

A Tutorial to R

Created for the “Modelling Course in Population and Evolutionary Biology”
at the ETH Zürich

<http://www.tb.ethz.ch/education/learningmaterials/modelingcourse.html>

by Viktor Müller

Course director: Sebastian Bonhoeffer
Theoretical Biology
Institute of Integrative Biology
ETH Zürich

Version date: June 3, 2023. This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>.



Contents

1	Introduction	3
2	A sample session	3
2.1	Creating and manipulating data	4
2.1.1	Getting started (with vectors)	4
2.1.2	Matrices	12
2.1.3	Lists and data frames	17
2.2	Random numbers	23
2.3	Control structures and logical operators	25
2.4	Avoiding loops	27
2.5	Writing your own functions	28
2.6	Getting help	30
3	Recommended reading	31

1 Introduction

We had a number of reasons to choose **R** as the programming language of our modelling course. First, **R** is “user-friendly”: it has a large number of useful built-in functions (which you therefore do not need to code yourself when you need them), and it is willing to handle many details automatically and flexibly^a. This makes it a good choice for a “first language”, but even experienced programmers might find it a useful tool to write code quickly and conveniently^b. Second, **R** is available free of charge for all platforms (Windows, UNIX/Linux, Mac), and a large community of developers ensures that it will remain to be updated and supported in the future: therefore, you will be able to use it wherever you go. Finally, **R** has particular strengths in two fields that are immensely useful to all scientists (not just theoreticians). It includes powerful statistics to analyze your data (in fact, this is what it is best known and most used for), and has very efficient graphics facilities to produce publication-quality figures from your results. In the course, we will focus on programming (implementing mathematical and simulation models), but the skills you are going to acquire will also help you automate your data analyses and generate high-quality figures easily in the future.

This document is intended as a concise tutorial to get you started with **R**. Below we will concentrate on the basic concepts and features that you will need to use during the course. If you are interested in further details (in particular, statistics, which we will hardly discuss here; or, e.g., in the history of the **R** language), please consult the sources listed at the end of this document. The easiest way to learn programming is by example: therefore, we will introduce the features of **R** by walking you through a sample session. If you are new to **R** we suggest that you use this Tutorial as a reference, coming back to it repeatedly during the course.

You can download the latest version of **R** from the **R** Project webpage (<http://www.r-project.org/>). Follow the installation instructions in the appropriate FAQ (Windows, Mac, UNIX/Linux). When you have installed **R**, we suggest downloading and installing **RStudio** (<http://rstudio.org>), as well, which provides a very convenient way to work with **R** under all platforms, and is also freely available.

2 A sample session

Start **RStudio** or an **R** GUI session, and load `sample.r` (e.g., by double-clicking on the file), which you can download from the course homepage. To execute code, run the selected lines by pressing **CTRL+ENTER** (Win/Linux) or **CMD+ENTER** (Mac), or by clicking on the appropriate icon in top left panel of **RStudio**. Alternatively, you can also type the commands at the Console (bottom left panel in **RStudio**) and hit **<ENTER>**. Below, lines of code will be typeset in **fixed width font** to distinguish them from the explanatory text.

^aE.g. the size and even the type of variables is handled flexibly, as long as they can be interpreted in a meaningful way. **R** is generally trying hard to figure out what you might have wanted it to do.

^bOf course, all these convenience features come at a cost of computation speed: **R** is much slower than **C** or other “basic” programming languages.

2.1 Creating and manipulating data

2.1.1 Getting started (with vectors)

Type: `a <- 5 + 3`

This is an *assignment*: using the assignment operator “<-”, you have just assigned the evaluated result of the right hand side (i.e. 8) to a variable called `a`, which was immediately created in the process. If you type the name of the variable (type `a` and hit <ENTER>), its contents will be shown in the Console^c:

```
> a
[1] 8
```

The number in angular brackets (`[1]`) before the value of your variable is just an index, not a part of your data. We will come back to this later.

If you wish to see the evaluated result of your assignment immediately, put the assignment in brackets, and the result is displayed in the Console:

```
> (a <- 5 + 3) # output on screen
[1] 8
```

Otherwise, `a` has been defined just as before. Notice the use of the hashmark (`#`): everything from the hashmark to the end of the line is ignored by `R`, which allows you to comment your code, as you will often observe in `sample.r`.

However, in `R`, there is no such thing as a lonely piece of data. The simplest data structure is a vector, which can hold a series of data of the same type. The variable you have just created is thus a vector of length 1: it happens to hold a single value initially, but it can be extended easily:

```
> a[2] <- 2.65
> a
[1] 8.00 2.65
```

The number in angular brackets in the assignment indicates the position of the element that we are referring to. By assigning a value to the second element of the vector, we have extended it automatically to length 2. Furthermore, `R` has silently changed the display mode to show all values to two decimal places.

^cIn `RStudio`, the new variable and its value are also displayed in the **Environment** tab of the top right panel immediately as the variable has been created. This tab contains a scrollable list of all data and functions currently defined.

Note that R is case sensitive, e.g., `a` differs from `A`, and the latter has not been defined in this session yet:

```
> A
Error: object 'A' not found
```

Now let us generate some regular sequences, which are often useful for programming. If the sequence you wish to create has a step of 1, you can use the colon operator for both increasing and decreasing sequences:

```
> a <- 8 # set a back to a single value
> (b <- a:33)
[1] 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
[21] 28 29 30 31 32 33
> 5:-4
[1] 5 4 3 2 1 0 -1 -2 -3 -4
```

In the first example, notice that the line has been broken due to its length. The numbers in square brackets at the beginning of each line of a printed vector indicate the index of the first element in that row. Starting at 8, 28 is the 21st element of the vector. These numbers are there just to help readability: they are not part of the vector. In fact, the break point of the line may be at a different position in your window of R or RStudio, if the width of the Console window and/or font size is set differently from those used for this demo. Try resizing the width of the window, and call `b` again.

