consider RStudio. This guide however does not discuss GUis and IDEs, although it occasionaly mentions menu items or similar things available in Rgui for Windows. Also, this is not a systematic introduction to **R**. Some links can be found at R resources, see also Section 7.1 below.

The aim of this document is to provide an overview of basic concepts and techniques, as well as tips on using **R** that I find useful. You need to read this document at least once to get a general feeling of what is available. In reality, you will strugle if you do not master the basic arithmetic, vector and matrix operations (Section 5) as well as basic data import (Section 10.2). As is the case with virtually any skill, "mastering" cannot be achieved by just reading and watching. It is important to do examples and experiment with similar ones.

Bear in mind that it is impossible to know everything about **R** and it is natural to forget things that you don't use regularly. To recall what you need quickly, it is important to be able to use confidently the help system of **R**. In many cases you can just grab an example from the help page of a function and modify it to your needs. Internet searches are also useful, especially when you want extended explanation on a topic.

2 Starting and quitting R

2.1 Starting from the Start menu

R can be launched as any other application, e.g., by clicking on an icon. On Windows the name of the program may be something like `R x64 4.0.3` or `R i3864.0.3`, where `4.0.3` is the version of R, `x64` is for 64 bit operating systems, and `i386` for 32 bit operating systems. Choose the most recent x64 version (`4.0.3` if the settings are as above), unless you have reasons to choose another version.

2.2 The working directory

The working directory is a directory where **R** looks for files when you do not tell it where to look for them. You can view and/or set it with the following commands:

```
getwd()           # Get Working Directory
setwd("P:/psi")   # Set Working Directory (replace "P:/psi" with the desired dir.)
```

Alternatively (on Windows) choose `Change dir...` from the `File` menu and either type the name of the desired folder, e.g. `P:\`, or use the browse button to navigate to it. **R** may create folders like `R-4.0.3` or `R` for its own use, do not put your stuff in them, create appropriately named folders for your own projects/courses.

2.3 Quitting R and saving the workspace image

You quit **R** as any other application. When you quit **R** it asks you whether to save the "Workspace image". If you answer "yes" a file named `.Rdata` is saved in your current working directory along with a

file `.Rhistory`. The file `.RData` contains the results of the computations you have done and `.Rhistory` records the **R** commands you have used.

You may save the workspace at any time (under a more desciptive name) during your **R** session by choosing the `Save workspace` item from the `File` menu. Similarly, the item `Save history` saves a transcript of your commands.

2.4 Starting with a previously saved workspace

Usually **R** is configured during installation so that you can click on a workspace file, such as `.RData`, to start **R**: it will be (almost) exactly in the state it was when the workspace was saved. The working directory will be the one where the `.RData` file resides.

You may also load a workspace image at any time using the `Load workspace` item from the `File` menu. Similarly, you may load a previously saved `Rhistory` file with the menu item `Load history`.

A lecturer who uses **R** may put such workspace images and history files, e.g. saved at the end of lectures, online. You can open the `.Rhistory` files in your browser or a text editor (such as `Notepad`) and copy and paste (or edit) the relevant commands. The simplest way to use `.RData` files is to download them to your working directory and proceed as outlined above.

3 Organising your work

This is of course a matter of personal style. Here are some hints I find useful.

Folders (directories) Create a folder for the course unit. For projects that are scattered across several files (such as documents, graphs, scripts), subfolders may be effective. Choose descriptive names for your folders.

Sometimes you may need to give the full path to a file in a command. In such cases you will feel happier if your folders had shorter names.

Working directory Make sure that the working directory is the one for the project you are working on.

Keep notes Keep notes with recipes about things you have done and that may be useful in future, especially if it is not immediately obvious where to look for a particular feature. I do not mean extensive notes here but if they are in electronic form the size is not an issue. For example, you could create a directory `MyRhints` and put there a file Notes and small scripts illustrating interesting features.

4 Typesetting conventions for these notes

Commands typed by the user (me or you) and results printed by **R** are typeset in a monospaced typewriter font. Lines entered by the user are shown as they were typed and can be copied and pasted into an **R** session. Output printed by **R** in response to the user commands are prefixed with `##`. For example:

```
2 + 2

## [1] 4
```

The first line shows what I have typed, `2 + 2`. The second line is the result of the computation.

Occasionally, when you enter a command, **R** will not compute anything but will respond by printing a `+` sign, as in:

```
> x <- (2 + 2) /
+
```

(`>` and `+` are command prompts printed by **R**). This means that your expression is incomplete and **R** expects you to complete it. Do not panic, complete the expression and press `Enter`:

```
x <- (2 + 2) /
+ 5
```

If you think that the expression is in fact complete, then there is something wrong with it (often a missing closing parenthesis). If this is the case, press the escape key, `Esc`, to abort the command, then correct it.

5 Basic usage

When you start **R** you will see the familiar toolbar with menus like `File`, `Edit`, `Windows`, `Help`, and a few more. You type commands in the `R console` window (sometimes called also the Command window) and get numerical and textual results of calculations in the same window. Graphs are drawn in a separate window (*graphics device* in **R** jargon).

To store the result(s) of a computation, say `2 + 2`, in an object, `x`, use

```
x <- 2 + 2
```

The assignment operator `<-` consists of the *less than* sign followed by a *minus* sign. The more familiar

```
x = 2 + 2
```

is acceptable as well and does the same job. I prefer `<-` for assignment since `=` has another important meaning (see section 8).

Names of objects may consist of letters, numbers, dots and underscore, `_`. The case of letters matters—`AirPassengers` is completely different name from `airpassengers`. You may see the value of an **R** object by typing its name, e.g.

```
x

## [1] 0.8
```

For large objects it is normally better to use `summary`, `head`, `tail`, `plot` or `show`.

For readability you may put as many spaces as you like in a command as long as you do not split syntactic entities, such as names of objects and the assignment operator `<-`.

5.1 Data types

The basic data unit in **R** is a vector. A scalar is just a vector containing a single value.

A vector can be `numeric` (numbers), `logical` (boolean values) and `character` (arbitrary text). There is also a specialised type, `factor`, for categorical variables (ordered or unordered).

Text values are enclosed in quotes (single or double). Logical values can be `TRUE` or `FALSE` (without quotes!).

To put some values in a vector use the function `c` (an abbreviation for *concatenate*), as in

```
c(1, 2, pi, exp(1), sqrt(2), NA, Inf, NaN) # numeric

## [1] 1.000000 2.000000 3.141593 2.718282 1.414214       NA      Inf      NaN

c("fruit", "veg", NA)                      # character

## [1] "fruit" "veg"   NA

c(TRUE, FALSE, TRUE, NA)                   # logical

## [1]  TRUE FALSE  TRUE    NA
```

These examples illustrate also some special values: `NA` represents a missing value, `Inf` represents infinity. It shows also that if you do not assign the result to a variable, then it is printed.

