

R/BioC Exercises: Introduction to R

Perry Moerland

April 12, 2010

☞ Information on how to log on to a PC in the exercise room and the UNIX server can be found here:

<http://bioinformaticslaboratory.nl/twiki/bin/view/BioLab/EducationBioinformaticsII>

1 Basics

Once you're logged on to your SARA account, you can start R just by typing R 2.10. While on Windows and MacOSX R comes with a graphical user interface, on UNIX the interface is rather minimal.

R is a command line driven environment. This means that you have to type in commands (line-by-line) for it to compute or calculate something. In its simplest form R can therefore be used as a pocket calculator

```
> 2 + 2
```

```
[1] 4
```

Or a little more complicated

```
> sin(log(pi))
```

```
[1] 0.9105985
```

☞ For those of you who have never programmed in R, I would like to urge you to really type in the commands listed in the rest of this document and experiment with them. This way you will remember them more easily.

Luckily R is more than just a calculator, it is a programming language with most of the elements that every programming language has: statements, variables, types, data structures, objects, classes *etc.* The goal of the exercises is to make you familiar with most of these. This document only introduces the main concepts. For more details you are referred to the various R manuals on the course website. Especially “An introduction to R” is worth reading.

1.1 Assignments

The left arrow `<-` is R's way of denoting the assignment statement that assigns a value to a variable.

```
> x <- 2
> x <- x^2
> x
```

```
[1] 4
```

The last line illustrates how you can examine the value of a variable: just type its name.

1.2 Vectors

A vector is an array of elementary objects and can be made using the construct `c(...)`

```
> x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

There are various ways of creating this vector. Here is another example

```
> x <- seq(from = 1, to = 10, by = 1)
```

Actually `c` and `seq` are examples of a *function*. Functions are explained in more detail in section 1.4.

You can calculate with vectors as with ordinary numbers

```
> x + x
```

```
[1] 2 4 6 8 10 12 14 16 18 20
```

```
> x^2
```

```
[1] 1 4 9 16 25 36 49 64 81 100
```

Note that the operations are carried out elementwise.

I With the cursor at the prompt, use the up and down arrows to scroll through previous commands. This is particularly useful if you type something slightly incorrectly the first time and want to bring back the command again to change something minor and re-enter the command.

I If at any stage you find that R is not responding, it has hung or you want to stop a command midway through its execution, type Ctrl-C (that is, push the Ctrl key and the C key at the same time).

1.3 Modes

R has several atomic modes, until now we have only used the *numeric* mode

```
> mode(x)
[1] "numeric"
```

The other two atomic modes are the *logical* mode

```
> x < 3
[1] TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> mode(x < 3)
```

```
[1] "logical"
```

and the *character* mode

```
> letters[x]
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
> mode(letters[x])
[1] "character"
```

Note that characters are specified using double quotes.

1.4 Functions

You have already seen some examples of the use of *functions*. The general syntax of a function call in R is

```
> name.of.function(argument1, argument2, ...)
```

This type of function call corresponds to what is known as *positional matching*. When we type

```
> seq(1, 10, 1)
```

R assumes that the first argument corresponds to the starting value of the vector, the second argument to the end value of the vector, and the third argument to the desired increment. You can use

```
> help(seq)
```

to learn more about *seq*. You can also use *?seq*. As you can see from this help page, functions often have sensible defaults for most of the arguments. This way we could avoid specifying the *length.out* and *along.with* arguments. The *help* function is defined for any function in R and is probably the one you will use the most.

For functions that one uses all the time positional matching is perfectly fine. However, to avoid programming mistakes and to make your code easier to understand it is often better to use *keyword matching*. The equivalent statement would then read

```
> x <- seq(from = 1, to = 10, by = 1)
```

Note that with positional matching the order of the arguments matters, with keyword matching any order is possible

```
> x <- seq(to = 10, from = 1, by = 1)
```

Positional and keyword matching can also be mixed

```
> x <- seq(1, by = 1, to = 10)
```

Note that

- Functions in R are always part of a package such as the `base` package for `seq`
- Only the standard packages are loaded when you start R: `base`, `graphics`, `stats`, and `utils`
- Other packages are loaded by the `library` command
- `library()` shows all installed packages
- `help(package=stats)` gives help on all functions defined in `stats`

Other useful commands for finding help are

- `help.search`
- `RSiteSearch`
- Running `help.start()` launches a web browser that allows the help pages to be browsed with hyperlinks, but try this at home: on our server it is painfully slow

Exercise 1: Using the elementary functions `/`, `-`, `^` and the functions `sum` and `length` calculate the mean

$$\bar{x} = \sum_i x_i/n$$

and the standard deviation

$$\sqrt{\sum_i (x_i - \bar{x})^2 / (n - 1)}$$

of `x=seq(1,10,1)`. You can verify your answer by using the functions `mean` and `sd`.

1.5 Matrices and arrays

A *matrix* in mathematics is just a two-dimensional array of numbers. Many special operations are defined on matrices, some of these will be discussed in the exercises on “Linear modeling and differential expression”. For the moment we will only use matrices as a convenient way of storing tables, say Excel files and the like. The general n -dimensional object is called an *array*.

Matrices can be created as follows