If you need a regular sequence with a step other than 1, you can use the function `seq()`. E.g.:

```
> (b <- seq(3,9,2))
[1] 3 5 7 9
```

Here the *function* `seq()` was called with 3 arguments (the numbers included in the brackets): the first number defined the start of the sequence, the second its end, while the third the step. Try `?seq`, which opens the help page that describes the function (in RStudio this will appear under the **Help** tab of the bottom right panel). We will later return to how functions can be called or defined.

Now try this:

```
> (x <- c(a,b*2))
[1] 8 6 10 14 18
```

The `c()` function joins together (concatenates) its arguments into a single vector. Notice that the second argument was evaluated before being joined to the first argument, and notice also

that `b*2` resulted in a vector which includes all elements of `b` multiplied by two. This is an important rule: arithmetic operations on a vector (and also on more complex data structures) are carried out element by element.

Due to the flexibility of `R`, very few names are “forbidden”, e.g. you can use `c <- c(a,b*2)`, in which case you will have a vector called `c` and will still be able to use the function of the same name (invoking it with arguments in brackets, `c(arg1,arg2,...)`, will distinguish it from the vector). This flexibility of `R` is often useful; however, it is best to avoid such re-use of built-in names, as this can, in some rare cases, lead to errors and strange behaviour in your program. Let us check what objects we have defined so far with the command `ls()`^d.

```
> ls()
[1] "a" "b" "c" "x"
```

And now remove the variable `c` with the command `rm(c)`. Repeat the `ls()` command to check that `c` has indeed been removed.

For a further example of element-by-element operations, type `1:5+1`. Notice that the sequence from 1 to 5 was created first, and then 1 was added to each of the five elements. If you need a sequence from 1 to `a+1`, then put the latter into brackets: `1:(a+1)`. If you are unsure about the order (“precedence”) of operations, it is always safest to group your expressions with brackets.

Element-by-element operations also work when two (or more) objects that each contain several elements are involved, e.g. check out `1:3 + 1:3`. What happens, however, when the objects have different length? E.g.:

```
> 1:3 + 1:6
[1] 2 4 6 5 7 9
```

In such cases, the object(s) with fewer elements will be “re-used” until they match the size of the largest object. E.g., in this case, the shorter sequence `1:3` was repeated twice, and the operation carried out was the following:

	1	2	3	1	2	3
+	1	2	3	4	5	6
=	2	4	6	5	7	9

This is called the “recycling rule”, which is applied very widely in `R`.

Elements of a vector can be selected by their position in the vector, which can be provided by a vector of the position indices in square brackets. For example, try:

^dIn `RStudio` you can also check the list of defined objects in the **Environment** tab.

```
> (d <- 1:3 + 1:6)
[1] 2 4 6 5 7 9
> d[1];d[2:4];d[c(1,3,4,1)]
[1] 2
[1] 4 6 5
[1] 2 6 5 2
```

Importantly, unlike in some other programming languages, indexing starts at 1 (not zero). Notice that the same index can occur more than once in the index vector, which will include the same element repeatedly. Notice also that several commands can be placed in the same line if they are separated by semi-colons.

Negative indices will remove the indexed elements, and return the rest of the vector, e.g.:

```
> d[-2];d[c(-1,-3)]
[1] 2 6 5 7 9
[1] 4 5 7 9
```

The elements of a vector can also have names, and these can then also be accessed by their names, as in:

```
> e = c(e1=5,e2=11,last=54)
> e["e1"]
e1
5
> e[c("last","e1")]
last  e1
54    5
```

Names of the elements are retained in the selection^e. Number indexing is always possible whether or not the elements have names, e.g. try `e[3:2]`.

Finally, in a very useful feature of R, logical vectors can be used to select elements of an object that fulfil a given criterion. Logical values can be `TRUE` or `FALSE`^f. You typically generate logical vectors by a logical test^g, e.g.:

```
> d>5
[1] FALSE FALSE  TRUE FALSE  TRUE  TRUE
```

^eNames can be removed with, e.g., `as.numeric(e)` or `names(e) <- NULL`.

^fThere are also pre-defined variables `T` and `F`, which initially hold these values. You can therefore use just the shorter `T/F` to refer to `TRUE/FALSE`; however, `T` and `F` are not protected, you can overwrite them, and therefore it is always safer to use the full forms (which are protected).

^gThe relational operators are `<`, `<=`, `>`, `>=`; `==` tests for exact equality and `!=` for inequality.

The test was carried out element by element on the `d` vector, and the result is a logical vector of the same length, with `TRUE` values where the elements in `d` fulfilled the test criterion, and `FALSE` values where they did not. The resulting logical vector can be used to select just those elements of `d` that fulfil the criterion:

```
> d[d>5]
[1] 6 7 9
```

The positions (indices) of the `TRUE` values in a logical vector can be found with the `which()` function, e.g. the indices of elements greater than 5 in `d` are obtained as:

```
> which(d>5)
[1] 3 5 6
```

Thus, selections of type `d[d>5]` and `d[which(d>5)]` tend to give the same result^h. You can also use a test on one vector to select from another vectorⁱ:

```
> (le <- letters[1:6]) # letters is a built-in vector of small letters
[1] "a" "b" "c" "d" "e" "f"
> le[d>5]
[1] "c" "e" "f"
```

Finally, you can also store the results of a logical test and then use the created logical vector to make your selection:

```
> testresult <- d>5
> le[testresult]
[1] "c" "e" "f"
```

Now let us wipe our slate clean by removing all objects defined so far, using the command `rm(list=ls())`^j. Previously, you have seen that variables can be created “on the fly” by assigning a value to a previously undefined object^k, and vectors could be extended simply by assigning a value to an element beyond the last index of the vector. However, even `R` has its limits of flexibility, and the following command will give you an error message:

^hThere is a difference when the result of the test contains `NA` values, which are dropped by `which()`.

ⁱIn typical use, you have a logical vector of the same length as the vector you are selecting from. If the lengths can differ in your program, make sure you have tested and understood how this is handled in `R`. In general: when you are unsure about how such a nontrivial situation is handled, it is always recommended to check out a simple example. Otherwise, you can easily create hidden “bugs” that do not crash the program, but result in unintended behaviour: this kind of bug is the most difficult to find.