A `list` is like a vector but it can contain values of different types:

```
wd <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
        "Saturday", "Sunday")
list(names = wd, mo_first = 1:7, su_first = c(2:7, 1), su_0 = c(1:6, 0))
```

```
## $names
## [1] "Monday"    "Tuesday"   "Wednesday" "Thursday"  "Friday"    "Saturday"
## [7] "Sunday"
##
## $mo_first
## [1] 1 2 3 4 5 6 7
##
## $su_first
## [1] 2 3 4 5 6 7 1
##
## $su_0
## [1] 1 2 3 4 5 6 0
```

Factors can be particularly useful for non-numeric variables when all possible values are known in advance. Such variables can be represented by character vectors. For example, let's suppose that the character vector `survey_resp` below represents the answers to one question from a survey, where the answers are on a standard a standard choice of responses to a question in a survey.

```
responses <- c("Strongly_disagree", "Disagree", "Neutral", "Agree", "Strongly_agree")
survey_resp <- sample(responses, 10, replace = TRUE)
survey_resp

##  [1] "Agree"             "Disagree"          "Strongly_agree"
##  [4] "Agree"             "Strongly_disagree" "Strongly_agree"
##  [7] "Agree"             "Disagree"          "Disagree"
## [10] "Agree"
```

This often is fine but here **R** will not know if all possible answers are given by at least one person, neither that the responses are ordered. We can supply this information by converting the data to factor vector:

```
facresp <- factor(survey_resp, levels = responses, ordered = TRUE)
facresp

##  [1] Agree             Disagree          Strongly_agree    Agree
##  [5] Strongly_disagree Strongly_agree    Agree             Disagree
##  [9] Disagree          Agree
## 5 Levels: Strongly_disagree < Disagree < Neutral < ... < Strongly_agree
```

If the values of the factor are not ordered, omit argument `ordered`.

5.2 Mathematical expressions

The arithmetic operators `+`, `-`, `*`, `/` have their usual meaning, `^` stands for raising to a power, names of mathematical functions are consistent (as far as possible) with traditional mathematical notation. Functions may be called using familiar syntax, e.g.

```
sin(pi / 4) + cos(3 * x^2) - exp(-2 * (y + 1) * (y - 1))
```

computes $\sin(\pi/4) + \cos(3x^2) - \exp(-2(y+1)(y-1))$, assuming that `x` and `y` have been assigned some values previously. Note that the multiplication sign in **R** expressions should not be omitted. Details about arithmetic functions can be obtained by the command

```
?Arithmetic
```

5.3 Creating and subsetting vectors

To put some numbers in a vector you can use the function `c`, see Section 5.1. There are also functions for easy creation of some frequently used patterned vectors. In the following examples the usage should be more or less clear, see the help pages of the functions for more details (`?seq`, etc).

```
numeric(5)          # a vector of five zeroes

## [1] 0 0 0 0 0

1:9                 # numbers from 1 to 9 (incremented by 1)

## [1] 1 2 3 4 5 6 7 8 9

seq(1, 9, by = 2)   # numbers from 1 to 9 (incremented by 2)

## [1] 1 3 5 7 9

9:1                 # numbers from 9 to 1 (incremented by -1)

## [1] 9 8 7 6 5 4 3 2 1

rep(1, 5)           # five repetitions of 1

## [1] 1 1 1 1 1

rep(c(1, 2), 5)     # five repetitions of c(1,2)

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

Vectors need not be numeric, they may be character as well. For example,

```
c("male", "female")

## [1] "male"   "female"
```

creates a character vector of length 2 whose first element is the word *male* and the second is *female*. There are some predefined character vectors in **R**, e.g.

```
letters

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

is a vector containing the lowercase Latin letters.

If you wonder why **R** prints `[1]` at the left of its output, the last example should clarify it: `[1]` says that the first item following it is the first element of the vector `letters`. Similarly, `[20]` on the second row states that the first element, the letter `t`, on that row is the 20th element of the vector.

Elements of vectors can be extracted by specifying the indices in square brackets. For example, the fourth letter of the alphabet is

```
letters[4]

## [1] "d"
```

More elements may be extracted in one go by giving the index as a vector of integers. For example,

```
letters[1:4]

## [1] "a" "b" "c" "d"
```

extracts the first four elements of the alphabet.

5.4 Matrices

The examples here show how to combine and reshape matrices and vectors.

```
u <- matrix(0, 3, 4)            # 3x4 matrix of zeroes
```

```
x <- c(1.1, 1.2, 1.3, 1.4, 2.1, 2.2, 2.3, 2.4)
```

```
y <- matrix(x, 2, 4)    # reshapes x as a matrix, x[1], x[2] in 1st col, etc.
```

```
z <- matrix(x, 2, 4, byrow = TRUE) # as above but now x[1], ..., x[4] in 1st row, etc.
```

```
rbind(y, u, z)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  1.1  1.3  2.1  2.3
## [2,]  1.2  1.4  2.2  2.4
## [3,]  0.0  0.0  0.0  0.0
## [4,]  0.0  0.0  0.0  0.0
## [5,]  0.0  0.0  0.0  0.0
## [6,]  1.1  1.2  1.3  1.4
## [7,]  2.1  2.2  2.3  2.4
```

The last command, `rbind`, forms a larger matrix by stacking a number of matrices and vectors on top of each other ("r" in `rbind` stands for *row*, i.e. assemble rows) The function `cbind` is similar but puts the matrices and vectors next to each other (i.e. assembles columns). Compare the following

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Subscripts work naturally:

```
z[2, 3]
```

```
## [1] 2.3
```

Vectors may be used as indices, as well:

```
z[1, 2:3]
```

```
## [1] 1.2 1.3
```

```
z[1, 1:4]
```

```
## [1] 1.1 1.2 1.3 1.4
```

The easiest way to take a whole column or row is to leave the corresponding index blank:

```
z[1, ]   # 1st row

## [1] 1.1 1.2 1.3 1.4

z[ , 2]  # 2nd column

## [1] 1.2 2.2
```

To extract a submatrix, use vectors for both indices:

```
z[1:2, 3:4]

##      [,1] [,2]
## [1,]  1.3  1.4
## [2,]  2.3  2.4
```

You may have noticed in the examples above that extracting a single row or column gives a vector, not a matrix. To keep the result a matrix use argument `drop` with value `FALSE`:

```
z[1,  , drop = FALSE]  # 1st row as a 1x4 matrix

##      [,1] [,2] [,3] [,4]
## [1,]  1.1  1.2  1.3  1.4

z[ , 2, drop = FALSE]  # 2nd column as a 2x1 matrix

##      [,1]
## [1,]  1.2
## [2,]  2.2
```

5.5 Data frames

Datasets are usually represented in **R** as data frames. Each column in a data frame contains data for one variable while each row contains the observations for a single subject/item. A data frame is similar to a matrix. The main difference is that the columns of a data frame may be of different types (e.g. numeric and character) while all elements of a matrix must have the same type.