```
> M <- matrix(seq(1, 10, 1), nrow = 5, ncol = 2)
> M
```

```

      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

```

The two dimensions of a matrix are called *rows* and *columns*. Note that a matrix is filled in a columnwise fashion. The dimensions of a matrix can be easily retrieved

```
> dim(M)
```

```
[1] 5 2
```

You can also glue vectors or matrices together to construct a new matrix, either columnwise

```
> cbind(1:5, 6:10)
```

```

      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

```

or rowwise

```
> rbind(c(1, 6), c(2, 7), c(3, 8), c(4, 9), c(5, 10))
```

```

      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

```

Exercise 2: Construct the following matrix using some of the commands you have seen until now. Try to find the shortest solution.

```
> L
```

```

      [,1] [,2] [,3] [,4] [,5]
[1,]   24   22   20   18   16
[2,]   14   12   10    8    6
[3,]    4    2    0   -2   -4
[4,]   -6   -8  -10  -12  -14
[5,]  -16  -18  -20  -22  -24

```

1.6 Naming

One of the most useful concepts in R is the option of naming. We can, for example, give names to rows and columns of M:

```
> rownames(M) <- c("array 1", "array 2", "array 3", "array 4",  
+ "array 5")  
> colnames(M) <- c("gene 1", "gene 2")  
> M
```

```
      gene 1 gene 2  
array 1     1     6  
array 2     2     7  
array 3     3     8  
array 4     4     9  
array 5     5    10
```

1.7 Indexing

Until now we have always worked with entire vectors or matrices. However, often you would like to have access to a particular element of an object. For doing so R has several ways of *indexing*. The basic index operator is `[]`

```
> x[5]
```

```
[1] 5
```

This also works on the left-hand side of an assignment so that you can modify the elements of a vector

```
> x[5] <- 11
```

```
> x
```

```
[1] 1 2 3 4 11 6 7 8 9 10
```

The indexing operator is easily extended to matrices

```
> M[3, 2]
```

```
[1] 8
```

for example, extracts the element on the third row and the second column. You can also extract entire rows (`M[4,]`) or entire columns (`M[,2]`). Another nice feature is the possibility of *negative* indexing. You can get the entire matrix *except* the fourth row

```
> M[-4, ]
```

```

      gene 1 gene 2
array 1      1      6
array 2      2      7
array 3      3      8
array 5      5     10

```

Instead of indexing by numbers, one can also index by names

```
> M[, "gene 1"]
```

```
array 1 array 2 array 3 array 4 array 5
      1      2      3      4      5
```

Booleans can also be used to index (T=TRUE, F=FALSE):

```
> x[c(T, F, T, F, T, F, T, F, T, F)]
```

```
[1]  1  3 11  7  9
```

```
> M[M[, 2] > 7, ]
```

```
      gene 1 gene 2
array 3      3      8
array 4      4      9
array 5      5     10

```

This last command extracts the rows for which the second element is greater than 7.

Exercise 3: How would you insert a 13 between `x[3]` and `x[4]`?

Exercise 4: Load the package that comes with Peter Dalgaard's book (Dalgaard 2002)

```
> library(ISwR)
```

and load one of the data sets in the package

```
> data(juul)
```

This extracts the dataset and stores it in variable `juul`. Investigate the structure of this variable in some more detail using the commands `summary`, `str`, `head`, `tail`. Extract the girls in this data set that are between seven and fourteen years of age. Hint: have a look at the operators `&` and `==`. What are the dimensions of the resulting matrix? What do the NA's occurring in this dataset denote?

Exercise 5: A function that often comes in handy when indexing is `which` that indicates which indices are TRUE in a logical object.

```
> x[which(x <= 7)]
```

```
[1] 1 2 3 4 6 7
```

```
> x[-which(x <= 7)]
```

```
[1] 11  8  9 10
```

Now explain the outcome of

```
> x[-which(x <= 0)]
```

1.8 Factors

It is common in statistical data to have categorical variables indicating some subdivision of data, such as, primary diagnosis, tumor stage *etc.* In R such variables can be specified via *factors*. The related terminology is that a factor has a set of *levels* - say four levels. Internally a four-level factor consists of two items

- a vector of integers between 1 and 4
- a character vector of length 4 containing strings describing the four levels of the factor

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

```
[1] none   severe medium medium mild   severe medium
Levels: none mild medium severe
```

Exercise 6: The *cut* function is used to create a factor from a numeric vector (*help(cut)*). Create a factor in which the *tanner* variable in the *juul* dataset is divided into the three intervals 1, 2-4, 5. Change the level names to “no”, “intermediate”, “late”.

A powerful feature of factors is that you can easily calculate frequency tables

```
> table(fpain)

fpain
  none   mild medium severe
    1     1     3     2
```

1.9 Lists

Until now we have seen objects with elements of the same mode. *Lists* provide a way of mixing different modes.

