Quantitative Bootcamp Tutorials T1-T8

Stephanie Palmer and Stefano Allesina

August 10, 2015

Contents

Tutorial 1

Introduction to R – Basics

1.1 What is R?

`R` is a software for statistical analysis. It comes with many built-in functions and excellent graphical capabilities. The main strength of `R` is that it is fully programmable: you can write code in `R` and have the software execute it. This means that it is very easy to automate your statistical and data analysis.

The fact that `R` is easy to program led to the development of thousands of packages, so that you can find a ready-made, specific package for almost any analysis you might want to perform. Because of this strength, `R` has become the most popular statistical software among biologists.

The main hurdle new users face when learning `R` is that it is based on a command-line interface: to make things happen, you write text commands in a "shell", and then the program executes them. This might seem unusual if you come from GUI-based software, where you tend to work by clicking on windows and buttons. However, the command-line is what makes it easy to automate your analysis — all you have to do is collect all the commands in a text file, and then run them in `R`.

For this brief introduction to `R`, we are going to use `RStudio`, a graphical interface that simplifies using `R` by giving you immediate access to the code, the shell, and the graphics.

1.2 Why using R?

Writing scripts for all your work, instead of manually typing commands and clicking buttons, makes your research easy to reproduce (just share the scripts with the interested scientists), well-documented (especially if you write meaningful comments to detail what you are doing), and easy to automate (once written a script to analyze a dataset, it is easy to make it analyze millions of similar datasets).

`R` is free software: it is free to use, but it also gives you the freedom to see the code (open source), modify it, and extend it.

1.3 Launching R

Either click on the `RStudio` icon, or open a terminal and type `rstudio`.

1.4 Finding help

Each command in `R` comes with a manual page. To access it, type `?NAMEOFCOMMAND` in the console (e.g., `?lm`).

1.5 R as a calculator

To start, we are going to explore some features of `R` typing commands in the console. In the console, a "greater sign" (`>`) means that `R` is ready to accept a command. You can navigate the history of the commands you typed by using the arrows on your keyboard.

Go to the console and type:

```
> 1 + 1
[1] 2
> 1 * 3
[1] 3
> 1.7 * 2
[1] 3.4
> 12 / 3
[1] 4
> 12 / 5
[1] 2.4
> 123 - 72
[1] 51
> 2.1 ^ 5
[1] 40.84101
> log(10)
[1] 2.302585
> log10(10)
[1] 1
> sqrt(9)
[1] 3
> trunc(12.11)
[1] 12
> floor(12.11)
[1] 12
> floor(12.71)
[1] 12
> trunc(12.71)
[1] 12
> ceiling(12.71)
```

```
[1] 13
> round(12.71, 1)
[1] 12.7
```

You can use R as a calculator, with the operators:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (remainder of integer division)
x %/% y	integer division

R has many built-in mathematical functions:

Function	Description
abs(x)	absolute value
sqrt(x)	square root
ceiling(x)	nearest integer $> x$
floor(x)	nearest integer $< x$
trunc(x)	integer part
round(x, digits=n)	round the number to n digits
cos(x), sin(x), tan(x), etc.	trigonometric functions
log(x)	natural logarithm
log10(x)	base 10 logarithm
exp(x)	e^x

and it can deal with logical values:

```
> 5 > 3
[1] TRUE
> 5 == (10 / 2)
[1] TRUE
> 6 > 2^4
[1] FALSE
> 6 >= (2 * 3)
[1] TRUE
> (5 > 3) & (7 < 5)
[1] FALSE
> (5 > 3) | (7 < 5)
[1] TRUE
```

1.6 Assignement and data types

When programming in R, you assign values to variables: a variable is a "box" which can contain an variable.

```
> x <- 5
> x * 2
[1] 10
> x <- 7
> x * 2
[1] 14
```

We assigned to the variable `x` the value 5, using the assignment command `<-`. Now we can use `x` to perform operations. If we assign a new value to `x`, the previous value is overwritten.

To list all the variables that you created, type `ls()`. To remove a variable you created, type `rm(NAMEOFVARIABLE)` (e.g., `rm(x)`).

R can handle different types of data:

Type	Description	Example
integer	natural numbers	`x <- as.integer(5)`
numeric	real numbers	`x <- pi`
complex	complex numbers	`x <- 1 + 3i`
logical	TRUE/FALSE	`x <- (5 > 7)`
character	strings	`x <- "hello"`

To determine the type of a variable, use the command `class(x)`. You can also test whether a certain variable is of a certain type by using the functions `is.numeric(x)`, `is.character(x)`, etc.