Here we create a data frame containing random samples from two distributions: `x` is standard normal, `y` is Poisson(2).

```
u <- data.frame(x = rnorm(10), y = rpois(10, lambda = 2))
u

##              x y
## 1  -0.47719270 2
## 2  -0.99838644 2
## 3  -0.77625389 1
## 4   0.06445882 2
## 5   0.95949406 1
## 6  -0.11028549 2
## 7  -0.51100951 3
## 8  -0.91119542 2
## 9  -0.83717168 1
## 10  2.41583518 3

colnames(u)   # what are the names of the columns of u?
```

```
## [1] "x" "y"

rownames(u)   # what are the names of the rows of u ()?

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

Parts of the data frames can be extracted similarly to matrices, or using list syntax, e.g.:

```
u[ , 1]      # extract column "x"
u[ , "x"]    #  "" same but more clear!
u$x          #  "" same
```

5.6 Mathematical operations on vectors and matrices

Arithmetic operations on vectors and matrices are applied elementwise. Study the following examples.

```
x <- 1:3      # x =   [1,   2,   3  ]
y <- 2:4      # y =   [2,   3,   4  ]
x + y         # x + y = [1 + 2, 2 + 3, 3 + 4]    (element-wise arithmetic)

## [1] 3 5 7

x * y         # x * y = [1 * 2, 2 * 3, 3 * 4]

## [1]  2  6 12

x ^ y         # x^y = [1^2, 2^3, 3^4]

## [1]  1  8 81
```

In general, the vectors in such operations need to have equal lengths but **R** has a more relaxed attitude. For example, in expressions like the following

```
x + 2

## [1] 3 4 5

x ^ 2

## [1] 1 4 9

2 ^ (1:5)

## [1]  2  4  8 16 32
```

the scalar is treated as if it were a vector of the same length as `x` obtained by repeating the scalar that many times (**R** calls this approach "the recycling rule").

If the argument of a standard mathematical function is a vector, then the function is applied to each element of the vector.

```
sqrt( c(1, 4, 9, 16, 25) )

## [1] 1 2 3 4 5
```

The function `sum` computes the sum of all elements of a vector, e.g.:

```
b <- 1:10
sum(b)

## [1] 55
```

The net effect of the R arithmetic rules is that most arithmetic calculations are simple translations of the traditional mathematical expressions. Here we compute the sample mean and sample variance:

```
a <- rnorm(10)   # put some numbers in a vector
n <- length(a)
abar <- sum(a)  /  n                    # mean of a
avar <- sum( (a - abar)^2 ) / (n-1)     # variance of a
c(mean = abar, var = avar)              # collect mean and var in a vector

##       mean        var
## -0.5061279  0.7377607
```

The function `t` transposes a matrix. Inner product and matrix multiplication are performed with the operator `%*%` (notice that this is NOT `*`, `*` multiplies elementwise, see above). Compare the following.

```
y <- matrix(1:6, nrow = 2)
y

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

y %*% t(y)         # matrix product

##      [,1] [,2]
## [1,]   35   44
## [2,]   44   56

y*y                # element-wise product can be written also y^2

##      [,1] [,2] [,3]
## [1,]    1    9   25
## [2,]    4   16   36
```

Notice that the matrix product of `y` by `y` is not defined but the elementwise product is.

The command `solve(a, b)` solves the linear system $ax = b$, while `solve(a)` gives the inverse matrix, a^{-1}, of `a`.

5.7 Saving objects to files

We discussed in section 2.3 how to save all objects in the workspace to file `.RData`. A more organised approach would be to save selected objects. For example, here we save three objects in file `abc.RData`.

```
a <- 1:3
b <- function(x) x^2
c <- "some text"
save(a, b, c, file = "abc.RData")
```

`.RData` and `.rda` are the conventional file extensions for such files. You can load the object in an **R** session as follows:

```
load("abc.RData")
```

Another efficient way to save objects for later use in other **R** sessions is offered by the function `saveRDS`. It saves a single object which can be imported with `readRDS`. For example, the following saves the object `a` to file `a.rds`:

```
saveRDS(a, file = "a.rds")
```

It can be read in back by

```
z <- readRDS("a.rds")
```

Notice that `saveRDS` doesn't save the name of the object to the file. When the file is read in, we are free to give the object whatever name we like.

6 Logical expressions and indexing

There are operators for testing equality (`==`), inequality (`!=`). The remaining comparison operators are `>`, `<`, `>=` and `<=` with obvious meaning. The logical operators are: `|` (logical OR), `&` (logical AND), and `!` (logical NOT).

Similarly to the arithmetic operators, all operators listed above are vectorised, i.e., when they are applied to vectors and matrices, they do the comparisons element by element and return vectors or matrices. For example,

```
v1 <- c(pi, exp(1), exp(2))
i3 <- v1 > 3
i3

## [1]  TRUE FALSE  TRUE
```

A very important use of the above comparison operators is for indexing. If `ind` is a logical vector of the same length as `x`, then `x[ind]` extracts the elements of `x` at indices where `ind[i]` is `TRUE`. Continuing the above example,

```
v1[i3]     # take the elements of v1 that are greater than 3

## [1] 3.141593 7.389056

v1[v1 > 3] # same, no need to create intermediate variable

## [1] 3.141593 7.389056
```

This illustrates the use of `|`. In this case we extract those elements of `v2` that are either negative or greater than 1.6.

