Pedro J. Aphalo

Learn R

As a Language

Contents

List of Figures

1

The R language: "Paragraphs" and "essays"

> An R script is simply a text file containing (almost) the same commands that you would enter on the command line of R.
>
> Jim Lemon
> *Kickstarting R*

1.1 Aims of this chapter

For those who have mainly used graphical user interfaces, understanding why and when scripts can help in communicating a certain data analysis protocol can be revelatory. As soon as a data analysis stops being trivial, describing the steps followed through a system of menus and dialogue boxes becomes extremely tedious.

Moreover, graphical user interfaces tend to be difficult to extend or improve in a way that keeps step-by-step instructions valid across program versions and operating systems.

Many times, exactly the same sequence of commands needs to be applied to different data sets, and scripts make both implementation and validation of such a requirement easy.

In this chapter, I will walk you through the use of R scripts, starting from an extremely simple script.

1.2 Writing scripts

In R language, the closest match to a natural language essay is a script. A script is built from multiple interconnected code statements needed to complete a given task. Simple statements can be combined into compound statements, which are the equivalent of natural language paragraphs. Scripts can vary from simple scripts containing only a few code statements, to complex scripts containing hundreds of

code statements. In the rest of the present section I discuss how to write readable and reliable scripts and how to use them.

1.2.1 What is a script?

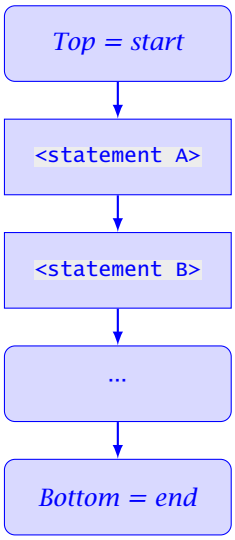
A *script* is a text file that contains (almost) the same commands that you would type at the console prompt. A true script is not, for example, an MS-Word file where you have pasted or typed some R commands.

When typing commands/statements at the R console, we "feed" one line of text at a time. When we end the line by typing the enter key, the line of text is interpreted and evaluated. We then type the next line of text, which gets in turn interpreted and evaluated, and so on. In a script we write nearly the same text in an editor and save multiple lines containing commands into a text file. Interpretation takes place only later, when we *source* the file as a whole into R.

A script file has the following characteristics.

- The script is a text file.
- The file contains valid R statements (including comments) and nothing else.
- Comments start at a `#` and end at the end of the line.
- The R statements are in the file in the order that they must be executed.
- R scripts have file names ending in `.r` or `.R`.

The statements in the text file, are read, interpreted and evaluated sequentially, from the start to the end of the file, as represented in the diagram. We use ··· to represent additional statements in the script.



As we will see later in the chapter, code statements can be combined into larger statements and evaluated conditionally and/or repeatedly, which allows us to control the realised sequence of evaluated statements. Scripts need to respect the R

syntax. In addition to being valid it is important that scripts are also understandable to humans, consequently a clear writing style and consistent adherence to it are important.

It is good practice to write scripts so that they are self-contained. To make a script self-contained, one must include calls to `library()` to load the packages used, load or import data from files, perform the data analysis and display and/or save the results of the analysis. Such scripts can be used to apply the same analysis algorithm to other data and/or to reproduce the same analysis at a later time. Such scripts document all steps used for the analysis.

1.2.2 How do we use a script?

A script can be "sourced" using function `source()`. If we have a text file called `my.first.script.r` containing the following text:

```
# this is my first R script
print(3 + 4)
```

and then source this file:

```
source("my.first.script.r")
## [1] 7
```

The results of executing the statements contained in the file will appear in the console. The commands themselves are not shown (by default the sourced file is not *echoed* to the console) and the results will not be printed unless you include explicit `print()` commands in the script. This applies in many cases also to plots—e.g., a figure created with `ggplot()` needs to be printed if we want it to be included in the output when the script is run. Adding a redundant `print()` is harmless.

From within RStudio, if you have an R script open in the editor, there will be a "source" icon visible with an attached drop-down menu from which you can choose "Source" as described above, or "Source with echo," or "Source as local job" for the script in the currently active editor tab.

When a script is *sourced*, the output can be saved to a text file instead of being shown in the console. It is also easy to call R with the R script file as an argument directly at the operating system shell or command-interpreter prompt—and obviously also from shell scripts. The next two chunks show commands entered at the OS shell command prompt rather than at the R command prompt.

```
> RScript my.first.script.r
```

You can open an operating system's *shell* from the Tools menu in RStudio, to run this command. The output will be printed to the shell console. If you would like to save the output to a file, use redirection using the operating system's syntax.

```
> RScript my.first.script.r > my.output.txt
```

Sourcing is very useful when the script is ready, however, while developing a

script, or sometimes when testing things, one usually wants to run (or *execute*) one or a few statements at a time. This can be done using the "run" button[1] after either positioning the cursor in the line to be executed, or selecting the text that one would like to run (the selected text can be part of a line, a whole line, or a group of lines, as long as it is syntactically valid). The key-shortcut Ctrl-Enter is equivalent to pressing the "run" button in RStudio.

1.2.3 How to write a script

As with any type of writing, different approaches may be preferred by different R users. In general, the approach used, or mix of approaches, will also depend on how confident you are that the statements will work as expected—you already know the best approach vs. you are exploring different alternatives.

If one is very familiar with similar problems One would just create a new text file and write the whole thing in the editor, and then test it. This is rather unusual.

If one is moderately familiar with the problem One would write the script as above, but testing it, step by step, as one is writing it. This is usually what I do.

If one is mostly playing around Then if one is using RStudio, one can type statements at the console prompt. As you should know by now, everything you run at the console is saved to the "History." In RStudio, the History is displayed in its own pane, and in this pane one can select any previous statement(s) and by clicking on a single icon, copy and paste them to either the R console prompt, or the cursor position in the editor pane. In this way one can build a script by copying and pasting from the history to your script file, the bits that have worked as you wanted.

 By now you should be familiar enough with R to be able to write your own script.

1. Create a new R script (in RStudio, from the File menu, "+" icon, or by typing "Ctrl + Shift + N").
2. Save the file as `my.second.script.r`.
3. Use the editor pane in RStudio to type some R commands and comments.
4. *Run* individual commands.
5. *Source* the whole file.

1.2.4 The need to be understandable to people

When you write a script, it is either because you want to document what you have done or you want re-use the script at a later time. In either case, the script itself

[1] If you use a different IDE or editor with an R mode, the details will vary, but a run command will be usually available.

although still meaningful for the computer, could become very obscure to you, and even more to someone seeing it for the first time. This must be avoided by spending time and effort on the writing style.

How does one achieve an understandable script or program?

- Avoid the unusual. People using a certain programming language tend to use some implicit or explicit rules of style—style includes *indentation* of statements, *capitalization* of variable and function names. As a minimum try to be consistent with yourself.
- Use meaningful names for variables, and any other object. What is meaningful depends on the context. Depending on common use, a single letter may be more meaningful than a long word. However self-explanatory names are usually better: e.g., using `n.rows` and `n.cols` is much clearer than using `n1` and `n2` when dealing with a matrix of data. Probably `number.of.rows` and `number.of.columns` would make the script verbose, and take longer to type without gaining anything in return.
- How to make the words visible in names: traditionally in R one would use dots to separate the words and use only lower case. Some years ago, it became possible to use underscores. The use of underscores is quite common nowadays because in some contexts it is "safer", as in some situations a dot may have a special meaning. What we call "camel case" is only infrequently used in R programming but is common in other languages like Pascal. An example of camel case is `NumCols`.

Here is an example of bad style in a script. Read *Google's R Style Guide* (`https://google.github.io/styleguide/Rguide.xml`), and edit the code in the chunk below so that it becomes easier to read.

```
a <- 2 # height
b <- 4 # length
C <-
    a *
b
C -> variable
      print(
"area: ", variable
)
```

The points discussed above already help a lot. However, one can go further in achieving the goal of human readability by interspersing explanations and code "chunks" and using all the facilities of typesetting, even of formatted maths formulas and equations, within the listing of the script. Furthermore, by including the results of the calculations and the code itself in a typeset report built automatically, we can ensure that the results are indeed the result of running the code shown. This greatly contributes to data analysis reproducibility, which is becoming a widespread requirement for any data analysis both in academia and in industry. It is possible not only to typeset whole books like this one, but also whole data-based web sites with these tools.

In the realm of programming, this approach is called literate programming and was first proposed by Donald Knuth (Knuth 1984) through his WEB system. In the case of R programming, the first support of literate programming was through 'Sweave', which has been mostly superseded by 'knitr' (Xie 2013). This package supports the use of Markdown or LaTeX (Lamport 1994) as the markup language for the textual contents and also formats and adds syntax highlighting to code chunks. Rmarkdown is an extension to Markdown that makes it easier to include R code in documents (see http://rmarkdown.rstudio.com/). It is the basis of R packages that support typesetting large and complex documents ('bookdown'), web sites ('blogdown'), package vignettes ('pkgdown') and slides for presentations (Xie 2016; Xie et al. 2018). The use of 'knitr' is very well integrated into the RStudio IDE.