1.7 Data structures

R has several "data structures", which can be used to organize your data. Each data structure comes with specific operations you can perform.

1.7.1 Vectors, matrices, and arrays

Vectors. The most basic data structure in R is the vector, which is an ordered collection of values. Vectors are defined by concatenating different values with the command `c()`:

```
> x <- c(2, 3, 5, 7, 11, 13, 17, 19)
> x
[1]  2  3  5  7 11 13 17 19
```

You can access the elements of a vector by their index: the first element is indexed at 1, the second at 2, etc.:

```
> x[3]
[1] 5
> x[8]
[1] 19
> x[9]
[1] NA
```

You can extract several elements at once (i.e., another vector), using the colon (`:`) command, or by concatenating the indices:

```
> x[1:3]
[1] 2 3 5
> x[4:7]
[1]  7 11 13 17
> x[c(1,3,5)]
[1]  2  5 11
```

You can find the length of a vector using the function `length(x)`.

Given that `R` was born for statistics, there are several statistical functions you can perform on vectors (# marks comments):

```
> min(x)
[1] 2
> max(x)
[1] 19
> sum(x) # sum all elements
[1] 55
> prod(x) # multiply all elements
[1] 3628800
> median(x) # median value
[1] 9
> mean(x) # arithmetic mean
[1] 9.625
> var(x) # unbiased sample variance
[1] 40.83929
> mean(x^2) - mean(x)^2 # population variance
[1] 35.73438
> summary(x) # print a nice summary
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   4.500   9.000   9.625  14.000  19.000
```

You can generate vectors of sequential numbers using the colon command:

```
> x <- 1:10
```

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

For more complex sequences, use `seq()`:

```
> seq(from = 1, to = 5, by = 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

To repeat a value or a sequence several time, use `rep()`:

```
> rep("abc", 3)
[1] "abc" "abc" "abc"
> rep(c(1,2,3), 3)
[1] 1 2 3 1 2 3 1 2 3
```

Exercise 1.1

1. Create a vector containing all the even numbers between 2 and 100 and store it in variable `z`.
2. What is the sum of all the elements of the vector?
3. Is it equal to $51 \cdot 50$?
4. Does `seq(2, 100, by = 2)` produce the same vector as `(1:50) * 2`?
5. Extract all the elements that z are divisible by 12. How many elements match this criterion?
6. What happens if you type `z ^ 2`?

Matrices. A matrix is a two-dimensional table of values. In case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, etc.):

```
> A <- matrix(c(1, 2, 3, 4), 2, 2) # values, nrows, ncols
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> A %*% A # matrix product
     [,1] [,2]
[1,]    7   15
[2,]   10   22
> solve(A) # matrix inverse
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
> A %*% solve(A)
     [,1] [,2]
```

```
[1,]    1    0
[2,]    0    1
> diag(A) # create a vector with diagonal elements
[1] 1 4
> B <- matrix(1, 3, 2)
> B
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    1    1
> B %*% t(B) # transpose
     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    2    2    2
[3,]    2    2    2
> Z <- matrix(1:9, 3, 3)
> Z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

To determine the dimension of a matrix, use `dim()`:

```
> dim(B)
[1] 3 2
> dim(B)[1]
[1] 3
> dim(B)[2]
[1] 2
```

You can access a particular row/column of a matrix:

```
> Z
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> Z[1,]
[1] 1 4 7
> Z[,2]
[1] 4 5 6
> Z[1:2, 2:3]
     [,1] [,2]
[1,]    4    7
[2,]    5    8
> Z[c(1,3), c(1,3)]
```

```
     [,1] [,2]
[1,]    1    7
[2,]    3    9
```

You can perform operations using all elements of the matrix:

```
> sum(Z)
[1] 45
> mean(Z)
[1] 5
```

Arrays. If you need more than two-dimensional tables, use arrays:

```
> M <- array(1:24, c(4, 3, 2))
> M
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

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

You can still determine the dimensions using

```
> dim(M)
[1] 4 3 2
```

and access the elements as done for the matrices. One thing you should be paying attention to: `R` drops dimensions that are not needed. So, if you access a "slice" of a 3-dimensional array:

```
> M[,,1]
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

you obtain a matrix:

```
> dim(M[,,1])
[1] 4 3
```

Exercise 1.2

For this exercise, we're going to use the data you will explore in the Workshop run by Dr. Leslie Osborne. First, we want to be all in the same directory. Hit CTRL+Shift+H: this will open a window asking to choose the working directory. Please go to
`QBio/Tutorials/1-T3-T5-Intro_to_R/Sandbox`
and click on "Open". Now that's the "working directory", where `R` is working. Now type:
`load("../../../Workshops/Osborne/Data/MTneuron.RData")`
to load the data.