```
v2 <- log(18:100 / 20)
ind <- v2 < 0 | v2 > 1.6
v2[ind]

## [1] -0.10536052 -0.05129329  1.60943791
```

See `?Logic` for more details.

6.1 Logical AND and OR for conditional expressions

There are also the logical operators `||` (logical OR) and `&&` (logical AND). They operate on single values and return return single `TRUE` or `FALSE`, which makes them suitable for use in conditional expressions, such as `if()`, see Section 12.3.

R has many useful functions that return a single TRUE or FALSE value, such as:

```r
is.numeric(x)   # is x a numeric vector?
is.character(x) # is x a character vector?

all(x)  # are all values in the logical vector x TRUE?
any(x)  # is there at least one TRUE value in the logical vector x?
any(!x) # is there at least one FALSE value in the logical vector x?
```

For further details on the logical operators and the difference between them, see their help page, `help("||")`.

7 Further operations on data frames and matrices

In this section I introduce the function `apply`, which may look scary initially. Don't be! Simply copy the examples and insert your data or function, as needed.

I will use as example the data frame `cars` available in **R**, see `?cars` for its description. Here are its first few rows:

```r
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

And here is a summary:

```r
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
```

Suppose that you wish to get the medians only or to compute the standard deviations of the columns. We could do this for each column:

```r
median(cars$speed)
```

```
## [1] 15
```

```r
median(cars$dist)
```

```
## [1] 36
```

but `median(cars)` will give an error message. There is a general solution with the help of the function `apply`, e.g.

```r
apply(cars, 2, median)
```

```
## speed  dist
##    15    36
```

This command can be deciphered as follows: *Dear function* `apply`*, please apply function* `median` *to each column of dataset* `cars`*.* The first argument specifies the dataset (data frame or matrix), the third argument gives the function to apply. The misterious number 2 in the second argument asks the function to be applied to columns. To apply it to rows, use 1 instead. This example is sufficient for vast number of tasks, simply change the dataset and the function as you need. See the help of `apply` for more details and examples.

You can give the name of a function or your own function:

```
apply(cars, 2, "sd") # standard deviations by column

##     speed      dist
##  5.287644 25.769377

apply(cars, 2, function(x) var(x) / mean(x)) # coef. of variation

##     speed      dist
##  1.815531 15.450461
```

7.1 Using the help system

The `Help` menu contains extensive **R** documentation in many formats. There are several manuals in PDF format (for Adobe Reader), among them is *An introduction to* ***R***. The help pages for the individual functions are available in html format for browsing, as a PDF document for reference, and via the `help` command for quick access. For example,

```
?matrix
```

shows information about the function `matrix`. Alternatively, the `help.search` command may be useful to find information when you do not know the name of a specific function,

```
help.search("five number summary")
```

`help.search` may give huge amount of information for popular topics. You can narrow the search by using the optional argument `package`. Most core **R** functions are either in package `stats` or package `base`.

```
help.search("Normal distribution", package = "stats")
help.search("matrices", package = "base")
```

Another useful command is `apropos` which lists functions and variables whose names contain a given string, e.g.

```
apropos("var")
```

If you use the help system routinely, you will progress quickly. Many commands and statistical concepts can be mastered very efficiently by doing and modifying the examples in the corresponding help pages.

7.2 Practicing with R

For meaningful exercises **R** provides many datasets. The command `data()` shows a list of the datasets available in your **R** session. Moreover, the help pages (see Section 7.1) of most functions contain examples that you may run with or without modification. It is not even necessary to retype them—just copy and paste the chunks of code you wish to execute.

The easiest way to create your own examples is simulation. For example, the following creates a data frame

```
x <- rnorm(100, mean = 178, sd = 20)
y <- -100 + x + rnorm(100, sd = 5)
u <-data.frame(height = x, weight = y)
```

This artificial example represents a population with average height 178 cm and weight generally related to height by the formula $w = h - 100 + \varepsilon$. We may fit a simple regression model to the data in the data frame.

```
lm(weight ~ height, data = u)

##
## Call:
## lm(formula = weight ~ height, data = u)
##
## Coefficients:
## (Intercept)       height
##     -94.897        0.976
```

7.3 Names for parts of objects

Parts of objects may be named to aid understanding and for easier reference. Matrices and data frames may have column names and row names.

Functions `names`, `rownames` and `colnames` may be used to get and change them:

```
colnames(cars)

## [1] "speed" "dist"
```

Names may be given when an object is created:

```
x.summary <- c(median = median(x), mean = mean(x), "Std. Dev" = sd(x))
x.summary

##    median      mean  Std. Dev
## 174.56219 176.69333  19.67338
```

They may also be set or changed after the object is created:

```
y.summary <- c(median(y), mean(y), sd(y))
names(y.summary) <- c("median", "mean", "Std.dev")
y.summary

##   median     mean  Std.dev
## 76.23697 77.55386 19.91067
```

The names may be set or changed individually, as well, e.g.

```
names(y.summary)[2] <- "Ybar"
y.summary

##   median     Ybar  Std.dev
## 76.23697 77.55386 19.91067
```

Some **R** functions propagate the names, e.g.

```
rbind(x.summary, y.summary)

##              median      mean Std. Dev
## x.summary 174.56219 176.69333 19.67338
## y.summary  76.23697  77.55386 19.91067
```

See also the help pages for `rownames` and `colnames`.

As a final example, here is a way to make the output of `fivenum` more readable:

```
x.5 <- fivenum(x)
names(x.5) <- c("low.whisker", "low.hinge", "median", "upp.hinge", "upp.whisker")
x.5

## low.whisker   low.hinge      median   upp.hinge upp.whisker
##    134.3992    161.9209    174.5622    190.1134    228.9798
```

7.4 Working with scripts

A script is a file of **R** commands. By convention the filenames of **R** scripts have extention `R` or `r`.

The command

```
source("myair.R")
```

runs the commands in file `myair.R` one after the other, (almost) as if they were typed on the command line. If **R** complains that the file does not exist, give the full path, e.g.

```
source("P:/myair.R")
```

(assuming `myair.R` is in `P:\`). In **R** commands use forward slash, /, not backslash, \, in names of folders. (Actually, on Windows you may use the backslash but it may give you unpleasant surprises occasionally since you need to type two backslashes before some symbols.)

I recommend you to create or edit scripts either using **R**'s built-in script editor (accessible through `File->New script` or `File->Open script`) or with a simple text editor, such as Notepad, that saves files in plain text.

Using scripts is a basic skill in most statistical systems and you should be able to use them even if you never write scripts yourself.

It is very common to do some calculations and then to decide to collect them in a script for future use. You can copy and paste commands from the Command window but you will need to delete the leading `>` and any output they generated. A better way is to use the command

```
history()
```

to get a list of the 25 or so most recent commands. To get more past commands, use the optional argument `max.show`, e.g.

```
history(max.show = Inf)
```

7.4.1 Editing scripts

R scripts are plain text files and can be created and edited with any editor capable of producing such files. Do not use MS Word or similar to create R scripts.

On Windows, Rgui provides an internal editor for creating, editing and running scripts. Use it if you do not have another favourite.