This is not strictly an R programming subject, as it concerns programming in any language. On the other hand, this is an incredibly important skill to learn, but well described in other books and web sites cited in the previous paragraph. This whole book, including figures, has been generated using 'knitr' and the source code for the book is available through Bitbucket at https://bitbucket.org/aphalo/learnr-book.

1.2.5 Debugging scripts

The use of the word *bug* to describe a problem in computer hardware and software started in 1946 when a real bug, more precisely a moth, got between the contacts of a relay in an electromechanical computer causing it to malfunction and Grace Hooper described the first computer *bug*. The use of the term bug in engineering predates the use in computer science, and consequently, the first use of bug in computing caught on easily because it represented an earlier-used metaphor becoming real.

A suitable quotation from a letter written by Thomas Alva Edison 1878 (as given by Hughes 2004):

> It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise–this thing gives out and [it is] then that "Bugs"–as such little faults and difficulties are called–show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.

The quoted paragraph above makes clear that only very exceptionally does any new design fully succeed. The same applies to R scripts as well as any other non-trivial piece of computer code. From this it logically follows that testing and debugging are fundamental steps in the development of R scripts and packages. Debugging, as an activity, is outside the scope of this book. However, clear programming style and good documentation are indispensable for efficient testing and reuse.

Even for scripts used for analyzing a single data set, we need to be confident that the algorithms and their implementation are valid, and able to return correct results. This is true both for scientific reports, expert data-based reports and any data analysis related to assessment of compliance with legislation or regulations. Of course, even in cases when we are not required to demonstrate validity, say

for decision making purely internal to a private organization, we will still want to avoid costly mistakes.

The first step in producing reliable computer code is to accept that any code that we write needs to be tested and, if possible, validated. Another important step is to make sure that input is validated within the script and a suitable error produced for bad input (including valid input values falling outside the range that can be reliably handled by the script).

If during testing, or during normal use, a wrong value is returned by a calculation, or no value (e.g., the script crashes or triggers a fatal error), debugging consists in finding the cause of the problem. The cause can be either a mistake in the implementation of an algorithm, as well as in the algorithm itself. However, many apparent *bugs* are caused by bad or missing handling of special cases like invalid input values, rounding errors, division by zero, etc., in which a program crashes instead of elegantly issuing a helpful error message.

Diagnosing the source of bugs is, in most cases, like detective work. One uses hunches based on common sense and experience to try to locate the lines of code causing the problem. One follows different *leads* until the case is solved. In most cases, at the very bottom we rely on some sort of divide-and-conquer strategy. For example, we may check the value returned by intermediate calculations until we locate the earliest code statement producing a wrong value. Another common case is when some input values trigger a bug. In such cases it is frequently best to start by testing if different "cases" of input lead to errors/crashes or not. Boundary input values are usually the telltale ones: e.g., for numbers, zero, negative and positive values, very large values, very small values, missing values (`NA`), vectors of length zero (`numeric()`), etc.

Error messages When debugging, keep in mind that in some cases a single bug can lead to a whole cascade of error messages. Do also keep in mind that typing mistakes, originating when code is entered through the keyboard, can wreak havock in a script: usually there is little correspondence between the number of error messages and the seriousness of the bug triggering them. When several errors are triggered, start by reading the error message printed first, as later errors can be an indirect consequence of earlier ones.

There are special tools, called debuggers, available, and they help enormously. Debuggers allow one to step through the code, executing one statement at a time, and at each pause, allowing the user to inspect the objects present in the R environment and their values. It is even possible to execute additional statements, say, to modify the value of a variable, while execution is paused. An R debugger is available within RStudio and also through the R console.

When writing your first scripts, you will manage perfectly well, and learn more by running the script one line at a time and when needed temporarily inserting `print()` statements to "look" at how the value of variables changes at each step. A debugger allows a lot more control, as one can "step in" and "step out" of function definitions, and set and unset break points where execution will stop, which is especially useful when developing R packages.

When reproducing the examples in this chapter, do keep this section in mind. In addition, if you get stuck trying to find the cause of a bug, do extend your search

both to the most trivial of possible causes, and to the least likely ones (such as a bug in a package installed from CRAN or R itself). Of course, when suspecting a bug in code you have not written, it is wise to very carefully read the documentation, as the “bug” may be just in your understanding of what a certain piece of code is expected to do. Also keep in mind that as discussed on page **??**, you will be able to find online already-answered questions to many of your likely problems and doubts. For example, Googling for the text of an error message is usually well rewarded.

 When installing packages from other sources than CRAN (e.g., development versions from GitHub, Bitbucket or R-Forge, or in-house packages) there is no warranty that conflicts will not happen. Packages (and their versions) released through CRAN are regularly checked for inter-compatibility, while packages released through other channels are usually checked against only a few packages.

Conflicts among packages can easily arise, for example, when they use the same names for objects or functions. In addition, many packages use functions defined in packages in the R distribution itself or other independently developed packages by importing them. Updates to depended-upon packages can “break” (make non-functional) the dependent packages or parts of them. The rigorous testing by CRAN detects such problems in most cases when package revisions are submitted, forcing package maintainers to fix problems before distribution through CRAN is possible. However, if you use other repositories, I recommend that you make sure that revised (especially if under development) versions do work with your own script, before their use in “production” (important) data analyses.

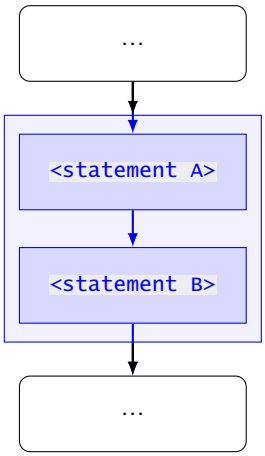
1.3 Control of execution flow

By default R statements in a script are evaluated (or executed) in the sequence they appear in the script *listing* or text. We give the name *control of execution constructs* to those special statements that allow us to alter this default sequence, by either skipping or repeatedly evaluating individual statements. The statements whose evaluation is controlled can be either simple or compound. Some of the control of execution flow statements, function like *ON-OFF switches* for program statements. Others allow statements to be executed repeatedly while or until a condition is met, or until all members of a list or a vector are processed.

These *control of execution constructs* can be also used at the R console, but it is usually awkward to do so as they can extend over several lines of text. In simple scripts, the *flow of execution* can be fixed and linear from the first to the last statement in the script. *Control of execution constructs* are a crucial part of most scripts. As we will see next, a compound statement can include multiple simple or nested compound statements.

1.3.1 Compound statements

Individual statements can be grouped into *compound statements* by enclosed them in curly braces. Conceptually is like putting several statements into a box that allows us to operate with them as an anonymous whole.




```
print("A")
## [1] "A"

{
  print("B")
  print("C")
}
## [1] "B"
## [1] "C"
```

The grouping of the last two statements above is of no consequence by itself, but grouping becomes useful when used together with control-of-execution constructs.

1.3.2 Conditional execution

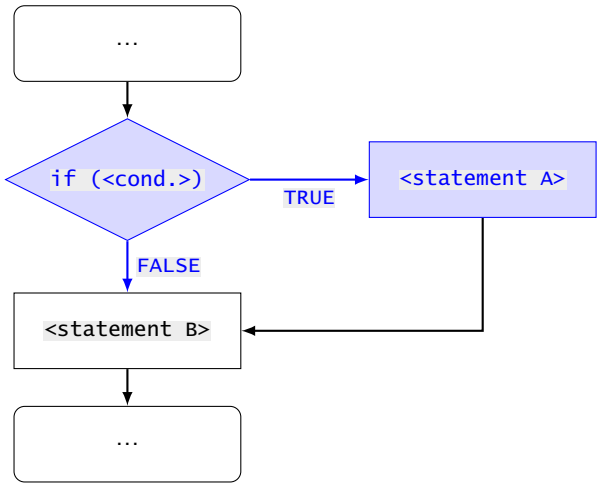
Conditional execution allows handling different values, such as negative and non-negative values, differently within a script. This is achieved by evaluating or not (i.e., switching ON and OFF) parts of a script based on the result returned by a logical expression. An R expression returning a logical value can be as simple as a logical value of `TRUE` or `FALSE` stored in a variable or the result of a computation done at the time of the flow-control decision.

We use the name *flag* for a `logical` variable set manually, preferably near the top of the script. Use of flags is most useful when switching between two script behaviors depends on multiple sections of code. A frequent use case for flags is jointly enabling and disabling printing of output from multiple statements scattered in over a long script.

R has two types of *if* statements, non-vectorized and vectorized. We will start with the non-vectorized one, which is similar to what is available in most other computer programming languages and controls the evaluation of a code statement, which can be either simple or compound.

1.3.2.1 Non-vectorized `if`, `else` and `switch`

The `if` construct "decides," depending on a `logical` value, whether the next code statement is executed (if `TRUE`) or skipped (if `FALSE`). The flow chart shows how `if` works: `<statement A>` is either evaluated or skipped depending on the value of `<condition>`, while `<statement B>` is always evaluated.