^jIn `RStudio` you can do the same by clicking on the `Clear` button of the `Environment` tab.

^kThis is a difference from low-level programming languages, which typically require the declaration of a variable before its first use.


```
> b[5] <- 10
Error in b[5] <- 10 : object 'b' not found
```

Thus, a particular position of a vector can be assigned a value only if that vector already exists. Once a vector has been created, extension works even in the following way:

```
> b <- 1
> b[5] <- 10
> b
[1] 1 NA NA NA 10
```

The initial assignment defined the value of the first element of the vector. After this, we were able to assign a value to the fifth element, which has automatically extended the vector to length 5. However, the values in between (i.e., `b[2:4]`) have not been defined, and these elements have automatically assumed the special value “NA”, which indicates undefined values.

The function `is.na()` lets you check which values of the vector are undefined:

```
> is.na(b)
[1] FALSE TRUE TRUE TRUE FALSE
```

R is very efficient at working with missing/undefined values, which is very useful when handling “real-life” data. Element-by-element operations return NA in the same positions where the original object had them:

```
> b*2
[1] 2 NA NA NA 20
> b>3
[1] FALSE NA NA NA TRUE
```

Furthermore, most built-in functions have intelligent ways to handle NAs, as can be seen in the following examples:

```
> mean(b)
[1] NA
> sum(b)
[1] NA
> mean(b, na.rm=TRUE)
[1] 5.5
> sum(b, na.rm=TRUE)
[1] 11
```

The mean and sum of a set of numbers are clearly undefined, if any of the values in the set is undefined. However, the `na.rm` argument of these functions allows the calculation of the mean or sum of just those values that are defined in the vector. Both answers have their use depending on the situation.

In some cases, it is also useful to create a vector without assigning any values to it (e.g. to prepare it to receive assigned values at indexed positions). Here we create `b` (erasing and overwriting its previous definition) as an “empty” vector of type numeric:

```
> b <- numeric()
> b
numeric(0)
```

The notation `numeric(0)` denotes a vector of length zero, with no elements in it, but now prepared to receive value assignments to any position, e.g.:

```
> b[3] <- 5.4
> b
[1] NA NA 5.4
```

By now, you have encountered all three basic types of data: numeric, character and logical. Numeric vectors¹ (and other data objects) hold numbers, character vectors hold character strings (not just single characters), and logical vectors hold the two kinds of logical values. Vectors of length n can be created with the commands `numeric(n)`, `character(n)` and `logical(n)`. The initial value of all elements will be zero, the empty string "", and `FALSE` for the three data types, respectively:

```
> numeric(5)
[1] 0 0 0 0 0
> character(4)
[1] "" "" "" ""
> logical(6)
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

The `c()` function can also be used to create character vectors; the character strings must be quoted:

```
> c.v <- c("apple", "banana", "supernova")
> c.v
[1] "apple"      "banana"     "supernova"
```

¹Numeric data come in several flavours: integer, double, complex. However, these are typically handled automatically by R and you will not need to worry about them in this course.

Conversion between the different types of data is typically simple if there is a meaningful way to make the conversion. For example, a character vector can easily accept a number assigned to one or more of its elements:

```
> b <- "Garfield"
> b[2:3] <- c(4,7.2)
> b
[1] "Garfield" "4"          "7.2"
```

The numbers 4 and 7.2 were added to the vector as character strings, and indeed, an attempted arithmetic operation aborts with an error message:

```
> b[3]/2
Error in b[3]/2 : non-numeric argument to binary operator
```

However, there is a whole class of functions that can convert the type of a vector. For example, `as.numeric()` can convert its arguments into numeric type:

```
> as.numeric(b)/2
[1] NA 2.0 3.6
Warning message:
NAs introduced by coercion
```

The character strings "4" and "7.2" could be meaningfully converted (back) to number, and then division by 2 was carried out. However, the string "Garfield" could not be converted, and was therefore replaced with an NA, which yielded NA also when divided by 2. Note that while the expression `as.numeric(b)` was evaluated as a numeric vector, the original vector `b` was itself not converted in the process, and is still of type character. To convert a vector (rather than to create a converted copy), you can change its mode:

```
> mode(b) <- "numeric"
Warning message:
In eval(expr, envir, enclos) : NAs introduced by coercion
> b
[1] NA 4.0 7.2
```

A warning message was issued (this is useful because such a problem may indeed indicate an error in the program), but the conversion was carried out as above. Not all conversions are possible (or meaningful), but two further types are sometimes useful:

```
> as.logical(c(1,5.64,0,567,0,-1))
[1] TRUE TRUE FALSE TRUE FALSE TRUE
> as.numeric(c(TRUE,FALSE,FALSE,TRUE))
[1] 1 0 0 1
```

Notice that all numeric values except zero are converted into `TRUE` and zero is converted into `FALSE`. In the reverse direction, `TRUE` is converted into 1, and `FALSE` is converted into zero. Some functions can make this conversion automatically, e.g., `sum()` can count the number of `TRUE` elements in a logical vector directly, without conversion:

```
> (a <- cos(1:5*10))
[1] -0.8390715  0.4080821  0.1542514 -0.6669381  0.9649660
> sum(a>0)
[1] 3
```

2.1.2 Matrices

Matrices are two-dimensional tables of data of a single type (can be numeric, character or logical, just like vectors). The `matrix()` command can create a matrix of a given size from a data vector:

```
> (m <- matrix(1:9,nrow=3,ncol=3))
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Notice that the values are filled into the matrix by column. It is also possible to fill a matrix by rows:

```
> (m <- matrix(1:9,nrow=3,ncol=3,byrow=TRUE))
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

It is possible to create a matrix with all values undefined, or with all elements filled with the same single value (this latter option takes advantage of the recycling rule):