7.5 Basic plotting

For two dimensional plots you can use `plot(x, y)`, where `x` contains the x-coordinates and `y` the corresponding y-coordinates, see Fig. 1. To add to an existing plot, use `points()` or `lines()`, instead of `plot()`, see Fig. 2.

```
x <- seq(-5, 5, length.out = 40)
y <- x ^ 2
plot(x, y)
```

```
plot(x, y)  # x,y - as on left plot
points(x, sqrt(abs(x)), col = "brown")
lines(x, abs(x), col = "blue")
```

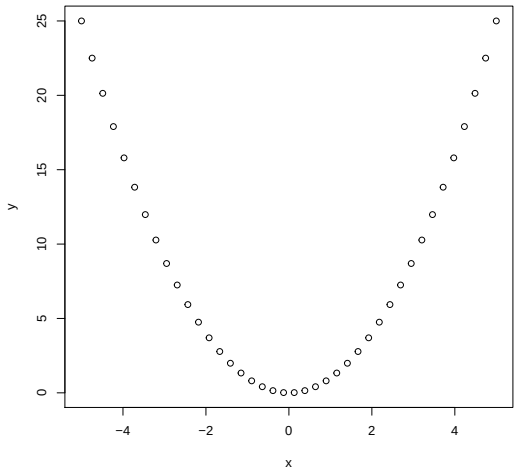


Figure 1: x^2

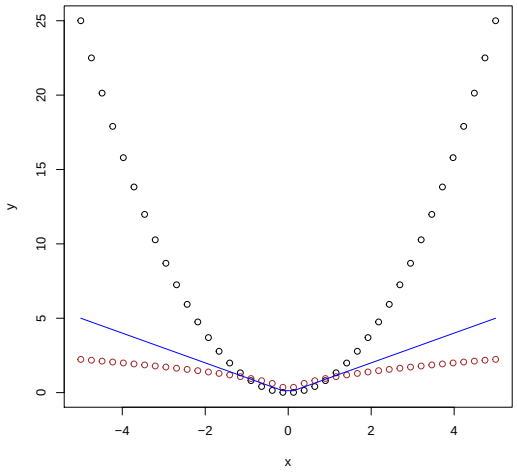


Figure 2: x^2 (black), $|x|$ (blue), $\sqrt{|x|}$ (brown)

Let me emphasise that the quickest way to get information about a particular function is through the `help` command which can be abbreviated to `?` (question mark). For example,

```
help(plot)
```

or

```
?plot
```

will create a window with information about the `plot` function. Do not get discouraged by the wealth of information, most of it is needed only if you need fine control, and even then you normally can use the examples as templates. Scroll down to the end of the help window about `plot` and look at the examples. Copy, for example, the command `plot(sin, -pi, 2 * pi)`, paste it into the `R console` window, and, surprise-surprise, a plot of the sine function in the interval $(-\pi, 2\pi)$ pops up, see Fig. 3.

Not interested in mathematical functions (having passed your calculus exam in Year 1)? Plots of data are even easier! A plot of the data in dataset `cars` is shown in Figure 4. The `cars` dataset comes with **R**, use the command `help(cars)` (or `?cars`) to see its description. To get a summary use

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
```

```
plot(sin, -pi, 2 * pi)
```

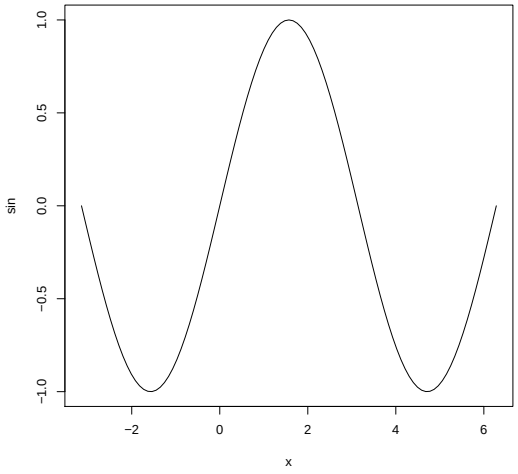


Figure 3: The sine function

```
plot(cars)
```

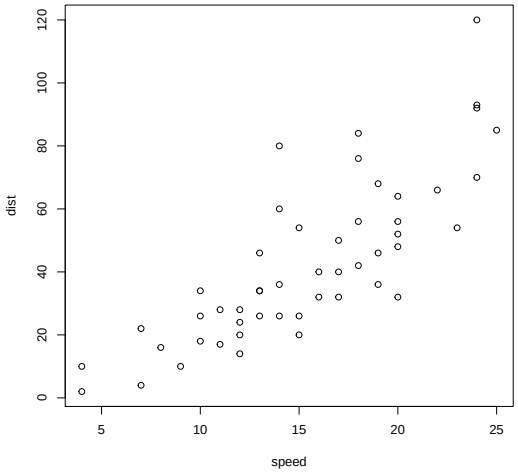


Figure 4: Cars data

```
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

Another dataset is the time series `AirPassengers`. Plots of the data (Figure 5) and the sample autocorrelation function (Figure 6) can be obtained by the commands

```
plot(AirPassengers)
plot(acf(AirPassengers))
```

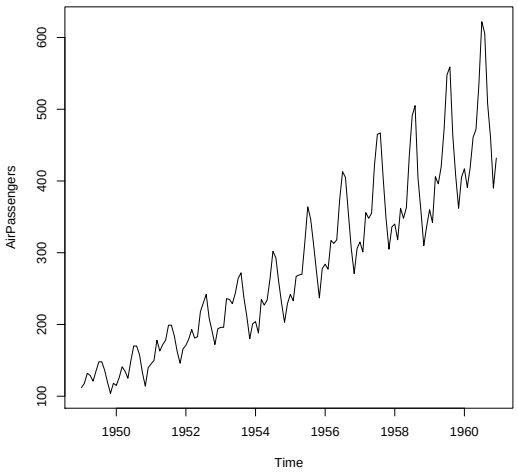


Figure 5: Airline data

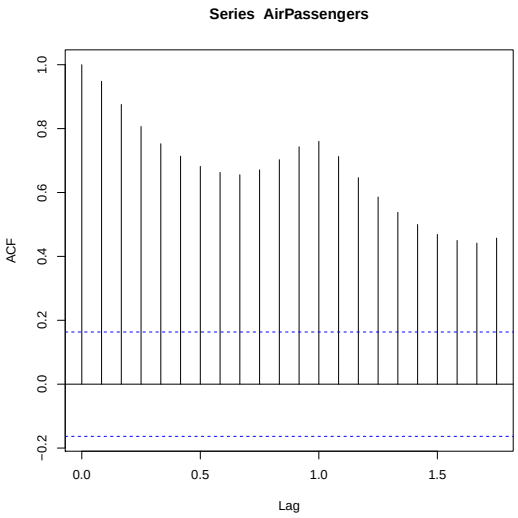


Figure 6: Acf of airline passengers data

It is not essential to know what time series and autocorrelation functions are. What is important is that the above graphs are quite different even though they were produced by simply calling the function `plot` with an argument. This is because the function `plot` "realises" that different kinds of objects should be drawn differently, and takes appropriate action. In the `plot` commands shown above the argument to `plot` was: (i) a function (`sin`), (ii) a dataset with two variables (`cars`), (iii) a time series (`AirPassengers`), and (iv) an autocorrelation function (that of `AirPassengers`).