We start with toy examples demonstrating how *if* statements work. Later we will see examples closer to real use cases.

```
flag <- TRUE
if (flag) print("Hello!")
## [1] "Hello!"
```

Play with the code above by changing the value assigned to variable `flag`, `FALSE`, `NA`, and `logical(0)`.

In the example above we use variable `flag` as the *condition*.

Nothing in the R language prevents this condition from being a `logical` constant. Explain why `if (TRUE)` in the syntactically-correct statement below is of no practical use.

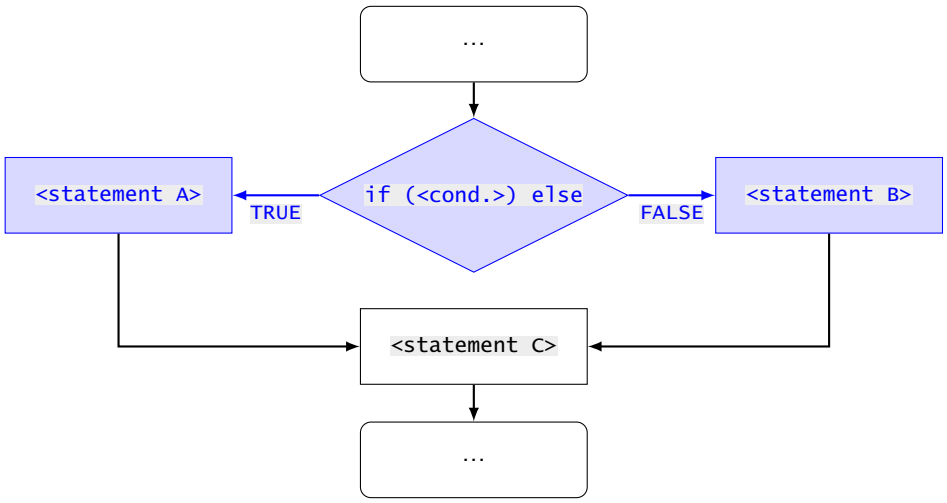
```
if (TRUE) print("Hello!")
## [1] "Hello!"
```

Conditional execution is much more useful than what could be expected from the previous example, because the statement whose execution is being controlled can be a compound statement of almost any length or complexity. A very simple example follows.

```
printing <- TRUE
if (printing) {
  print("A")
  print("B")
}
## [1] "A"
## [1] "B"
```

The condition passed as an argument to `if`, enclosed in parentheses, can be anything yielding a `logical` vector of length one. As this condition is *not* vectorized, a longer vector will trigger an R warning or error depending on R's version.

The `if … else` construct "decides," depending on a `logical` value, which of two code statements is executed. The flow chart shows how `if` works: either `<statement A>` or `<statement B>` is evaluated and the other skipped depending on the value of `<condition>`, while `<statement C>` is always evaluated.




```
a <- 10.0
if (a < 0.0) print("'a' is negative") else print("'a' is not negative")
## [1] "'a' is not negative"

print("This is always printed")
## [1] "This is always printed"
```

As can be seen above, the statement immediately following `if` is executed if the condition returns `TRUE` and that following `else` is executed if the condition returns `FALSE`. Statements after the conditionally executed `if` and `else` statements are always executed, independently of the value returned by the condition.

Play with the code in the chunk above by assigning different numeric vectors to `a`.

Do you still remember the rules about continuation lines?

```
# 1
a <- 1
if (a < 0.0) print("'a' is negative") else print("'a' is not negative")
## [1] "'a' is not negative"
```

Why does the statement below (not evaluated here) trigger an error while the one above does not?

```
# 2 (not evaluated here)
if (a < 0.0) print("'a' is negative")
else print("'a' is not negative")
```

How do the continuation line rules apply when we add curly braces as shown below.

```
# 1
a <- 1
if (a < 0.0) {
    print("'a' is negative")
  } else {
    print("'a' is not negative")
  }
## [1] "'a' is not negative"
```

In the example above, we enclosed a single statement between each pair of curly braces, but as these braces create compound statements, multiple statements could have been enclosed between each pair.

Play with the use of conditional execution, with both simple and compound statements, and also think how to combine `if` and `else` to select among more than two options.

In R, the value returned by any compound statement is the value returned by the last simple statement executed within the compound one. This means that we can assign the value returned by an `if` and `else` statement to a variable. This style is less frequently used, but occasionally can result in easier-to-understand scripts.

```
a <- 1
my.message <-
  if (a < 0.0) "'a' is negative" else "'a' is not negative"
print(my.message)
## [1] "'a' is not negative"
```

If the condition statement returns a value of a class other than `logical`, R will attempt to convert it into a logical. This is sometimes used instead of a comparison to zero, as the conversion from `integer` yields `TRUE` for all integers except zero. The code below illustrates a rather frequently used idiom for checking if there is something available to display.

```
message <- "abc"
if (length(message)) print(message)
## [1] "abc"
```

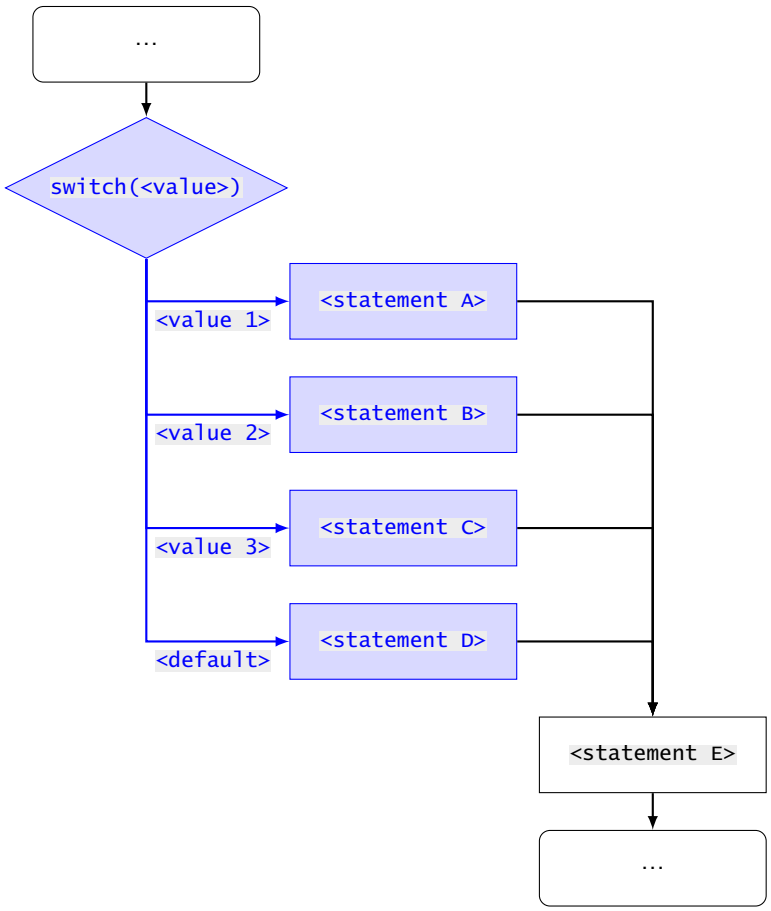
Study the conversion rules between `numeric` and `logical` values, run each of the statements below, and explain the output based on how type conversions are interpreted, remembering the difference between *floating-point numbers* as implemented in computers and *real numbers* ($\mathbb{R}$) as defined in mathematics.

```
if (0) print("hello")
if (-1) print("hello")
if (0.01) print("hello")
if (1e-300) print("hello")
if (1e-323) print("hello")
if (1e-324) print("hello")
if (1e-500) print("hello")
if (as.logical("true")) print("hello")
if (as.logical(as.numeric("1"))) print("hello")
if (as.logical("1")) print("hello")
if ("1") print("hello")
```

Hint: if you need to refresh your understanding of the type conversion rules, see section **??** on page **??**.

In addition to `if ()` and `if () … else`, there is in R a `switch()` statement, which we describe next. It can be used to select among *cases*, or several alternative statements, based on an expression evaluating to a `numeric` or a `character` value of length equal to one. While `if ()` and `if () … else` allow for binary choices as they are controlled by a logical value, `switch()` can control execution of many more statements. The usual way in which `switch()` statement is used is by assignment of the returned value, as described as being rather unusual, but legal, for `if () … else` on page 12.

The switch statement returns a value, the value returned by the `switch()` statement is that returned by the statement corresponding to the matching switch value, or the default (similar to `else`) if there is no match and a default return value has been defined. Each optional statement can be thought as a *case* from a set of possible cases.