```

> matrix(nrow=2,ncol=3)
      [,1] [,2] [,3]
[1,]    NA    NA    NA
[2,]    NA    NA    NA
> matrix(0,nrow=2,ncol=3)
      [,1] [,2] [,3]
[1,]     0     0     0
[2,]     0     0     0

```

The indexing of a matrix can be easily remembered from the way it is printed (look at the row and column indices in the row and column headers). `m[i,]` selects the i^{th} row, `m[,j]` selects the j^{th} column, and `m[i,j]` selects the j^{th} element of the i^{th} row. Multiple rows and columns (and any combination of these) can also be selected by providing multi-element vectors as i and j . Some examples:

```

> m[2,2]
[1] 5
> m[2,]
[1] 4 5 6
> m[,1]
[1] 1 4 7
> m[1:2,2]
[1] 2 5
> m[c(3,1),c(1:2)]
      [,1] [,2]
[1,]     7     8
[2,]     1     2

```

In addition, the elements of a matrix can also be selected by vector indexing:

```

> m[4]
[1] 2
> m[c(1,5,6)]
[1] 1 5 8

```

In this case, the positions are interpreted in column-wise order (Figure 1).

The same convention is used when a matrix is converted to vector:

```

> as.vector(m)
[1] 1 4 7 2 5 8 3 6 9

```

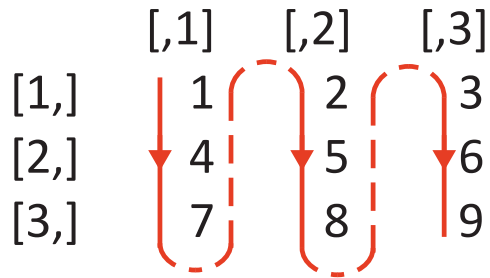


Figure 1: When a matrix is indexed with a single index (vector indexing), the positions are interpreted in column-wise order.

If you define a matrix with the default column-wise order, then vector indexing returns the original vector that was used to fill the matrix:

```
> (m <- matrix(1:9,nrow=3,ncol=3))
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[4]
[1] 4
> as.vector(m)
[1] 1 2 3 4 5 6 7 8 9
```

Sometimes it is useful to switch the column and row indices of a matrix, i.e. to obtain the transpose of a matrix. This can be done with the `t()` function:

```
> t(m)
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

The resulting matrix will be such as if you had created a matrix with the same data as in the original matrix, but with the alternative order of filling (row-wise instead of column-wise, or vice versa).

There are two simple functions that allow you to find out the row and columns indices of each element of a matrix:

```
> col(m)
```

```

      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
> row(m)
      [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3

```

While this may look trivial, it can be useful in many cases, e.g. if you wish to find out without calculations the row and column indices of an element selected by vector indexing:

```

> col(m)[4];row(m)[4]
[1] 2
[1] 1

```

The basic (“scalar”) arithmetic operations work on matrices element by element, just as we have seen for vectors:

```

> (n <- m/10)
      [,1] [,2] [,3]
[1,]  0.1  0.4  0.7
[2,]  0.2  0.5  0.8
[3,]  0.3  0.6  0.9

```

Two matrices with the same dimensions^m can also be used in arithmetic expressions for element-by-element operations:

```

> m*n
      [,1] [,2] [,3]
[1,]  0.1  1.6  4.9
[2,]  0.4  2.5  6.4
[3,]  0.9  3.6  8.1

```

Matrix multiplication can be performed with the special operator `%*%`:

```

> m %*% n
      [,1] [,2] [,3]
[1,]  3.0  6.6 10.2
[2,]  3.6  8.1 12.6
[3,]  4.2  9.6 15.0

```

^mA vector and a matrix can also be used together, but two matrices of different dimensions can not.

Two examples on how the individual elements of the product have been calculated:

```
> sum(m[1,]*n[,1])
[1] 3
> sum(m[2,]*n[,1])
[1] 3.6
```

The rows and columns of a matrix can be named, and then these names can also be used (also in combination with number indices) to select elements or blocks of a matrix:

```
> (m <- matrix(1:9,nrow=3,ncol=3,dimnames=list(letters[1:3],letters[4:6])))
  d e f
a 1 4 7
b 2 5 8
c 3 6 9
> m["a","e"]
[1] 4
> m["a",2:3]
e f
4 7
```

Names can be removed (and also added or changed) with the `dimnames()` function:

```
> dimnames(m) <- NULL
> m
      [,1] [,2] [,3]
[1,]     1     4     7
[2,]     2     5     8
[3,]     3     6     9
```

NULL is a special object, which is typically used to indicate the lack of an attribute.

Finally, elements of a matrix can also be selected by an index matrix, constructed such that each row of the index matrix contains the row and column indices of one element of the indexed matrix. That is, an index matrix of size $n \times 2$ will select n elements of the indexed matrix:

```
> (ind <- matrix(c(2,3,1,2,3,3),nrow=3,byrow=TRUE))
      [,1] [,2]
[1,]     2     3
[2,]     1     2
[3,]     3     3
> m[ind] # each row of the matrix pulls out an element
[1] 8 4 9
```


Finally, functions `rbind()` and `cbind` allow you to join matrices of appropriate dimensions (equal number of columns or rows, respectively) by rows or columns:

```
> rbind(m,n)
      [,1] [,2] [,3]
[1,]  1.0  4.0  7.0
[2,]  2.0  5.0  8.0
[3,]  3.0  6.0  9.0
[4,]  0.1  0.4  0.7
[5,]  0.2  0.5  0.8
[6,]  0.3  0.6  0.9
> cbind(m,n)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]     1     4     7 0.1  0.4  0.7
[2,]     2     5     8 0.2  0.5  0.8
[3,]     3     6     9 0.3  0.6  0.9
```

In addition to two-dimensional matrices, you can also create data tables of higher dimensions with the `array()` function:

```
> array(1:24,dim=c(3,4,2))
, , 1
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8    11
[3,]     3     6     9    12

, , 2
      [,1] [,2] [,3] [,4]
[1,]    13    16    19    22
[2,]    14    17    20    23
[3,]    15    18    21    24
```

High-dimensional arrays are printed in “slices” of two-dimensional matrices. The order of filling and indexing works such that the *f*irst index moves *f*astest.