Many functions in **R** are kind enough to behave in this way, besides `plot` the most notable ones are `print` and `summary`. In the **R** jargon such functions are called *generic*.

Of course, you may change details of the plot yourself, or even take complete charge of it. by using additional arguments in the call to `plot`, see Section 9.

7.6 Including R results in reports

You may copy and paste results from **R** into your word processing application. You can also click on a graph, then right-click and choose one of the offered options. For copying and saving, prefer "`as metafile`" to "`as bitmap`".

I prefer to save graphics files programmatically. For example, to create a file `sinplot.pdf` containing a plot of the sin() function you could use the following commands:

```
pdf("sinplot.pdf")      # store following graphs in file sinplot.pdf
plot(sin, -pi, 2 * pi)  # plot and store a graph
dev.off()               # stop storing graphs in sinplot.pdf
```

7.7 Number of digits in printed numbers

By default **R** prints numbers with 7 digits precision. You can change this using the function `options`. For example,

```
pi                    # print pi with the default precision
options(digits = 4)     # change the precision to 4 digits
pi                    # from now on R prints 4 digits

options(digits = 7)     # restore the default 7 digits
```

Individual numbers can be formatted with `format`, as in

```
format(pi, digits = 14)

## [1] "3.1415926535898"

format(pi, digits = 4)

## [1] "3.142"
```

This and related functions have a lot of optional parameters for fine control. you can start with the help page of `format` when you need special features.

8 Named arguments of functions

In a function call like the following

```
seq(1, 10, 2)

## [1] 1 3 5 7 9
```

R recognises that the first argument refers to the start of the required sequence, the second to the end and the third to the difference between successive numbers. Thus, the meaning of the arguments is determined by their position in the call.

R allows functions to be used with named arguments or a mixture of named and positional arguments. A named argument is given in the form `name=value`, e.g. the above example could be given also as

```
seq(from = 1, to = 10, by = 2)

## [1] 1 3 5 7 9
```

or

```
seq(from = 1, by = 2, to = 10)

## [1] 1 3 5 7 9
```

i.e. the order of the named arguments does not matter. This is extremely useful for functions with many arguments since with named arguments it is easy to specify only those that are of interest to us. In fact, even the function `seq` has other arguments which come in handy occasionally. For example,

```
seq(1, length.out = 10, by = 2)

##  [1]  1  3  5  7  9 11 13 15 17 19
```

will create a vector of 10 odd numbers starting with one.

The function `matrix` has a useful named argument `byrow`, see section 5.4 for its usage.

9 More on plotting

The plot function has many options. Fortunately, the meaning of many of them is obvious. Unfortunately, the meaning of many others is not. It may be difficult to find out how some things can be done. This section points out some useful bits.

9.1 The par function

Some properties of plots can be set separately with the function `par`. The style of usage is that you first call `par` to set some special properties of subsequent plots, then draw the plots and finally restore the original properties (the last step is optional but it helps in avoiding frustrating surprises in subsequent plots). I will illustrate the method with one of the often used features.

It is often desirable to arrange several plots in an array. The command `par(mfrow=c(2,2))` instructs **R** to split the plotting area into an array of 2 rows and 2 columns and to draw subsequent figures in this array filling the cells *by row* (`mfcol` is similar but fills *by column*). Figures 7-8 demonstrate the usage. The misterious `op` variable is used to store the previous state of the graphics parameters. Thus, `par(op)` restores them when the effect of the `mfrow` command is no longer needed.

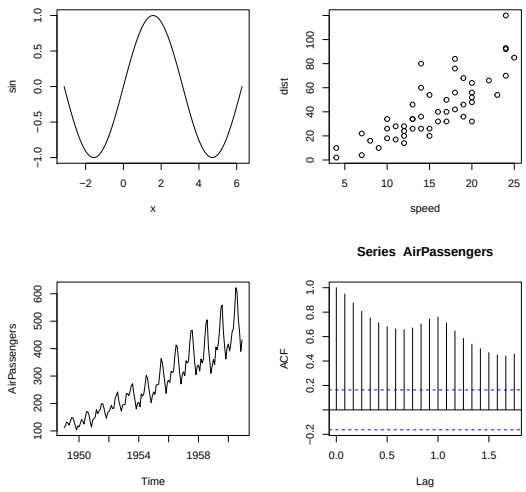
9.2 Creating new windows for plots

Plotting commands normally erase the previous plot or add to it. To create a completely new plot without destroying an existing one, issue the command

```
dev.new()
```

This creates a new window ("graphics device" in **R** jargon). Any subsequent plots will go there. The example in section 7.6 shows how to redirect the graphics to a file.

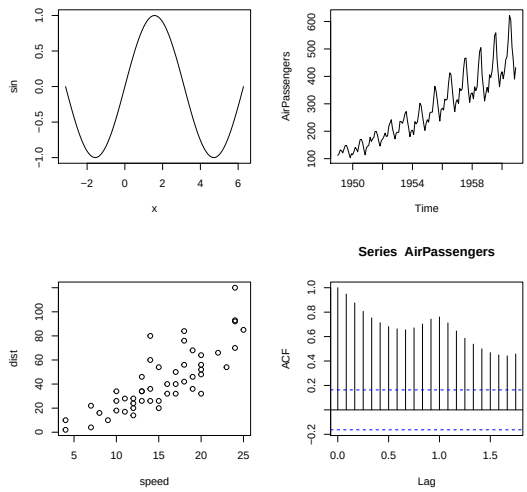
```
op <- par(mfrow=c(2,2))
plot(sin, -pi, 2 * pi)
plot(cars)
plot(AirPassengers)
acf(AirPassengers)
```




```
par(op)
```

Figure 7: Using "mfrow"

```
op <- par(mfcol=c(2,2))
plot(sin, -pi, 2 * pi)
plot(cars)
plot(AirPassengers)
acf(AirPassengers)
```




```
par(op)
```

Figure 8: Using "mfcol"

10 Importing data into R

These days most data is available in electronic form. The command `read.table` or its relatives, like `read.csv`, is suitable in most cases. For data data which is not in tabular format function `scan` may come handy.