In the first example we use character constants saved in a variable as a condition, with the last statement with no tag being the default. Instead of the name of variable `my.object`, we could have used a complex expression returning a suitable `character` value of length one.

```
my.object <- "two"
b <- switch(my.object,
            one = 1,
            two = 1 / 2,
            four = 1 / 4,
            0
)
b
## [1] 0.5
```

Multiple condition values can share the same statement.

```
my.object <- "two"
b <- switch(my.object,
            one =, uno = 1,
            two =, dos = 1 / 2,
            four =, cuatro = 1 / 4,
            0
)
```

```
b
## [1] 0.5
```

Do play with the use of the switch statement. Look at the documentation for `switch()` using `help(switch)` and study the examples at the end of the help page. Explore what happens if you set `my.object <- "ten"`, `my.object <- "three"`, `my.object <- NA_character_` or `my.object <- character()`. Then remove the `, 0` as default value, and repeat.

When the expression used as a condition returns a value that is not a `character`, it will be interpreted as an `integer` index. In this case no names are used for the cases, and the last one is always interpreted as the default.

```
my.number <- 2
b <- switch(my.number,
            1,
            1 / 2,
            1 / 4,
            0
)
b
## [1] 0.5
```

Continue playing with the use of the switch statement. Explore what happens if you set `my.number <- 10`, `my.number <- 3`, `my.number <- NA` or `my.object <- numeric()`. Then remove the `, 0` as default value, and repeat.

The statements for the different values of the condition in a `switch()` statement can be compound statements as in the case of `if`, and they can even be used for a side effect. We can for example modify the example above to print a message when the default value is returned.

```
my.object <- "ten"
b <- switch(my.object,
            one = 1,
            two = 1 / 2,
            three = 1 / 4,
            {print("No match! Using default"); 0}
)
## [1] "No match! Using default"

b
## [1] 0
```

The `switch()` statement can substitute for chained `if … else` statements when all the conditions can be described by constant values or distinct values returned by the same test. The advantage is more concise and readable code. The

equivalent of the first `switch()` example above when written using `if … else` becomes longer. Given how terse code using `switch()` is, those not yet familiar with its use may find the more verbose style used below easier to understand. On the other hand, with numerous cases `switch()` is easier to read and understand.

```
my.object <- "two"
if (my.object == "one") {
  b <- 1
} else if (my.object == "two") {
  b <- 1 / 2
} else if (my.object == "four") {
  b <- 1 / 4
} else {
  b <- 0
}
b
## [1] 0.5
```

1.3.2.2 Vectorized `ifelse()`

Vectorized *ifelse* is a peculiarity of the R language, but very useful for writing concise code that may execute faster than logically equivalent but not vectorized code. Vectorized conditional execution is coded by means of *function* `ifelse()` (written as a single word). This function takes three arguments: a `logical` vector usually the result of a test (parameter `test`), an expression to use for `TRUE` cases (parameter `yes`), and an expression to use for `FALSE` cases (parameter `no`). At each index position along the vectors, the value included in the returned vector is taken from `yes` if `test` is `TRUE` and from `no` if `test` is `FALSE`. All three arguments can be any R statement returning the required vectors. In the case of vectors passed as arguments to parameters `yes` and `no`, recycling will take place if they are shorter than the logical vector returned by the expression passed as argument to `test`. No recycling ever applies to `test`, even if `yes` and/or `no` are longer than `test`.

The flow chart for `ifelse()` is similar to that for `if … else` shown on page 10 but applied in parallel to the individual members of vectors; e.g. the condition expression is evaluated at index position `1` controls which value will be present in the returned vector at index position `1`, and so on.

It is customary to pass arguments to `ifelse` by position. We give a first example with named arguments to clarify the use of the function.

```
my.test <- c(TRUE, FALSE, TRUE, TRUE)
ifelse(test = my.test, yes = 1, no = -1)
## [1]  1 -1  1  1
```

In practice, the most common idiom is to have as an argument passed to `test`, the result of a comparison calculated on the fly. In the first example we compute the absolute values for a vector, equivalent to that returned by R function `abs()`.

```
nums <- -3:+3
ifelse(nums < 0, -nums, nums)
## [1] 3 2 1 0 1 2 3
```

Some additional examples to play with, with a few surprises. Study the examples below until you understand why returned values are what they are. In addition, create your own examples to test other possible cases. In other words, play with the code until you fully understand how `ifelse` works.

```
a <- 1:10
ifelse(a > 5, 1, -1)
ifelse(a > 5, a + 1, a - 1)
ifelse(any(a > 5), a + 1, a - 1) # tricky
ifelse(logical(0), a + 1, a - 1) # even more tricky
ifelse(NA, a + 1, a - 1) # as expected
```

Hint: if you need to refresh your understanding of `logical` values and Boolean algebra see section ?? on page ??.

In the case of `ifelse()`, the length of the returned value is determined by the length of the logical vector passed as an argument to its first formal parameter (named `test`)! A frequent mistake is to use a condition that returns a `logical` vector of length one, expecting that it will be recycled because arguments passed to the other formal parameters (named `yes` and `no`) are longer. However, no recycling will take place, resulting in a returned value of length one, with the remaining elements of the vectors passed to `yes` and `no` being discarded. Do try this by yourself, using logical vectors of different lengths. You can start with the examples below, making sure you understand why the returned values are what they are.

```
ifelse(TRUE, 1:5, -5:-1)
## [1] 1

ifelse(FALSE, 1:5, -5:-1)
## [1] -5

ifelse(c(TRUE, FALSE), 1:5, -5:-1)
## [1]  1 -4

ifelse(c(FALSE, TRUE), 1:5, -5:-1)
## [1] -5  2

ifelse(c(FALSE, TRUE), 1:5, 0)
## [1] 0 2
```

Write, using `ifelse()`, a single statement to combine numbers from the two vectors `a` and `b` into a result vector `d`, based on whether the corresponding value in vector `c` is the character `"a"` or `"b"`. Then print vector `d` to make the result visible.

```
a <- -10:-1
b <- +1:10
c <- c(rep("a", 5), rep("b", 5))
# your code
```

If you do not understand how the three vectors are built, or you cannot guess

the values they contain by reading the code, print them, and play with the arguments, until you understnd what each parameter does. Also use `help(rep)` and/or `help(ifelse)` to access the documentation.

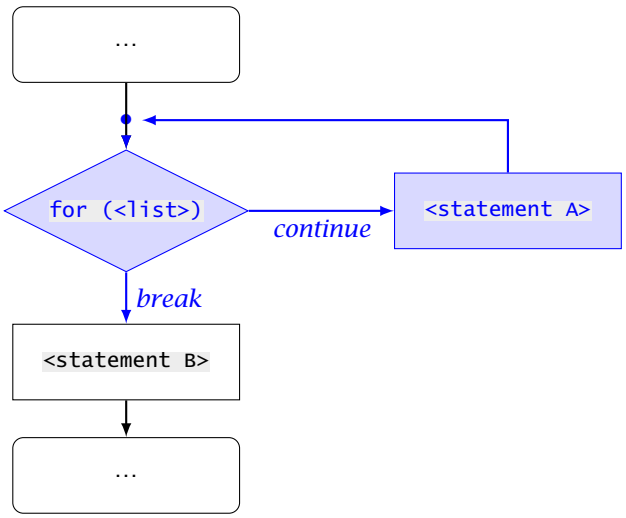
1.3.3 Iteration

We give the name *iteration* to the process of repetitive execution of a program statement (simple or compound)—e.g., *computed by iteration*. We use the same word, iteration, to name each one of these repetitions of the execution of a statement—e.g., the second iteration.

The section of computer code being executed multiple times, forms a loop (a closed path). Most loops contain a condition that determines when the flow of execution will exit the loop and continue at the next statement following the loop. In R three types of iteration loops are available: those using `for`, `while` and `repeat` constructs. They differ in how much flexibility they provide with respect to the values they iterate over, and how the condition that terminates the iteration is tested. When the same algorithm can be implemented with more than one of these constructs, using the least flexible of them usually results in the easiest to understand R scripts. In R, rather frequently, explicit loops as described in this section can be replaced advantageously by calls to the *apply* functions described in section 1.4 on page 26.

1.3.3.1 `for` loops

The most frequently used type of loop is a `for` loop. These loops work in R on lists or vectors of values to act upon. The implicit test for the end of the vector or list takes place at the top of the construct before the loop statement is evaluated. The flow chart has a *loop* as the execution can be directed to an earlier position in the sequence of statements, allowing the same code to be evaluated multiple times.




```
b <- 0
for (a in 1:5) b <- b + a
```

```
b
## [1] 15

b <- sum(1:5) # built-in function (faster)
b
## [1] 15
```

Here the statement `b <- b + a` is executed five times, with variable `a` sequentially taking each of the values in `1:5`. Instead of a simple statement used here, a compound statement could also have been used for the body of the `for` loop.

 It is important to note that a list or vector of length zero is a valid argument to `for()`, that triggers no error, but skips the statements in the loop body.