2.1.3 Lists and data frames

While vectors and matrices/arrays can hold only data of a single type, lists and data frames can combine data of different types. The list is the most flexible data type: its components can be vectors and arrays holding any type of data, and even lists; and the sizes of the components can differ. The components of a list are typically named. An example:

```

> course <- list(name="Modelling",level=c("masters","PhD"),
  date=list(from="3 June",to="14 June"),maxnumber=20)
> course
$name
[1] "Modelling"

$level
[1] "masters" "PhD"

$date
$date$from
[1] "3 June"

$date$to
[1] "14 June"

$maxnumber
[1] 25

```

This list gathers several statistics of this course. \$name and \$level are character vectors, \$date is a list, and \$maxnumber is a numeric vector (of length 1 in this case). Components of a list can be selected both by their names (using the \$ sign) and by the number of their position. Components or elements of components can be selected by subindexing:

```

> course[[1]]
[1] "Modelling"
> course$level
[1] "masters" "PhD"
> course[[3]][[2]]
[1] "14 June"
> course$date$from
[1] "3 June"
> course$level[2]
[1] "PhD"

```

Note that we use `[[]]` to select components of a list, and `[]` to select elements of a vector. The latter can also be used on list components; however, in this case, the selected component(s) will remain embedded in a list:

```

> course[1]
$name
[1] "Modelling"
> is.list(course[1])
[1] TRUE

```

```
> course[1:2]
$name
[1] "Modelling"
$level
[1] "masters" "PhD"
```

This type of selection is typically needed when you need to select more than one component of a list, which is not possible in vector context.

While the components of a list cannot normally be referenced directly just by their component names (without typing the name of the list and `$`), this can be made possible with the `with()` function:

```
> level
Error: object 'level' not found
> with(course,{level})
[1] "masters" "PhD"
```

The function looks into its first argument (the list `course` in this case), to find the names referenced in its second argument, which is an expression. Expressions (enclosed in curly brackets) can include any number of commands:

```
> with(course,{
+   a <- maxnumber*2
+   b <- date$from
+   c <- name=="Modeling"
+   list(l1=a,l2=b,l3=c)
+ })
$l1
[1] 40

$l2
[1] "3 June"

$l3
[1] FALSE
```

This can save you a lot of typing, if you perform a number of operations on the components of a list (or data frame). Notice that the prompt at the beginning of the lines changed to `+` within the expression: this indicates an incomplete expression, when some brackets (round or curly) have been opened, but not closed yet. The `list()` in the last line of the expression is used to collect and return the results of all the operations; as a general rule, only the value of

the last commandⁿ is returned when an expression is evaluated. Any objects created within the expression will be lost after its evaluation (try to call `a` or `b`).

There is a further way to refer to component names directly. The `attach()` function attaches the object in its argument to the *search path* where R is looking for the meaning of any name that it encounters:

```
> attach(course)
> level;maxnumber
[1] "masters" "PhD"
[1] 20
```

However, use this feature with extreme caution. There are three common pitfalls that you should try to avoid. First, if the list or data frame that you attach has a component with a name that you have also used to create an independent (“global”) data object, then any reference to the name will still call the global object, and not the component of the attached list or data frame^o. As a general rule, do not use the same names for stand-alone data objects and components. Second, `attach()` works by adding a copy of your object to the search path, rather than the original object itself. If you assign a new value to the component of the attached object (using its full name), the value of the attached copy is not going to change:

```
> course$maxnumber <- 5
> maxnumber
[1] 20
> course$maxnumber
[1] 5
```

This can result in confusion and potential errors, just like the opposite situation: if you assign a value using just the component name, it will create a new global object of the same name, holding the new value—which is probably not what you would want. As a general rule, do not change the content of an attached object. Third, the most frequent problem with `attach()` is that it is very easy to forget about it when the attached object is no longer needed. Attached copies of earlier objects can lurk unnoticed in the search path, sometimes causing strange behaviour. It is very important to remove a detached object with `detach()` when it is no longer needed. In particular, re-running your code during program development can easily accumulate multiple (and possibly different!) copies of the same object in the search path, if you forget to (include or) execute the `detach()` command at the end. E.g., if you now attach the same list again:

```
> attach(course)
```

ⁿNote that only non-assignment commands return a value.

^oThis is because the attached object is, per default, added to the second position of the search path, following the `.GlobalEnv` environment that holds the names of the global objects.

The following object is masked from course (position 3):

```
date, level, maxnumber, name
> search()
[1] ".GlobalEnv"      "course"          "course"
[4] "tools:rstudio"    "package:stats"   "package:graphics"
[7] "package:grDevices" "package:utils"   "package:datasets"
[10] "package:methods"  "Autoloader"      "package:base"
```

You get a warning message, but a second copy is attached to the search path, nonetheless, “masking” the values of the earlier attached copy. If you have changed the attached object in the meantime, confusion may easily arise as to which copy is currently in use. ALWAYS clean up; in this case, you need to execute `detach(course)` twice.

The last data structure that we are going to use is the data frame, which is not unlike a cross-breed between a matrix and a list. It is a two-dimensional table of data, like a matrix, but each of its columns can hold a different type of data. The typical use of a data frame is to collect data from a (real or simulated) experiment, with each row holding the data from one experiment or study subject, and the columns holding the different types of data recorded for each experiment/subject. Here is an example:

```
> first.names <- c("John","Sarah","Agnes","Ernest","Angela")
> family.names <- c("Smith","Miller","White","Hillary","Bennett")
> gender <- c("m","f","f","m","f")
> age <- c(15,43,21,76,8)
> df <- data.frame(firstname = first.names, surname = family.names,
  age=age, sex=gender)
> df
  firstname surname age sex
1     John   Smith  15   m
2     Sarah  Miller  43   f
3     Agnes   White  21   f
4   Ernest Hillary  76   m
5   Angela Bennett   8   f
```