The `load()` function loads data that has been previously saved in `R`. To save your data, simply type `save(obj1, obj2, file = "filename.RData")`, where `obj1` and `obj2` are the objects (you can have as many as you want) to save

Note that you can hit `Tab` to auto-complete the path. You should **always** use `Tab` to complete your paths/filenames, as in this way you can make sure you use long, expressive file names, and that you are not introducing some typos.

If everything went well, you should see the following objects:

```
> ls()
[1] "directions" "dirtune"    "RFmap"      "theta"
```

1. Determine the dimension of `directions`, and `RFmap`. Are these vectors, matrices, or arrays?
2. How many coefficients are in `RFmap`?
3. What is the mean value of the coefficients in `RFmap`? What is the median?
4. How many elements of `RFmap` are > 0?

1.7.2 Lists

Vectors are good if each element is a simple variable (e.g., a number, a string, etc.). Lists are used when each element is a more complex object (e.g., a vector, a matrix, another list!). Each element of the list can be indexed either by its index, or by a label:

```
> mylist <- list(Names = c("a", "b", "c", "d"), Values = c(1, 2, 3))
> mylist
$Names
[1] "a" "b" "c" "d"

$Values
```

```
[1] 1 2 3

> mylist[[1]]
[1] "a" "b" "c" "d"
> mylist[[2]]
[1] 1 2 3
> mylist$Names
[1] "a" "b" "c" "d"
> mylist[["Names"]]
[1] "a" "b" "c" "d"
```

1.7.3 Data frames

Data frames contain data organized like in a spreadsheet. The columns (typically representing different measurements) can be of different types (e.g., a column could be the date of measurement, another the weight of the individual, or the volume of the cell, or the treatment of the sample), while the rows are typically different samples.

When you read a spreadsheet file in R, it is automatically stored as a data frame. The difference between a matrix and a data frame is that in a matrix all the values are of the same type (e.g., all numeric), while in a data frame they can be of different types.

Because typing a data frame by hand would be tedious, let's use a dataset that is already available in R:

```
> data(trees) # Dataset with grith, height and volume of cherry
   trees
> is.data.frame(trees)
[1] TRUE
> dim(trees)
[1] 31  3
> head(trees)
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
> trees$Girth
 [1]  8.3  8.6  8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3
[12] 11.4 11.4 11.7 12.0 12.9 12.9 13.3 13.7 13.8 14.0 14.2
[23] 14.5 16.0 16.3 17.3 17.5 17.9 18.0 18.0 20.6
> trees$Height[1:5]
[1] 70 65 63 72 81
> trees[1:3,]
```

```
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
> trees[1:3,]$Volume
[1] 10.3 10.3 10.2
```

Exercise 1.3

1. What is the average height of the cherry trees?
2. What is the average girth of those that are more than 75 ft tall?
3. What is the maximum height of trees with a volume between 15 and 35 ft^3?

1.8 Reading and writing data

Reading data frames is very easy: simply use the command `read.table()`, which takes as argument the file name, and can be customized to define a delimiter (comma by default), the presence of a header for the column names, etc.

Let's read a file taken from Dr. John Novembre's workshop (make sure you're in the `Sandbox` directory first!):

```
> ch6 <- read.table("../Data/H938_Euro_chr6.geno", header = TRUE)
```

where `header = TRUE` means that we want to take the first line to be a header containing the column names.

How big is this table?

```
> dim(ch6)
[1] 43141     7
```

we have 7 columns, but more than 40k rows! Let's see the first few:

```
> head(ch6)
  CHR        SNP A1 A2 nA1A1 nA1A2 nA2A2
1   6  rs4959515  A  G     0    17   107
2   6   rs719065  A  G     0    26    98
3   6  rs6596790  C  T     0     4   119
4   6  rs6596796  A  G     0    22   102
5   6  rs1535053  G  A     5    39    80
6   6 rs12660307  C  T     0     3   121
```

and the last few:

```
> tail(ch6)
      CHR        SNP A1 A2 nA1A1 nA1A2 nA2A2
```

```
43136   6 rs10946282  C  T      0    16   108
43137   6  rs3734763  C  T     19    56    48
43138   6   rs960744  T  C     32    60    32
43139   6  rs4428484  A  G      1    11   112
43140   6  rs7775031  T  C     26    56    42
43141   6 rs12213906  C  T      1    11   112
```