Some examples of use of `for` loops—and of how to avoid their use.

```
a <- c(1, 4, 3, 6, 8)
for(x in a) {print(x*2)} # print is needed!
## [1] 2
## [1] 8
## [1] 6
## [1] 12
## [1] 16
```

A call to `for` does not return a value. We need to assign values to an object so that they are not lost. If we print at each iteration the value of this object, we can follow how the stored value changes. Printing allows us to see, how the vector grows in length, unless we create a long-enough vector before the start of the loop.

```
b <- for(x in a) {x*2}
b
## NULL

b <- numeric()
for(i in seq(along.with = a)) {
  b[i] <- a[i]^2
  print(b)
}
## [1] 1
## [1]  1 16
## [1]  1 16  9
## [1]  1 16  9 36
## [1]  1 16  9 36 64

b
## [1]  1 16  9 36 64

# runs faster if we first allocate a long enough vector
b <- numeric(length(a))
for(i in seq(along.with = a)) {
  b[i] <- a[i]^2
  print(b)
}
```

```
## [1] 1 0 0 0 0
## [1]  1 16  0  0  0
## [1]  1 16  9  0  0
## [1]  1 16  9 36  0
## [1]  1 16  9 36 64

b
## [1]  1 16  9 36 64

# a vectorized expression is simplest and fastest
b <- a^2
b
## [1]  1 16  9 36 64
```

In the previous chunk we used `seq(along.with = a)` to build a new numeric vector with a sequence of the same length as vector `a`. Using this *idiom* is best as it ensures that even the case when `a` is an *empty* vector of length zero will be handled correctly, with `numeric(0)` assigned to `b`.

Look at the results from the above examples, and try to understand where the returned value comes from in each case. In the code chunk above, `print()` is used within the *loop* to make intermediate values visible. You can add additional `print()` statements to visualize other variables, such as `i`, or run parts of the code, such as `seq(along.with = a)`, by themselves.

In this case, the code examples trigger no errors or warnings, but the same approach can be used for debugging syntactically correct code that does not return the expected results.

In the examples above we show the use of `seq()` passing a vector as an argument to its parameter `along.with`. Run the examples below and explain why the two approaches are equivalent only when the length of `a` is one or more. Find the answer by assigning to `a`, vectors of different lengths, including zero (using `a <- numeric(0)`).

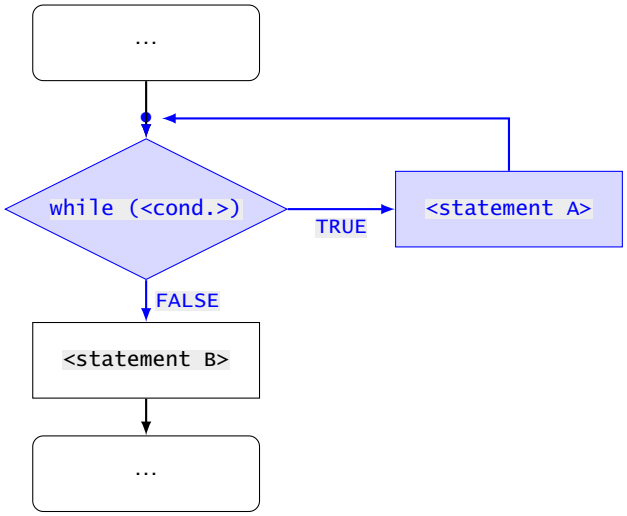
```
b <- numeric(length(a))
for(i in seq(along.with = a)) {
  b[i] <- a[i]^2
}
print(b)

c <- numeric(length(a))
for(i in 1:length(a)) {
  c[i] <- a[i]^2
}
print(c)
```

`for` loops as described above, in the absence of errors, have statically predictable behavior. The compound statement in the loop will be executed once for each member of the vector or list. Special cases may require the alteration of the

normal flow of execution in the loop. Two cases are easy to deal with, one is stopping iteration early, which we can do with `break()`, and another is jumping ahead to the start of the next iteration, which we can do with `next()`.

1.3.3.2 `while` loops

`while` loops are frequently useful, even if not as frequently used as `for` loops. Instead of a list or vector, they take a logical argument, which is usually an expression, but which can also be a variable.




```
a <- 2
while (a < 50) {
  print(a)
  a <- a^2
}
## [1] 2
## [1] 4
## [1] 16

print(a)
## [1] 256
```

 Make sure that you understand why the final value of `a` is larger than 50.

 The statements above can be simplified to:

```
a <- 2
print(a)
while (a < 50) {
  print(a <- a^2)
}
```

Explain why this works, and how it relates to the support in R of *chained* assignments to several variables within a single statement like the one below.

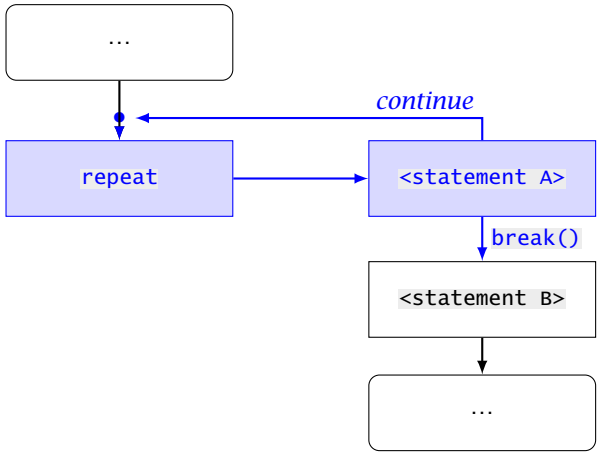
```
a <- b <- c <- 1:5
a
```

Explain why a second `print(a)` has been added before `while()`. Hint: experiment if necessary.

`while` loops as described above will terminate when the condition tested is `FALSE`. In those cases that require stopping iteration based on an additional test condition within the compound statement, we can call `break()` in the body of an `if` or `else` statement.

1.3.3.3 `repeat` loops

The `repeat` construct is less frequently used, but adds flexibility as termination will always depend on a call to `break()`, which can be located anywhere within the compound statement that forms the body of the loop. To achieve conditional end of iteration, function `break()` must be called, as otherwise, iteration in a `repeat` loop will not stop.




```
a <- 2
repeat{
  print(a)
  if (a > 50) break()
  a <- a^2
}
## [1] 2
## [1] 4
## [1] 16
## [1] 256
```

Please explain why the example above returns the values it does. Use the approach of adding `print()` statements, as described on page 20.

Although `repeat` loop constructs are easier to read if they have a single condition resulting in termination of iteration, it is allowed by the R language for the compound statement in the body of a loop to contain more than one call to `break()`, each within a different `if` or `else` statement.

Function `break()` can be also used within `for` and `while` loops. In such case, for clarity, the use of `break()` should be reserved for exceptional conditions. In a `for` loop, `next()` interrupts the current iteration and jumps to the start of the next one.

1.3.4 Explicit loops can be slow in R

If you have written programs in other languages, it will feel natural to you to use loops (`for`, `while`, `repeat`) for many of the things for which in R one would normally use vectorization. In R, using vectorization whenever possible keeps scripts shorter and easier to understand (at least for those with experience in R). More importantly, as R is an interpreted language, vectorized arithmetic tends to be much faster than the use of explicit iteration. In recent versions of R, byte-compilation is used by default and loops may be compiled on the fly, which relieves part of the burden of repeated interpretation. However, even byte-compiled loops are usually slower to execute than efficiently coded vectorized functions and operators.

Execution speed needs to be balanced against the effort invested in writing faster code. However, using vectorization and specific R functions requires little effort once we are familiar with them. The simplest way of measuring the execution time of an R expression is to use function `system.time()`. However, the returned time is in seconds and consequently the expression must take long enough to execute for the returned time to have useful resolution. See package 'microbenchmark' for tools for benchmarking code with better time resolution.

```
system.time({a <- numeric()
             for (i in 1:1000000) {
               a[i] <- i / 1000
               }
             })
##    user  system elapsed
##    0.25    0.01    0.27
```

Whenever working with large data sets, or many similar data sets, we will need to take performance into account. As vectorization usually also makes code simpler, it is good style to use vectorization whenever possible. For operations that are frequently used, R includes specific functions. It is thus important to consider not only vectorization of arithmetic but also check for the availability of performance-optimized functions for specific cases. The results from running the code examples in this box are not included, because they are the same for all chunks. Here we are interested in the execution time, and we leave this as an exercise.

```
a <- rnorm(10^7) # 10 000 0000 pseudo-random numbers
```

```r
# b <- numeric()
b <- numeric(length(a)-1) # pre-allocate memory
i <- 1
while (i < length(a)) {
  b[i] <- a[i+1] - a[i]
  print(b)
  i <- i + 1
}
b
```

```r
# b <- numeric()
b <- numeric(length(a)-1) # pre-allocate memory
for(i in seq(along.with = b)) {
  b[i] <- a[i+1] - a[i]
  print(b)
}
b
```

```r
# although in this case there were alternatives, there
# are other cases when we need to use indexes explicitly
b <- a[2:length(a)] - a[1:length(a)-1]
b
```

```r
# or even better
b <- diff(a)
b
```

Execution time can be obtained with `system.time()`. For a vector of ten million numbers, the `for` loop above takes 1.1 s and the equivalent `while` loop 2.0 s, the vectorized statement using indexing takes 0.2 s and function `diff()` takes 0.1 s. The `for` loop without pre-allocation of memory to `b` takes 3.6 s, and the equivalent while loop 4.7 s—i.e., the fastest execution time was more than 40 times faster than the slowest one. (Times for R 3.5.1 on my laptop under Windows 10 x64.)