Each row (automatically numbered when printed) holds the data on one individual, and the columns hold different data types. While NAs are allowed, each row and column must be complete. A data frame can be indexed both as a matrix or as a list, with the columns corresponding to named components^P:

```
> df$firstname
```

^PNote that if you are using an older version of R preceding R 4.0.0, your output will look slightly different for those columns that hold strings, because these used to be converted to factors per default.

```

[1] "John"    "Sarah"   "Agnes"   "Ernest"  "Angela"
> df$age[2]
[1] 43
> df[,1]
[1] "John"    "Sarah"   "Agnes"   "Ernest"  "Angela"
> df[,3]
[1] 15 43 21 76 8
> df[1,"sex"]
[1] "m"

```

You can use the following commands to obtain some information on the contents of your dataframe:

```

> str(df) # basic structure of the data frame
'data.frame': 5 obs. of 4 variables:
 $ firstname: chr  "John" "Sarah" "Agnes" "Ernest" ...
 $ surname : chr  "Smith" "Miller" "White" "Hillary" ...
 $ age      : num  15 43 21 76 8
 $ sex      : chr  "m" "f" "f" "m" ...
> summary(df) # summary of the data columns
  firstname      surname      age      sex
Length:5      Length:5      Min.   : 8.0  Length:5
Class :character Class :character 1st Qu.:15.0 Class :character
Mode  :character Mode  :character Median :21.0 Mode  :character
                        Mean  :32.6
                        3rd Qu.:43.0
                        Max.   :76.0

```

Indexing by logical vectors is a very efficient way to select those rows (individuals) that fulfil one or more criteria:

```

> df[age>20,]
  firstname surname age sex
2    Sarah  Miller  43   f
3    Agnes  White  21   f
4   Ernest Hillary  76   m

```

Several criteria can be combined. E.g., here we select those who are both males *and* aged 20 or more; then those who are either males *or* aged 20 or more (or both):

```

> df[df$sex=="m" & age>20,]
  firstname surname age sex

```

```

4    Ernest Hillary  76    m

> df[df$sex=="m" | age>20,]
  firstname surname age sex
1      John   Smith  15    m
2     Sarah  Miller  43    f
3     Agnes   White  21    f
4   Ernest Hillary  76    m

```

2.2 Random numbers

Generating random numbers is extremely important for computer simulations, and R offers a large suite of functions for this. Here we provide just two basic examples, and use these also to illustrate how distributions of numeric data can be efficiently inspected in R.

In programming, the most frequent type of random number generation involves drawing a number from a uniform distribution over the interval (0,1). This can be done in R with the `runif(1)` command. However, R provides greater flexibility: the `runif()` function itself can take three arguments, setting the number of random numbers to be generated, and the limits of the interval for the uniform distribution, e.g.⁹:

```

> runif(5,min=1,max=3)
[1] 1.205268 2.410345 1.641227 2.578157 2.952897

```

All statistically important distributions have their generative function in R. For a further example, see the binomial distribution, which describes the number of occurrences of an event, in a series of trials, given a certain probability of “success” (occurrence) per trial. In the following example, we generate random numbers for the occurrences of an event that has a probability of 0.2, given 10 trials in an experiment:

```

> rbinom(5,10,0.2)
[1] 1 3 3 1 3

```

The first parameter instructed R to generate 5 numbers, which have been generated independently from the distribution parameterized by the other two arguments. The resulting vector corresponds to the simulated result of 5 independent experiments, with 10 trials each. For a list of distributions available in R, type `?Distributions`^f.

⁹Of course, you will see different numbers when you execute the command. Each execution generates a different set of random numbers from the same distribution.

^fIn addition to the generative functions that have names like `rxxx` (e.g. `rbinom` or `runif`), there are functions for the density, cumulative distribution and quantile functions of all these distributions with names following the convention `dxxx`, `pxxx` and `qxxx`, respectively.

Another important case of random decisions is choosing a random sample from a set of values. This can be done with the aptly named `sample()` function, which works on vectors of any data type. Per default, sampling occurs without replacement, but the function can also be parameterized to sample with replacement:

```
> sample(2:7,3)
[1] 4 6 7
> fruits <- c("apple","pear","orange")
> sample(fruits,2)
[1] "pear" "apple"
> sample(fruits,6,replace=TRUE)
[1] "pear" "apple" "apple" "pear" "pear" "pear"
```

There is a convenience feature: if `sample()` is called with a first argument that is an integer vector with a single value, it will select integer values from the range 1 to the value of the argument⁸. E.g. `sample(5,3)` will select three values from 1:5 (equivalent to `c(1,2,3,4,5)`).

Now let us generate a large set of random numbers sampled from 1:10 by `sample1 <- sample(10,1000,replace=TRUE)`, and then use the `hist(sample1)` function to plot a histogram of the set, which is a very efficient way of inspecting the distribution of your numeric data.

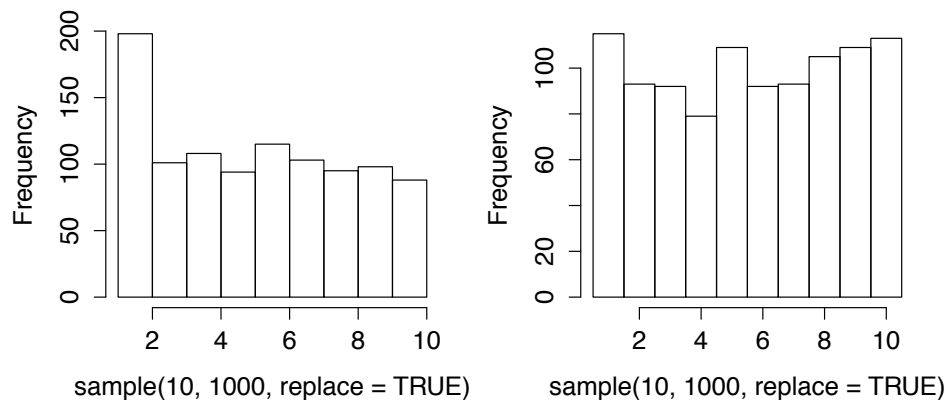


Figure 2: Histogram of random numbers selected with equal probabilities from 1:10. Left panel: automatic breakpoints; right panel: adjusted breakpoints.