It comes without saying that you should never copy and paste values from an application or web page into your R session. If you really think that this is necessary, put the data in a file and import it with the recommended functions.

10.1 Spreadsheet-like data editing

You may use the command `fix` to enter or edit values in a data frame or matrix, e.g.

```
fix(z)
```

will open a spreadsheet-like window where you can change the values in `z` and add additional rows or columns if needed. Make sure that the argument is a matrix or data frame since otherwise the window opened by **R** will not be a spreadsheet. To start a new data frame, first create an empty data frame and then call `fix`,

```
z <- data.frame()       # create an empty data frame
fix(z)                  # edit it---add variables and data.
```

The same functionality is available from the `Data editor` item in the `Edit` menu.

This command is useful for minor corrections in data. Even then, it is better to correct the values in the source file from which the data was imported and import the file again. It is very easy to forget that the data was wrong and import it again later.

10.2 The read.xxx family of functions

The most commonly used function for import of tabular data is `read.table()` and its variants, such as `read.csv` for comma separated files, consult the documentation for details.

After reading in the data check if you have what you expect, e.g.:

```
df <- read.table("mydata.txt", header = TRUE) # or header = FALSE, as appropriate
head(df)
summary(df)
```

The most common error is wrong argument `header`. If the first row in the file contains names of variables, use argument `header = TRUE`, otherwise `header = FALSE`.

The above command assumes that the file `mydata.txt` is in the current working directory. Otherwise you need to specify the path to it, something like `P:/data/mydata.txt`. If you don't know what `path` means, put the file in your working directory, see Section 2.2.

Another common source of error is the delimiter. By default `read.table()` assumes that the data values are separated by empty space. If they are separated by another symbol, specify it with argument `sep`, e.g.

```
df <- read.table("mydata.txt", header = TRUE, sep = ",") # values separated by comma
```

If your data are in an Excel file, export them to a csv (comma separated file) or a tab delimited file and use `read.csv` or `read.delim`, respectively.

The **R** package `foreign` provides functions for import of data in various format, e.g.:

```
library("foreign")
d <- read.mtp("mydata.mtp")     # import a Minitab data file
```

10.3 The function "scan"

10.3.1 Importing data from a file using function "scan"

Suppose that file `mydata.txt` has the following content

```
1 2
3  4 5
6
```

Then `scan("file.txt")` will create the vector

```
## [1] 1 2 3 4 5 6
```

Notice that the values are read "by row", do not need to be in tabular form and the only requirement is that they are separated by one or more blanks.

If file `t.txt` contains a matrix of numbers, say 5 rows, each containing 2 numbers, you can import them in a matrix `w` using, for example,

```
tmp <- scan("t.txt")
w <- matrix(tmp, 5, 2, byrow = TRUE)
```

It is possible to do this with a single command and, in addition, let **R** count the number of rows:

```
w1 <- matrix(scan("t.txt"), ncol = 2, byrow = TRUE)
```

10.3.2 Interactive use of the function "scan"

This method is strongly discouraged. It is always better to put the data in a file and import the file.

The `scan` function can be used to enter data from files or interactively. Its simplest form is

```
v <- scan()
```

R will prompt you with `1:`, meaning that it is waiting for the first data value. Each time you press enter it will print `n:`, meaning that it is waiting for the nth value. Enter your data and leave a blank line to tell **R** that all the data have been entered. After that the screen may look something like:

```
v <- scan()
1: 1 2 3
4: 8 7 6
7:
```

The value of `v` is

```
[1] 1 2 3 8 7 6
```

You may now cast it into a matrix using, e.g.

```
matrix(v, 2, 3, byrow = TRUE)

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    8    7    6
```

11 Using packages

Many individuals and organisations have contributed addtional functions that extend the functionality of the core **R**. These functions are usually provided as packages. To use the functions in a package you issue the command `library`, giving the name of the package as argument. For example,

```
library("foreign")
```

loads package "foreign" which provides functions for importing data from other statistical systems. One of these functions is `read.mtp` which imports data from Minitab, see the previous section.

11.1 Installing packages

If **R** complains that a package is not available when you issue a `library` command, you need to install it. To install package `pkgname` run

```
install.packages("pkgname")
```

Alternatively, choose `Packages->Install package(s)...` from the menu, select a site for your download from the list of CRAN mirrors, highlight the packages you need when presented with a list, and click OK.

If you have a binary package on your computer, choose

```
Packages->Install package(s) from local zip files...
```

navigate to the directory holding the package and select it.

Consider security implications: packages from the major repositories, CRAN and Bioconductor, are safe. They are also subject to quality control and so are likely to be reliable. Packages obtained from or recommended by a lecturer teaching you a course are likely to be safe, as well.

12 Basic programming

You can do many computations and statistical analyses in **R** without writing a single function or loop. On the other hand, even basic usage of these tools can make your work more efficient and less error prone.

12.1 Writing your own functions

You can easily create your own functions. For example, the following command creates a function, `skewness`, that calculates the sample skewness of a sample.

```
skewness <- function(x) sum( (x - mean(x)) ^ 3 / (length(x) - 1) ) / sd(x)^3
```

You can call `skewness` like any other **R** function, e.g.

```
a <- rnorm(100)  # create a random sample from N(0,1)
skewness(a)

## [1] -0.3494821
```

The above example uses a single statement to implement one of the formulae for sample skewness. If you need more than one statement, put an opening curly brace before the first statement and a closing brace after the last statement. Put each statement on a line of its own or put semicolon between statements that are on the same line. The value computed by the last statement will be returned as the value of the function.

For example, the following definition is equivalent to the above one and maybe somewhat more readable.

```
skewness1 <- function(x){
    n <- length(x)
    xbar <- mean(x)
    s <- sd(x)
    sum((x - xbar) ^ 3 / (n - 1)) / s^3
}
```

You could cram (not recommended) some or all statements on a line, separating them with semicolons, e.g.

```
skewness2 <- function(x){
    n <- length(x);  xbar <- mean(x); s <- sd(x)  # bad layout
    sum((x-xbar)^3 / (n-1)) / s^3
}
```

Try

```
skewness(a)
skewness1(a)
skewness2(a)
```

12.2 Loops

To repeat a set of commands when a variable, say i, takes a sequence of values, say $1, 2, \ldots, 10$, you can use a `for` loop. For example, the following calculates the sum $\sum_{i=1}^{10} i^2$.

```
s10 <- 0
for(i in 1:10){
    s10 <- s10 + i^2
}
s10

## [1] 385
```