1.3.5 Nesting of loops

All the execution-flow control statements seen above can be nested. We will show an example with two `for` loops. We first create a matrix of data to work with:

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

```
row.sum <- numeric()
for (i in 1:nrow(A)) {
  row.sum[i] <- 0
  for (j in 1:ncol(A))
    row.sum[i] <- row.sum[i] + A[i, j]
}
print(row.sum)
##  [1] 105 110 115 120 125 130 135 140 145 150
```

The code above is very general, it will work with any two-dimensional matrix with at least one column and one row. However, sometimes we need more specific calculations. `A[1, 2]` selects one cell in the matrix, the one on the first row of the second column. `A[1, ]` selects row one, and `A[ , 2]` selects column two. In the example above, the value of `i` changes for each iteration of the outer loop. The value of `j` changes for each iteration of the inner loop, and the inner loop is run in full for each iteration of the outer loop. The inner loop index `j` changes fastest.

 1) Modify the code in the example in the last chunk above so that it sums the values only in the first three columns of `A`, 2) modify the same example so that it sums the values only in the last three rows of `A`, 3) modify the code so that matrices with dimensions equal to zero (as reported by `ncol()` and `nrow()`).

Will the code you wrote continue working as expected if the number of rows in `A` changed? What if the number of columns in `A` changed, and the required results still needed to be calculated for relative positions? What would happen if `A` had fewer than three columns? Try to think first what to expect based on the code you wrote. Then create matrices of different sizes and test your code. After that, think how to improve the code, so that wrong results are not produced.

 If the total number of iterations is large and the code executed at each iteration runs fast, the overhead added by the loop code can make a big contribution to the total running time of a script. When dealing with nested loops, as the inner loop is executed most frequently, this is the best place to look for ways of reducing execution time. In this example, vectorization can be achieved easily for the inner loop, as R has a function `sum()` which returns the sum of a vector passed as its argument. Replacing the inner loop by an efficient function can be expected to improve performance significantly.

```
row.sum <- numeric(nrow(A)) # faster
for (i in 1:nrow(A)) {
  row.sum[i] <- sum(A[i, ])
}
print(row.sum)
##  [1] 105 110 115 120 125 130 135 140 145 150
```

`A[i, ]` selects row `i` and all columns. Reminder: in R the row index comes first.

Both explicit loops can be eliminated if we use an *apply* function, such as `apply()`, `lapply()` or `sapply()`, in place of the outer `for` loop. See section 1.4 below for details on the use of the different *apply* functions.

```
row.sum <- apply(A, MARGIN = 1, sum) # MARGIN=1 indicates rows
print(row.sum)
##  [1] 105 110 115 120 125 130 135 140 145 150
```

Calculating row sums is a frequent operation, so R has a built-in function for this. As earlier with `diff()`, it is always worthwhile to check if there is an existing R function, optimized for performance, capable of doing the computations we need. In this case, using `rowSums()` simplifies the nested loops into a single function call, both improving performance and readability.

```
rowSums(A)
##  [1] 105 110 115 120 125 130 135 140 145 150
```

1) How would you change this last example, so that only the last three columns are added up? (Think about use of subscripts to select a part of the matrix.) 2) To obtain column sums, one could modify the nested loops (think how), transpose the matrix and use `rowSums()` (think how), or look up if there is in R a function for this operation. A good place to start is with `help(rowSums)` as similar functions may share the same help page, or at least be listed in the "See also" section. Do try this, and explore other help pages in search for some function you may find useful in the analysis of your own data.

1.3.5.1 Clean-up

Sometimes we need to make sure that clean-up code is executed even if the execution of a script or function is aborted by the user or as a result of an error condition. A typical example is a script that temporarily sets a disk folder as the working directory or uses a file as temporary storage. Function `on.exit()` can be used to record that a user supplied expression needs to be executed when the current function, or a script, exits. Function `on.exit()` can also make code easier to read as it keeps creation and clean-up next to each other in the body of a function or in the listing of a script.

```
file.create("temp.file")
## [1] TRUE

on.exit(file.remove("temp.file"))
# code that makes use of the file goes here
```

1.4 *Apply* functions

Apply functions apply a function passed as an argument to parameter `FUN` or equivalent, to elements in a collection of R objects passed as an argument to parameter `X` or equivalent. Collections to which `FUN` is to be applied can be vectors, lists, data frames, matrices or arrays. As long as the operations to be applied are

independent—i.e., the results from one iteration are not used in another iteration— apply functions can replace `for`, `while` or `repeat` loops.

Conceptually, `for`, `while` and `repeat` loops are interpreted as controlling sequential evaluation of program statements. In contrast, R's *apply* functions are, conceptually, thought as evaluating a function in parallel for each of the different members of their input. So, while in loops the results of earlier iterations through a loop can be stored in variables and used in subsequent iterations, this is not possible in the case of *apply* functions.

The different *apply* functions in base R differ in the class of the values they accept for their `X` parameter, the class of the object they return and/or the class of the value returned by the applied function. `lapply()` and `sapply()` expect a `vector` or `list` as an argument passed through `X`. `lapply()` returns a `list` or an `array`; and `vapply()` always *simplifies* its returned value into a vector, while `sapply()` does the simplification according to the argument passed to its `simplify` parameter. All these *apply* functions can be used to apply an R function that returns a value of the same or a different class as its argument. In the case of `apply()` and `lapply()` not even the length of the values returned for each member of the collection passed as an argument, needs to be consistent. In summary, `apply()` is used to apply a function to the elements along a dimension of an object that has two or more *dimensions*, and `lapply()` and `sapply()` are used to apply a function to the members of a vector or list. `apply()` returns an array or a list or a vector depending on the size, and consistency in length and class among the values returned by the applied function.

1.4.1 Applying functions to vectors and lists

We first exemplify the use of `lapply()`, `sapply()` and `vapply()`. In the chunks below we apply a user-defined function to a vector.

A constraint is that the individual member objects in the list or vector passed as argument to the `x` parameter of *apply* functions will be always passed as a positional argument to the first formal parameter of the applied function, i.e., the function passed as argument to `FUN` must be compatible with this approach.

```
set.seed(123456) # so that a.vector does not change
a.vector <- runif(6) # A short vector as input to keep output short
str(a.vector)
##  num [1:6] 0.798 0.754 0.391 0.342 0.361 ...
```

```
my.fun <- function(x, k) {log(x) + k}
```

```
z <- lapply(X = a.vector, FUN = my.fun, k = 5)
str(z)
```

```
## List of 6
##  $ : num 4.77
##  $ : num 4.72
##  $ : num 4.06
##  $ : num 3.93
##  $ : num 3.98
##  $ : num 3.38
```

```
z <- sapply(X = a.vector, FUN = my.fun, k = 5)
str(z)
##  num [1:6] 4.77 4.72 4.06 3.93 3.98 ...
```

```
z <- sapply(X = a.vector, FUN = my.fun, k = 5, simplify = FALSE)
str(z)
## List of 6
##  $ : num 4.77
##  $ : num 4.72
##  $ : num 4.06
##  $ : num 3.93
##  $ : num 3.98
##  $ : num 3.38
```

We can see above that the computed results are the same in the three cases, but the class and structure of the objects returned differ.

Anonymous functions can be defined on the fly and passed to `FUN`, allowing us to re-write the examples above more concisely (only the second one shown).

```
z <- sapply(X = a.vector, FUN = function(x, k) {log(x) + k}, k = 5)
str(z)
##  num [1:6] 4.77 4.72 4.06 3.93 3.98 ...
```

Of course, as discussed in section 1.3.4 on page 23, when suitable vectorized functions are available, their use should be preferred. On the other hand, even if *apply* functions are usually not as fast as vectorized functions, they are faster than the equivalent `for()` loops.

```
z <- log(a.vector) + 5
str(z)
##  num [1:6] 4.77 4.72 4.06 3.93 3.98 ...
```

Function `vapply()` can be safer to use as the mode of returned values is enforced. Here is a possible way of obtaining means and variances across member vectors at each vector index position from a list of vectors. These could be called *parallel* means and variances. The argument passed to `FUN.VALUE` provides a template for the type of the return value and its organization into rows and columns. Notice that the rows in the output are now named according to the names in `FUN.VALUE`.