The figure will look something like the left panel of Figure 2. The heights of the columns indicate the frequency of numbers falling into disjunct intervals (“bins”) whose limits (break-points) can be read from the horizontal axis. The automated choice of breakpoints is usually quite efficient; however, in this case, the value of 2 was apparently still binned to the lowest

⁸This convenience feature is also a pitfall for programmers. If you have a sampling step acting on a vector of variable length in your program, make sure to handle the situation when the vector happens to hold a single value, say `x`, which should then be sampled automatically, but the function will sample from the expanded vector `1:x` instead.

category, together with values of 1. This can be corrected by supplying an appropriate set of breakpoints^t, e.g., `hist(sample(10,100000,replace=TRUE),breaks=0:10+0.5)` was used to generate the right panel of Figure 2.

2.3 Control structures and logical operators

Control structures allow for conditional or repeated execution of a part of your program code. You can use the `if()` function to make the execution of one or more commands dependent on a condition, e.g.:

```
> x = 4
> if (x>3) print("x is greater than 3")
[1] "x is greater than 3"
> if (x<3) print("x<3") else print("x>=3")
[1] "x>=3"
> if (x%%2 == 0) {
+   print("x is an even number")
+   y = x^2
+ } else print("x is an odd number")
[1] "x is an even number"
> y
[1] 16
```

The condition for the `if` statement must evaluate to a single logical `TRUE` or `FALSE` value^u. Notice the use of `else` to execute code only if the condition is *not true*. Curly brackets can again be used to group any number of commands into an expression, and make the whole block of code within the brackets executed conditionally. The *modulo* operator `%%` in the example finds the remainder of division, in this case the remainder of the division of `x` by 2.

Several conditions can be combined with the logical operators *AND* (`&&`), *OR* (`||`), and *exclusive OR* (`xor()`), while the *negation* operator (`!`) changes `TRUE` to `FALSE` and vice versa. E.g., the following code executes the conditional block in the curly brackets when `x` is both defined (not `NA`) and greater than three:

```
if (!is.na(x) && x>3) {...}
```

The “double” versions of *AND* (`&&`) and *OR* (`||`) inspect only the first element of the condition vectors, and yield a single logical value, which is exactly what a conditional `if()` statement needs. However, both operators have also a short version, which performs element by element operations, and therefore the result is a logical vector of length equal to that of the

^tFor further details of the `hist()` function, see its help `?hist`.

^uIf the condition is a vector with more than one element, only the first element is evaluated, a warning is issued, but the execution of the program is not interrupted.

longest vector in the expression (shorter vectors are recycled with a warning message). Exclusive OR (`xor()`) and negation (`!`) always work elementwise on a vector. Some examples:

```
> 11 <- c(TRUE,TRUE,FALSE,TRUE,FALSE)
> 12 <- c(FALSE,TRUE,FALSE,TRUE,TRUE)
> 11 & 12
[1] FALSE TRUE FALSE TRUE FALSE
> 11 | 12
[1] TRUE TRUE FALSE TRUE TRUE
> 11 && 12
[1] FALSE
> 11 || 12
[1] TRUE
> xor(11,12)
[1] TRUE FALSE FALSE FALSE TRUE
> !11
[1] FALSE FALSE TRUE FALSE TRUE
```

As a general rule, use the long (double) versions of AND and OR in conditional `if()` statements (and look for warning messages when the condition has more than one element, which is probably not what you want in this context), and use the short (single) versions when you use multiple conditions to select elements of a data object (as in the section *Lists and data frames*). For more on logical operators type `help("Logic",package="base")`.

Further control structures let you execute a block of code repeatedly in what is called a “loop”. E.g., in a `for` loop an index variable cycles through all elements of a vector, taking one value at a time, and executes the block of code in the “body” of the loop as many times as there are elements in the vector, e.g.:

```
> for (i in 1:5) {
+   if (i%2==0) {print(i^2)}
+   else {print(i)}
+ }
[1] 1
[1] 4
[1] 3
[1] 16
[1] 5
```

In this example, the index variable is called `i`, and it cycles through elements of the vector `c(1,2,3,4,5)` (created by `1:5`) one after the other. The “body” of the loop consists of two lines (in this case: of a conditional statement), which are executed once for each value of the index variable, i.e. with `i=1` in the first cycle, `i=2` in the second cycle, etc. When the last value of the index vector has been used, the program exits the loop and proceeds to the next line to be

executed. You will see more examples on the use of the `for` loop below. For more information (including further control statements, e.g., `while`, `repeat` or `switch`) type `?Control`.

2.4 Avoiding loops

Loops are very useful structures that are likely to feature heavily in almost any code you are going to write. However, the use of explicit loop statements (as described in the previous subsection) is often not efficient in R when compared to “vectorized calculations”. E.g., remember that arithmetic operations between data objects are carried out element-by-element (where the dimensions of the objects allow this), which makes loops unnecessary in many cases. Compare calculating the elementwise sum of two vectors with an explicit loop and with vectorized arithmetic:

```
> (a1 <- sample(10,10,replace=TRUE))
[1] 7 4 2 8 8 4 5 3 10 4
> (a2 <- sample(10,10,replace=TRUE))
[1] 8 6 9 2 2 4 10 1 8 5
> result <- numeric(length(a1))
> for (i in 1:length(a1)) {
+   result[i] = a1[i] + a2[i]
+ }
> result
[1] 15 10 11 10 10 8 15 4 18 9
>
> # now taking advantage of vectorized calculation:
> a1+a2
[1] 15 10 11 10 10 8 15 4 18 9
```

Wherever available, vectorized calculations yield more concise code, which is easier to read; and also a great improvement in running time, which can be important in more complex cases. Many functions and even logical tests in R are able to operate on vectors and other data objects in an elementwise fashion. In the following example, we wish to process a vector of integer numbers, such that the result should contain the square of all even numbers and the cube of all odd numbers. We first do this by processing each number in turn using a loop:

```
> (a <- sample(10,10,replace=TRUE))
[1] 8 5 3 8 4 9 6 6 1 4
> b <- numeric(length(a))
> for (i in seq(along=a)) {
+   if (a[i]%2==0) b[i]=a[i]^2
+   else b[i]=a[i]^3
+ }
> b
```

```
[1] 64 125 27 64 16 729 36 36 1 16
```

Note that we initialized the result vector (called `b`) so we could assign values to its indexed elements within the loop. The expression `seq(along=a)` creates a vector of integers from one to `length(a)`, which is just what we need to cycle through all values in `av`.