```
> list(gene = "gene 1", number = 5)
```

```
$gene
[1] "gene 1"
```

```
$number
[1] 5
```

gene and *number* in the previous example are called *components*. Components can also be of different type:

```
> l <- list(genes = c("BRCA1", "BRCA2"), expr = matrix(1:10, 2,
+   5))
> l
```

```
$genes
[1] "BRCA1" "BRCA2"
```

```
$expr
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Lists can be indexed in various ways. Firstly, as vectors using square brackets. This returns a list and is not often used

```
> l[1]
```

```
$genes
[1] "BRCA1" "BRCA2"
```

With double square brackets, this extracts a component

```
> l[[1]]
[1] "BRCA1" "BRCA2"
```

Or equivalently, by name using the `$` operator:

```
> l$genes
[1] "BRCA1" "BRCA2"
```

1.10 Data frames

A special kind of list is a matrix with mixed modes, *e.g.*, rows are individuals and columns correspond to various types of information. In R, this is dealt with by a `data.frame`. Data frames are extremely useful when reading in external data (of the Excel type, let's say) via `read.table`.

Data frames are indexed as expected

```
> d <- data.frame(x = x, y = x + 1)
> d
```

```
   x y
1  1 2
2  2 3
3  3 4
4  4 5
5 11 12
6  6 7
7  7 8
8  8 9
9  9 10
10 10 11
```

```
> d$x
[1] 1 2 3 4 11 6 7 8 9 10

> d[3, 2]

[1] 4
```

Exercise 7: Write the `juul` dataset to a tab-delimited file using `write.table`. If you are not familiar with any UNIX editor, transfer this file to your PC using WinSCP.¹ View the file with a text editor and replace all NA values by `.` and then read the file back again into R with `read.table`.

1.11 Flow control

Until now we have mainly seen simple assignment statements but R is a real programming language. Statements can be grouped

```
> z <- {
+   x <- 2
+   y <- x + 2
+ }
> z

[1] 4
```

R also has a *conditional* construct

```
if (expr_1) expr_2 else expr_3

where expr_1 has to be a Boolean expression

> z <- if (x < 2) 4 else 3
> z

[1] 3
```

Of course R also has an *iterative* construct

```
for (name in expr_1) expr_2
```

- `name` is the loop variable
- `expr_1` is a vector expression such as `1:20`
- `expr_2` is often a grouped expression containing `name`

Let's look at a simple example where we calculate the mean of the columns of our matrix `M`

¹See <http://bioinformaticslaboratory.nl/twiki/bin/view/BioLab/EducationBioinformaticsII> for how to find WinSCP and other programs on your PC.

```

      gene 1 gene 2
array 1     1     6
array 2     2     7
array 3     3     8
array 4     4     9
array 5     5    10

```

```

> results <- numeric(2)
> for (i in 1:2) {
+   results[i] <- mean(M[, i])
+ }
> results

```

```
[1] 3 8
```

Repetitions in R are slow, it is better to *vectorize* your code using *apply*

```
> apply(M, 2, mean)
```

```

gene 1 gene 2
     3     8

```

1.12 Writing functions

Further abstraction is possible through *functions*

```
name <- function(arg_1,arg_2, ...) expr
```

- `expr` is, in general, a grouped expression containing the arguments `arg_1`, `arg_2` ...
- A call to the function is of the form `name(expr_1,expr_2, ...)`
- The value of the expression `expr` is the value returned for the function

Exercise 8: Adapt your answer to Exercise 1 into a function that returns the mean and the standard deviation of a numerical vector.

Here you might want to write your own function *myFirstFunction.R* using a text editor, for example ConTEXT. The ConTEXT editor has syntax highlighting for R which you can select from the dropdown box in the toolbar. Next you can try to save this file and transfer it to your SARA account using WinSCP. Finally, you can load the function into R with `source("myFirstFunction.R")`.

Exercise 9: Generalize your answer to Exercise 3 into a function that inserts a value `v` just after index `k` of a numerical vector.