We first use `lapply()` to create the object `a.list` containing artificial data. One or more additional *named* arguments can be passed to the function to be applied.

```
set.seed(123456)
a.list <- lapply(rep(4, 5), rnorm, mean = 10, sd = 1)
str(a.list)
## List of 5
##  $ : num [1:4] 10.83 9.72 9.64 10.09
##  $ : num [1:4] 12.3 10.8 11.3 12.5
##  $ : num [1:4] 11.17 9.57 9 8.89
##  $ : num [1:4] 9.94 11.17 11.05 10.06
##  $ : num [1:4] 9.26 10.93 11.67 10.56
```

We define the function that we will apply, a function that returns a numeric vector of length 2.

```
mean_and_sd <- function(x, na.rm = FALSE) {
       c(mean(x, na.rm = na.rm),  sd(x, na.rm = na.rm))
    }
```

We next use `vapply()` to apply our function to each member vector of the list.

```
values <- vapply(X = a.list,
                 FUN = mean_and_sd,
                 FUN.VALUE = c(mean = 0, sd = 0),
                 na.rm = TRUE)
class(values)
## [1] "matrix" "array"

values
##            [,1]       [,2]     [,3]       [,4]      [,5]
## mean 10.0725427 11.7254442 9.657997 10.5573814 10.605846
## sd    0.5428149  0.7844356 1.050663  0.6460881  1.005676
```

As explained in section ?? on page ??, class `data.frame` is derived from class `list`. Apply function `mean_and_sd()` defined above to the data frame `cars` included as example data in R. The aim is to obtain the mean and standard deviation for each column.

1.4.2 Applying functions to matrices and arrays

In the next example we use `apply()` and `mean()` to compute the mean for each column of matrix `a.matrix`. In R the dimensions of a matrix, rows and columns, over which a function is applied are called *margins*. The argument passed to parameter `MARGIN` determines over which margin the function will be applied. If the function is applied to individual rows, we say that we operate on the first margin, and if the function is applied to individual columns, over the second margin. Arrays can have many dimensions, and consequently more margins. In the case of arrays with more than two dimensions, it is possible and useful to apply functions over multiple margins at once.

A constraint on the function to be applied is that the vector or "slice" will

always be passed as a positional argument to the first formal parameter of the applied function.

```
a.matrix <- matrix(runif(100), ncol = 10)
z <- apply(a.matrix, MARGIN = 1, FUN = mean)
str(z)
##  num [1:10] 0.247 0.404 0.537 0.5 0.504 ...
```

Modify the example above so that it computes row means instead of column means.

Look up the help pages for `apply()` and `mean()` and study them until you understand how additional arguments can be passed to the applied function. Can you guess why `apply()` was designed to have parameter names fully in uppercase, something very unusual for R code style?

If we apply a function that returns a value of the same length as its input, then the dimensions of the value returned by `apply()` are the same as those of its input. We use, in the next examples, a "no-op" function that returns its argument unchanged, so that input and output can be easily compared.

```
a.small.matrix <- matrix(rnorm(6, mean = 10, sd = 1), ncol = 2)
a.small.matrix <- round(a.small.matrix, digits = 1)
a.small.matrix
##      [,1] [,2]
## [1,] 11.3 10.4
## [2,] 10.6  8.6
## [3,]  8.2 11.0
```

```
no_op.fun <- function(x) {x}
```

```
z <- apply(X = a.small.matrix, MARGIN = 2, FUN = no_op.fun)
class(z)
## [1] "matrix" "array"

z
##      [,1] [,2]
## [1,] 11.3 10.4
## [2,] 10.6  8.6
## [3,]  8.2 11.0
```

In the chunk above, we passed `MARGIN = 2`, but if we pass `MARGIN = 1`, we get a return value that is transposed! To restore the original layout of the matrix we can transpose the result with function `t()`.

```
z <- apply(X = a.small.matrix, MARGIN = 1, FUN = no_op.fun)
z
```

```
##      [,1] [,2] [,3]
## [1,] 11.3 10.6  8.2
## [2,] 10.4  8.6 11.0

t(z)
##      [,1] [,2]
## [1,] 11.3 10.4
## [2,] 10.6  8.6
## [3,]  8.2 11.0
```

A more realistic example, but difficult to grasp without seeing the toy examples shown above, is when we apply a function that returns a value of a different length than its input, but longer than one. When we compute column summaries (`MARGIN = 2`), a matrix is returned, with each column containing the summaries for the corresponding column in the original matrix (`a.small.matrix`). In contrast, when we compute row summaries (`MARGIN = 1`), each column in the returned matrix contains the summaries for one row in the original array. What happens is that by using `apply()` the dimension of the original matrix or array over which we compute summaries "disappears." Consequently, given how matrices are stored in R, when columns collapse into a single value, the rows become columns. After this, the vectors returned by the applied function, are stored as rows.

```
mean_and_sd <- function(x, na.rm = FALSE) {
       c(mean(x, na.rm = na.rm),  sd(x, na.rm = na.rm))
    }
```

```
z <- apply(X = a.small.matrix, MARGIN = 2, FUN = mean_and_sd, na.rm = TRUE)
z
##           [,1]   [,2]
## [1,] 10.033333 10.000
## [2,]  1.625833  1.249
```

```
z <- apply(X = a.small.matrix, MARGIN = 1, FUN = mean_and_sd, na.rm = TRUE)
z
##            [,1]     [,2]     [,3]
## [1,] 10.8500000 9.600000 9.600000
## [2,]  0.6363961 1.414214 1.979899
```

In all examples above, we have used ordinary functions. Operators in R are functions with two formal parameters which can be called using infix notation in expressions—i.e., `a + b`. By back-quoting their names they can be called using the same syntax as for ordinary functions, and consequently also passed to the `FUN` parameter of apply functions. A toy example, equivalent to the vectorized operation `a.vector + 5` follows. We enclosed operator `+` in back ticks (`` ` ``) and pass by name a constant to its second formal parameter (`e2 = 5`).

```
set.seed(123456) # so that a.vector does not change
a.vector <- runif(10)
z <- sapply(X = a.vector, FUN = `+`, e2 = 5)
str(z)
##  num [1:10] 5.8 5.75 5.39 5.34 5.36 ...
```

Apply functions vs. loop constructs Apply functions cannot always replace explicit loops as they are less flexible. A simple example is the accumulation pattern, where we "walk" through a collection that stores a partial result between iterations. A similar case is a pattern where calculations are done over a "window" that moves at each iteration. The simplest and probably most frequent calculation of this kind is the calculation of differences between successive members. Other examples are moving window summaries such as a moving median (see page 23 for other alternatives to the use of explicit iteration loops).

1.5 Functions that replace loops

R provides several functions that can be used to avoid writing iterative loops in R. The most frequently used are taken for granted: `mean()`, `var()` (variance), `sd()` (standard deviation), `max()`, and `min()`. Replacing code implementing an iterative algorithm by a single function call simplifies the script's code and can make it easier to understand. These functions are written in C and compiled, so even when iterative algorithms are used, they are fast. A table with examples of additional functions available in base R that implement iterative algorithms is provided below. All these functions take a vector of arbitrary length as their first argument, except for `inverse.rle()`.

Function	Computation	Value, length
`sum()`	$\sum_{i=1}^{n} x_i$	numeric, 1
`prod()`	$\prod_{i=1}^{n} x_i$	numeric, 1
`cumsum()`	$\sum_{i=1}^{1} x_i, \cdots \sum_{i=1}^{j} x_i, \cdots \sum_{i=1}^{n} x_i$	numeric, $n_{\text{out}} = n_{\text{in}}$
`cumprod()`	$\prod_{i=1}^{1} x_i, \cdots \prod_{i=1}^{j} x_i, \cdots \prod_{i=1}^{n} x_i$	numeric, $n_{\text{out}} = n_{\text{in}}$
`cummax()`	cumulative maximum	numeric, $n_{\text{out}} = n_{\text{in}}$
`cummin()`	cumulative minimum	numeric, $n_{\text{out}} = n_{\text{in}}$
`runmed()`	running median	numeric, $n_{\text{out}} = n_{\text{in}}$
`diff()`	$x_2 - x_1, \cdots x_i - x_{i-1}, \cdots x_n - x_{n-1}$	numeric, $n_{\text{out}} = n_{\text{in}} - 1$
`diffinv()`	inverse of diff	numeric, $n_{\text{out}} = n_{\text{in}} + 1$
`factorial()`	$x!$	numeric, $n_{\text{out}} = n_{\text{in}}$
`rle()`	run-length encoding	$n_{\text{out}} < n_{\text{in}}$
`inverse.rle()`	run-length decoding	$n_{\text{out}} > n_{\text{in}}$

Build a `numeric` vector such as `x <- c(1, 9, 6, 4, 3)` and pass it as argument to the functions in the table above. Do the corresponding computations manually until you are sure to understand what each function calculates.

1.6 Object names and character strings

In all assignment examples before this section, we have used object names included as literal character strings in the code expressions. In other words, the names are "decided" as part of the code, rather than at run time. In scripts or packages, the object name to be assigned may need to be decided at run time and, consequently, be available only as a character string stored in a variable. In this case, function `assign()` must be used instead of the operators `<-` or `->`. The statements below demonstrate its use.

First using a `character` constant.

```
assign("a", 9.99)
a
## [1] 9.99
```

Next using a `character` value stored in a variable.

```
name.of.var <- "b"
assign(name.of.var, 9.99)
b
## [1] 9.99
```

The two toy examples above do not demonstrate why one may want to use `assign()`. Common situations where we may want to use character strings to store (future or existing) object names are 1) when we allow users to provide names for objects either interactively or as `character` data, 2) when in a loop we transverse a vector or list of object names, or 3) we construct at runtime object names from multiple character strings based on data or settings. A common case is when we import data from a text file and we want to name the object according to the name of the file on disk, or a character string read from the header at the top of the file.