Now we solve the same problem without loop, taking advantage of indexing by logical tests:

```
> b <- a^3
> b[a%%2==0] <- a[a%%2==0]^2
> b
[1] 64 125 27 64 16 729 36 36 1 16
```

Note that we also used the trick of initializing the `b` vector with the cubed values of all the elements in `a`; this way we only needed to overwrite the elements where `a` had an even number. Finally, the same result can also be achieved with a single line using the convenient `ifelse()` function of R^w:

```
> (b <- ifelse(a%%2==0,a^2,a^3))
[1] 64 125 27 64 16 729 36 36 1 16
```

2.5 Writing your own functions

You have already seen a few examples for functions, which generally have the form *functionname(arg1,arg2,...)*. Functions take one or more *arguments* that you need to provide in brackets, perform some operation using these arguments, and then return a value, a vector, or a complex data structure (e.g., a list) as a result. Writing your own functions is simple, and very useful when you need to carry out a block of code repeatedly, using flexible parameterization. Using loops lets you repeat the same actions in one continuous series; embedding the required actions in a function allows you to execute (*call*) this code at multiple (possibly distant) points in your program, and to customize each call by setting the arguments. Let us start with a very simple example:

```
> a <- 1:5
> f1 <- function(x) {
+   x^2
+ }
> f1(a)
[1] 1 4 9 16 25
```

^vAlternatively, we could have used `for (i in 1:length(a))` to equivalent effect.

^wCheck out its help to see how it works.

Functions are defined with the `<-` assignment operator, assigning the “body” of the function (a single command, or any number of commands enclosed in curly brackets) to the name you wish to give to your function (`f1` in this example). In R you do not need to define the type of the argument(s) at the definition of the function: just make sure to include operations that can be carried out on the type of data you had in mind. The function we have just defined calculates the square of all elements in the data provided as its single argument, and can take any argument on which this operation is meaningful (vectors or even arrays of numbers). Importantly, the value that a function call (here `f1(a)`) returns is the value of the last command in the body of the function. Please, note that assignments do not return anything (unless enclosed in brackets). In this case, `x^2` was a non-assignment expression, and its value was duly returned. However, try:

```
> f2 <- function(x) {  
+   result <- x^2  
+ }  
>  
> f2(a)  
>
```

A call to `f2(a)` returns nothing, because the last command was an assignment. Importantly, the vector called `result` that was created within the function call, is also not available (try typing it at the Console). Anything defined within a function exists only within the function and is lost when the function call is finished. To resolve this, you need to collect all the results you wish to obtain from the function call at the end of the function in a non-assignment command. E.g., lists can be used to collect even data of different types:

```
> f3 <- function(x) {  
+   result <- x^2  
+   char <- letters[1:length(x)]  
+   list(res=result,chr=char)  
+ }  
>  
> f3(a)  
$res  
[1] 1 4 9 16 25  
  
$chr  
[1] "a" "b" "c" "d" "e"
```

The `return()` command (all commands are, in fact, functions in R) lets you return values from any point within a function (forcing also an immediate exit from the function); check out its help.

In the definition of a function it is possible to set default values to one or more arguments, and these values will then be assumed if the given arguments are not provided with a function

call. E.g., the following “customized” version of `sample` selects one value, when the size of the sample is not provided (but you can still overwrite this default by providing a value); and samples without replacement:

```
> my.sample <- function(from,howmany=1,replace=FALSE) {  
+   sample(from,howmany,replace=replace)  
+ }  
>  
> my.sample(3:8)  
[1] 5  
> my.sample(3:8,4)  
[1] 6 3 7 4
```

You can print a defined function simply by typing its name (without brackets) in the Console. Try `my.sample` and try also `sample` to see the definition of the original function.

A good strategy to work with functions is to develop the intended code first outside a function, which makes is much easier to experiment with. When you have made sure the code works fine, then you can wrap a function around it. In some cases, it is also efficient to re-use the same function you have written in several different programs. In this case, write your function into a separate file and load it with the `source()` function in your main program. Download `distance.r` from the course website into your current working directory and try:

```
> source("distance.r")  
> point1 <- c(5,3)  
> point2 <- c(1,6)  
> distance(point1,point2)  
[1] 5  
> distance  
function(p1,p2){  
  diff1 = p1[1] - p2[1]  
  diff2 = p1[2] - p2[2]  
  diff = sqrt(diff1^2 + diff2^2)  
  diff
```

The `distance` function that we defined here calculates the Euclidian distance of two points in a 2D coordinate system. Now try to write a function that can do this in a generalized way for points of any dimension (i.e., the coordinates of the points can be defined by vectors of arbitrary length, corresponding to the number of dimensions).

2.6 Getting help

Within R you can call help on a particular function with `?functionname`. A more general search for an expression (searching also in the description of the functions) is possible with

`help.search("expression")`. Finally, in RStudio you can simply move the cursor to any function in your script (in the top left script window) and press <F1> to call up the help text on the function in the Help tab of the bottom right panel.

3 Recommended reading

For further reading on R we recommend the documentation (Manuals, FAQs and R Wiki) available at the R Project website (www.r-project.org/). For the use of the program we recommend in particular the latest version of *An Introduction to R*.

There are also many online courses available; unfortunately, in recent years, even introductory-level courses have mostly been converted to a fee-based model.