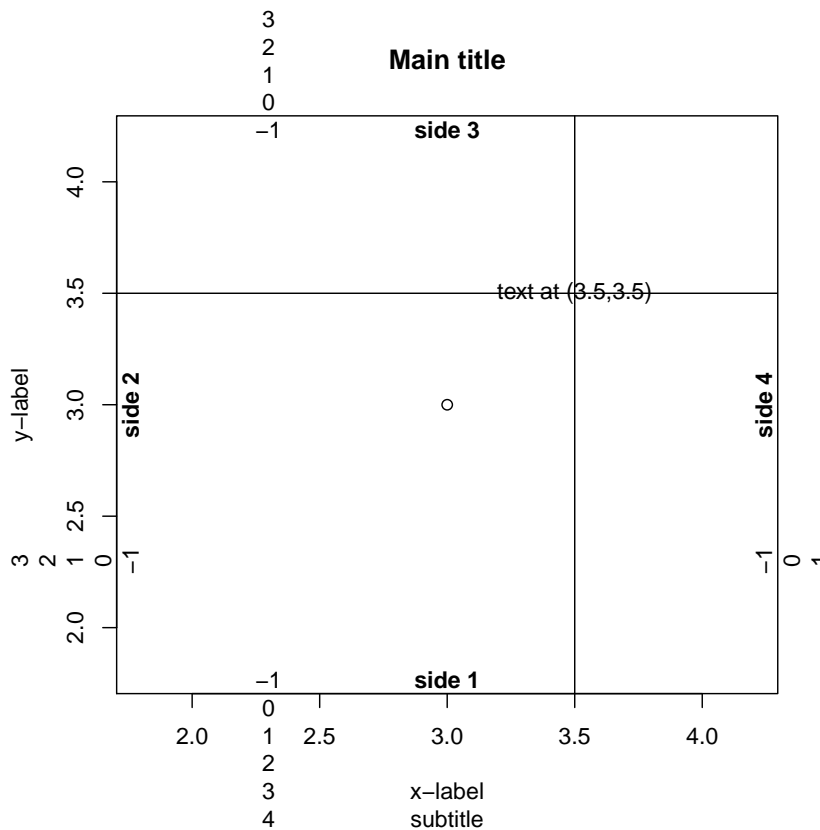
2 Graphics

The graphics subsystem of R offers a very flexible toolbox for high-quality graphing. There are typically three steps to producing useful graphics

- Creating the basic plot
- Enhancing the plot with labels, legends, colors *etc.*
- Exporting the plot from R for use elsewhere

In the graphics model that R uses, a figure region consists of a central plotting region surrounded by margins. The basic graphics command is *plot*. The following piece of code illustrates the most common options²

```
> plot(3, 3, main = "Main title", sub = "subtitle", xlab = "x-label",  
+      ylab = "y-label")  
> text(3.5, 3.5, "text at (3.5,3.5)")  
> abline(h = 3.5, v = 3.5)  
> for (side in 1:4) mtext(-1:4, side = side, at = 2.3, line = -1:4)  
> mtext(paste("side", 1:4), side = 1:4, line = -1, font = 2)
```



²To forward plots from SARA to your PC, you'll need to start Exceed.

In this figure one can see that

- Sides are labelled clockwise starting with x -axis
- Coordinates in the margins are specified in *lines of text*
- Default margins are not all wide enough to hold all numbers -1,...,4

You might want to use the *help* function to investigate some of the other functions and options.

Plots can be further finetuned with the *par* function. For instance, the default margin sizes can be changed using *par*. The default settings are

```
> par("mar")  
[1] 5.1 4.1 4.1 2.1
```

This explains why only side 1 in the above figure had a wide enough margin. This can be remedied by doing

```
> par(mar = c(5, 5, 5, 5))
```

before plotting the above figure. Try this out yourself.

Finally, you might want to save the figure you just made. For this you have to know that R sends graphics to a *device*. The default device on most operating systems is the screen. Saving a figure, therefore, requires changing R's current device. See *help(device)* for the options.

Exercise 10: Save the figure you just made to a *png* and a *pdf* file.

☞ We only covered relatively basic ground in R. If there is still some time left you might want to have a close look at the R manuals listed here <http://bioinformaticslaboratory.nl/twiki/bin/view/BioLab/Lab1>. Especially “An introduction to R” and the “R reference card” will be useful for the rest of this module.

References

Dalgaard, P. (2002). *Introductory Statistics with R*. New York: Springer.

This document was generated with

```
> sessionInfo()  
  
R version 2.10.1 (2009-12-14)  
i386-pc-mingw32  
  
locale:  
[1] LC_COLLATE=English_United Kingdom.1252  
[2] LC_CTYPE=English_United Kingdom.1252  
[3] LC_MONETARY=English_United Kingdom.1252  
[4] LC_NUMERIC=C
```

```
[5] LC_TIME=English_United Kingdom.1252
```

```
attached base packages:
```

```
[1] stats      graphics  grDevices  utils      datasets  methods    base
```

```
other attached packages:
```

```
[1] ISwR_2.0-4
```

```
loaded via a namespace (and not attached):
```

```
[1] tools_2.10.1
```