Another case is when `character` values are the result of a computation.

```
for (i in 1:5) {
   assign(paste("zz_", i, sep = ""), i^2)
}
ls(pattern = "zz_*")
## [1] "zz_1" "zz_2" "zz_3" "zz_4" "zz_5"
```

The complementary operation of *assigning* a name to an object is to *get* an object when we have available its name as a character string. The corresponding function is `get()`.

```
get("a")
## [1] 9.99

get("b")
## [1] 9.99
```

If we have available a character vector containing object names and we want to create a list containing these objects we can use function `mget()`. In the example below we use function `ls()` to obtain a character vector of object names matching a specific pattern and then collect all these objects into a list.

```
obj_names <- ls(pattern = "zz_*")
obj_lst <- mget(obj_names)
str(obj_lst)
## List of 5
##  $ zz_1: num 1
##  $ zz_2: num 4
##  $ zz_3: num 9
##  $ zz_4: num 16
##  $ zz_5: num 25
```

Think of possible uses of functions `assign()`, `get()` and `mget()` in scripts you use or could use to analyze your own data (or from other sources). Write a script to implement this, and iteratively test and revise this script until the result produced by the script matches your expectations.

1.7 The multiple faces of loops

To close this chapter, I will mention some advanced aspects of the R language that are useful when writing complex scrips—if you are going through the book sequentially, you will want to return to this section after reading chapters ?? and ??. In the same way as we can assign names to `numeric`, `character` and other types of objects, we can assign names to functions and expressions. We can also create lists of functions and/or expressions. The R language has a very consistent grammar, with all lists and vectors behaving in the same way. The implication of this is that we can assign different functions or expressions to a given name, and consequently it is possible to write loops over lists of functions or expressions.

In this first example we use a *character vector of function names*, and use function `do.call()` as it accepts either character strings or function names as its first argument. We obtain a numeric vector with named members with names matching the function names.

```
x <- rnorm(10)
results <- numeric()
fun.names <- c("mean", "max", "min")
for (f.name in fun.names) {
   results[[f.name]] <- do.call(f.name, list(x))
   }
results
##       mean        max        min
##  0.5453427  2.5026454 -1.1139499
```

When traversing a *list of functions* in a loop, we face the problem that we cannot access the original names of the functions as what is stored in the list are the definitions of the functions. In this case, we can hold the function definitions in the loop variable (`f` in the chunk below) and call the functions by use of the function call notation (`f()`). We obtain a numeric vector with anonymous members.

```
results <- numeric()
funs <- list(mean, max, min)
for (f in funs) {
   results <- c(results, f(x))
   }
results
## [1]  0.5453427  2.5026454 -1.1139499
```

We can use a named list of functions to gain full control of the naming of the results. We obtain a numeric vector with named members with names matching the names given to the list members.

```
results <- numeric()
funs <- list(average = mean, maximum = max, minimum = min)
for (f in names(funs)) {
   results[[f]] <- funs[[f]](x)
   }
results
##    average    maximum    minimum
##  0.5453427  2.5026454 -1.1139499
```

Next is an example using model formulas. We use a loop to fit three models, obtaining a list of fitted models. We cannot pass to `anova()` this list of fitted models, as it expects each fitted model as a separate nameless argument to its `...` parameter. We can get around this problem using function `do.call()` to call `anova()`. Function `do.call()` passes the members of the list passed as its second argument as individual arguments to the function being called, using their names if present. `anova()` expects nameless arguments so we need to remove the names present in `results`.

```
my.data <- data.frame(x = 1:10, y = 1:10 + rnorm(10, 1, 0.1))
results <- list()
models <- list(linear = y ~ x, linear.orig = y ~ x - 1, quadratic = y ~ x + I(x^2))
for (m in names(models)) {
   results[[m]] <- lm(models[[m]], data = my.data)
   }
str(results, max.level = 1)
## List of 3
##  $ linear     :List of 12
##   ..- attr(*, "class")= chr "lm"
##  $ linear.orig:List of 12
##   ..- attr(*, "class")= chr "lm"
##  $ quadratic  :List of 12
##   ..- attr(*, "class")= chr "lm"

do.call(anova, unname(results))
## Analysis of Variance Table
##
## Model 1: y ~ x
## Model 2: y ~ x - 1
## Model 3: y ~ x + I(x^2)
##   Res.Df     RSS Df Sum of Sq      F    Pr(>F)
## 1      8 0.05525
## 2      9 2.31266 -1   -2.2574 306.19 4.901e-07 ***
## 3      7 0.05161  2    2.2611 153.34 1.660e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If we had no further use for `results` we could simply build a list with nameless members by using positional indexing.

```
results <- list()
models <- list(y ~ x, y ~ x - 1, y ~ x + I(x^2))
for (i in seq(along.with = models)) {
   results[[i]] <- lm(models[[i]], data = my.data)
   }
str(results, max.level = 1)
## List of 3
##  $ :List of 12
##   ..- attr(*, "class")= chr "lm"
##  $ :List of 12
##   ..- attr(*, "class")= chr "lm"
##  $ :List of 12
##   ..- attr(*, "class")= chr "lm"

do.call(anova, results)
## Analysis of Variance Table
##
## Model 1: y ~ x
## Model 2: y ~ x - 1
## Model 3: y ~ x + I(x^2)
##   Res.Df     RSS Df Sum of Sq      F    Pr(>F)
## 1      8 0.05525
## 2      9 2.31266 -1   -2.2574 306.19 4.901e-07 ***
## 3      7 0.05161  2    2.2611 153.34 1.660e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

1.8 Data pipes in base R

The 'tidyverse' has popularized the use of data pipes in R. Recently base R has gained a pipe operator, `|>`, and in combination with well known base R functions it provides a concise notation for data selection and transformations. In general whenever we use temporary variables to store values that are used only once, we can chain the statements making the saving into a temporary variable implicit instead of explicit. In many cases, nested function calls can also be transformed into easier to read *pipes*.

Addition of computed variables to a data frame using `within()` and selecting rows with `subset()` are combined in our first 'toy' example. We use the `_` placeholder to indicate the value returned by the preceding function in the pipe.

```
data.frame(x = 1:10, y = rnorm(10)) |>
  within(data = _,
         {
           x4 <- x^4
           is.large <- x^4 > 1000
         }) |>
  subset(x = _, is.large)
##     x           y is.large    x4
```

```
## 6   6 -0.85586700     TRUE  1296
## 7   7  0.06955833     TRUE  2401
## 8   8 -1.04619827     TRUE  4096
## 9   9 -2.74886838     TRUE  6561
## 10 10 -1.12985961     TRUE 10000
```

To summarize variables we can use `aggregate()`.

```
data.frame(group = factor(rep(c("T1", "T2", "Ctl"), each = 4)),
           y = rnorm(12)) |>
  subset(x = _, group %in% c("T1", "T2")) |>
  aggregate(data = _, y ~ group, mean)
##   group          y
## 1    T1 -0.5017887
## 2    T2  0.7069193
```

Although the extraction operators are not accepted on the rhs of a pipe, function `getElement()` can be used to extract a member by name, in this case a column.

```
data.frame(group = factor(rep(c("T1", "T2", "Ctl"), each = 4)),
           y = rnorm(12)) |>
  subset(x = _, group %in% c("T1", "T2")) |>
  aggregate(data = _, y ~ group, mean) |>
  getElement("y")
## [1] -0.5608237 -0.3212313
```

1.9 Further reading

For further readings on the aspects of R discussed in the current chapter, I suggest the books *The Art of R Programming: A Tour of Statistical Software Design* (Matloff) and *Advanced R* (Wickham).

Bibliography

Hughes, T. P. (2004). *American Genesis*. The University of Chicago Press. 530 pp. ISBN: 0226359271 (cit. on p. 6).

Knuth, D. E. (1984). "Literate programming". In: *The Computer Journal* 27.2, pp. 97–111 (cit. on p. 6).

Lamport, L. (1994). *LaTeX: a document preparation system*. English. 2nd ed. Reading: Addison-Wesley, p. 272. ISBN: 0-201-52983-1 (cit. on p. 6).

Lemon, J. (2020). *Kickstarting R*. URL: `https://cran.r-project.org/doc/contrib/Lemon-kickstart/kr_intro.html` (visited on 02/07/2020).

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843 (cit. on p. 37).

Wickham, H. (2019). *Advanced R*. 2nd ed. Taylor & Francis Inc. 588 pp. ISBN: 0815384572 (cit. on p. 37).

Xie, Y. (2013). *Dynamic Documents with R and knitr*. The R Series. Chapman and Hall/CRC, p. 216. ISBN: 1482203537 (cit. on p. 6).

— (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC. ISBN: 9781138700109 (cit. on p. 6).

Xie, Y., J. J. Allaire, and G. Grolemund (2018). *R Markdown*. Chapman and Hall/CRC. 304 pp. ISBN: 1138359335 (cit. on p. 6).

General index

Index of R names by category

Alphabetic index of R names