And the following computes the skewness of the vector `a`, created above.

```
n <- length(a);
abar <- mean(a);
s <- sd(a)
sk <- 0
for(i in 1:n){
   sk <- sk + (a[i]-abar)^3
}
sk <- sk / (n-1)
sk <- sk / s^3
```

Notice that in these (and similar) examples the loops can be replaced by single statements, such as `sum((1:10)^2)` and `sk <- sum( (a-abar)^3 )`. The following example would be difficult without loops.

```
# first 10 Fibonacci numbers
n <- 10
fib <- numeric(n)       # create a vector to hold the generated sequence
fib[1] <- 1             # set the first couple of values
fib[2] <- 1
for(i in 3:n){          # fill the rest using the familiar formula.
    fib[i] <- fib[i-1] + fib[i-2]
}
fib

##  [1]  1  1  2  3  5  8 13 21 34 55
```

Sometimes the number of iterations is not known in advance. In such cases you can use the `while(expr)` loop. The loop is repeated until `expr` evaluates to FALSE. The following example finds the solution of the equation $\sin(x) - 0.25 = 0$ in the interval $[0, \pi/2]$ using a bisection method. At each iteration the value of the function is evaluated at the middle point of the current interval. The interval is narrowed by changing `xmin` or `xmax`, depending on the value of the function at the new point. The calculation is repeated while the difference between the value of the function and the target value is larger than 10^{-4}. Notice that, to avoid the possibility that the loop will be run forever, we terminate the loop if it is repeated too many times (here, 100).

```
niter <- 0
xmin <- 0
xmax <- pi/2
crit <- 1
while(crit > 1e-4  &&  niter < 100){
    niter <- niter + 1
    xcur <- (xmin + xmax) / 2
    fval <- sin(xcur)
    if(fval >= 0.25)
        xmax <- xcur
    else
        xmin <- xcur

    crit <- abs(fval - 0.25)
}

xcur # the solution

## [1] 0.2527233

niter

## [1] 12
```

In practice, such a calculation is best put in a function with arguments including `xmin`, `xmax`, `niter` and the threshold for declaring convergence (here it was 10^{-4}).

12.3 Conditional statements (if-else)

To execute a piece of code only if certain conditions are met, create a variable, say `bool` which holds a **single logical value**: `TRUE` if the condition holds, `FALSE` otherwise. Then use the folloing construct:

```
if(bool) {
    statements to execute if bool is TRUE
} else {
    statements to execute if bool is FALSE
}
```

The else part can be omitted if not needed. Also, notice the emphasis on the word 'single' above.

You can omit the curly braces in the "if" part if it contains a single statement and similarly for the "else" part.

However, forgetting the braces when there is more than one statement is a typical cause of errors. for this reason and for consistency, many **R** users always use the braces.

As an example of a typical use of conditional statements, let's compute the sum of the geometric progression, $\sum_{i=0}^{\infty} x^k$ using the fact that it is equal to $1/(1-x)$, provided that $x \in (-1, 1)$. We don't want to return a wrong result if the user supplies a value of x outside this interval, so I stop the execution of the function with an error message if this is the case.

```
geomsum <- function(x) {
   if( any(x <= -1) || any(x >= 1) ) # are there values outside (-1,1)?
       stop("all values in 'x' must be in (-1, 1)")
   1 / (1 - x)
}

geomsum(0.25)

## [1] 1.333333

geomsum(c(0.1, 0.25, 0.75))

## [1] 1.111111 1.333333 4.000000
```

Some examples of calling `geomsum`:

```
geomsum(c(0.1, 0.25, 2, 0.75))
```

The function works also if `x` is a vector. In that case it throws error whenever at least one of the values in `x` is outside the interval $(-1, 1)$.

12.4 Going a little bit beyond the basics

You can transform the commands for the sum of the squares of the first 10 numbers into a function, e.g.

```
# sum of squares of the first n natural numbers
#   example usage:  nsq(10); nsq(20)
nsq <- function(n){
    s <- 0
    for(i in 1:n){
        s <- s + i^2
    }
    s   # the value of the last statement is the value of the function
}
```

By adding an additional argument, you easily modify `nsq` to find sums of any powers, not necessarily squares.

```
# sum of squares of the first n natural numbers
#   example usage:  sumnk(10, 2); sumnk(10, 3)
sumnk <- function(n, k){
    s <- 0
    for(i in 1:n){
        s <- s + i^k
    }
    s
}
```

If you think that $k = 2$ will be used in most cases, you can make it the default as follows.

```
# sum of k-th powers of the first n natural numbers
#   example usage:  sumnkd(10); sumnkd(10, 2); sumnk(10, 3)
sumnkd <- function(n, k = 2){          # k = 2 designates 2 as default value of k
    s <- 0
    for(i in 1:n){
        s <- s + i^k
    }
    s
}
```

As an exercise, you could rewrite these functions without "for" loops, analogously to the skewness examples at the beginning of the section.

12.5 Reusing your functions

When you have written one or more functions and checked that they work, you can put them in a file, say `myfuns.R`, and use the command `source("myfuns.R")` to make them available in your **R** sessions.

12.6 Formattting R code

R does not care how the code is formatted but good formatting helps human readers and helps in writing correct code. In particular, it is a good idea to format the source of your functions following the style recommended by the R core team. Type the name of any **R** function, e.g. `cor`, to see an example. Usually each command should be on its own line, nested structures (like `if` and `for`) should be indented, arithmetic and other binary operations (including assignments) should be surrounded by a space on both sides. You will go along way by using formatting as in the examples in this document (except those indicated to be bad, of course).

These days **R** usually is set up to print function definitions the way we formatted them, try

```
skewness2

## function(x){
##     n <- length(x);  xbar <- mean(x); s <- sd(x)  # bad layout
##     sum((x-xbar)^3 / (n-1)) / s^3
## }

print(skewness2)

## function(x){
##     n <- length(x);  xbar <- mean(x); s <- sd(x)  # bad layout
##     sum((x-xbar)^3 / (n-1)) / s^3
## }
```

A quick way to see how your function would look like in the recommended style, use funtion `print` with argument `useSource`. For `skewness2` defined earlier we get:

```
print(skewness2, useSource = FALSE)

## function (x)
## {
##     n <- length(x)
##     xbar <- mean(x)
##     s <- sd(x)
##     sum((x - xbar)^3/(n - 1))/s^3
## }
```

Notice that now all expressions are on separate lines, compare to the original definition of `skewness2` above. Note though that the rendering is not ideal (some of the operations on the last line still look crammed) and for more complex code may not be particularly readable.

You can find excellent advice on formatting **R** code at http://adv-r.had.co.nz/Style.html.