The data contains the number of homozygotes (nA1A1, nA2A2) and heterozygotes (nA1A2), for a number of SNPs obtained by sequencing European individuals:

CHR The chromosome (6 in this case)

SNP The code for the Single-Nucleotide Polymorphism

A1 One of the alleles

A2 The other allele

nA1A1 The number of individuals with the particular combination of alleles.

Exercise 1.4

1. How many individuals were sampled? Find the maximum of the sum `nA1A1 + nA1A2 + nA2A2`. Notes: you can access the columns by index (e.g., `ch6[,5]`), or by name (e.g., `ch6$nA1A1`, or also `ch6[,"nA1A1"]`).
2. You can try using the function `rowSums` to obtain the same result.
3. For how many SNPs do we have that all sampled individuals are homozygotes (i.e., all `A1A1` or all `A2A2`?
4. For how many SNPs, more than 99% of the sampled individuals are homozygote?

Tutorial 2

Inference

Tutorial 3

Introduction to R – Code flow

3.1 The flow of the program

Now that we're more familiar with `R`, we turn to writing programs. Typically, you will write your programs in a text file (called a script), with extension `.R`. Then, you can run the script in `R` by invoking `source(MyScript.R)`.

When you execute the file, `R` reads the lines of code in order from the top to the bottom. Every time `R` encounters a command, it will execute it. So in its simplest form, an `R` program is simply a sequence of commands.

However, it is often important to modify this simple flow of the code: you might have commands that need to be run only if certain conditions are met; commands that need to be run over several files/datasets; commands that you repeat several times; etc.

In this second tutorial on `R`, we will see how you can modify the flow of a program to suit your needs, and how to organize your code to make it readable and easy to understand.

3.2 Why writing a script?

Before we start, a little motivation on writing scripts. In fact, you can accomplish almost everything you need for your research without writing any — simply type the commands in the shell one at a time. However, organizing your work into well-documented scripts is really important because:

Recycle: you will encounter similar problems in the future, and you will be almost done before you even start.

Automate: you will need to repeat the analysis on a different dataset, or slightly tweak it in response to comments. This will take no time at all.

Document: by writing everything in a script, you'll know exactly what you did to get your results. When you'll be writing the Methods section of your paper, you'll be happy you wrote everything down.

Share: believe it or not, people will read what you write, and try to apply your analysis to their own data. Having a script to share will help with this process.

3.3 Getting started

From now on, we'll write scripts, and save them into the `Sandbox` directory within the `T1-T2-T3-Intro_to_R` directory.

In `RStudio`, hit `CTRL+Shift+N` to start a new script. Remember to save it in the right place, or you'll get confused later on...

To execute the script, click on the `Source` button in the upper-right corner of the area where you see the script, or type `source("Myscript.R")` within the console.

3.4 Branching

The simplest modification of the linear flow of a program is given by conditional branching: if a certain condition is met, then certain commands are executed; otherwise, other commands are executed.

Let's create a new script called `conditional.R` and save it in the `sandbox`. Make sure to set the working directory to the `sandbox`. Now type:

```
z <- readline(prompt = "Enter a number: ")
```

The function `readline()` reads input from the user. It returns a string. Let's convert the string to numbers:

```
z <- readline(prompt = "Enter a number: ")
z <- as.numeric(z)
```

Now we want to determine whether the number is even or odd, and print the answer. If a number z is even, then `z %% 2 == 0`.

```
z <- readline(prompt = "Enter a number: ")
z <- as.numeric(z)

if (z %% 2 == 0){
  print(paste(z, "is even"))
} else {
  print(paste(z, "is odd"))
}
```

The anatomy of the `if` statement:

```
if (a condition is met){
  execute these commands
} else {
  execute these other commands [optional]
}
```

The `paste` function concatenates strings, and the `print` function prints the results to the console.

Let's try to run the script a few times:

```
> source('conditional.R')
Enter a number: 12
[1] "12 is even"
> source('conditional.R')
Enter a number: 27
[1] "27 is odd"
```

Exercise 3.1

Add code to the script so that:

1. If $z > 100$, the program prints z^3
2. If z is divisible by 17, prints $\sqrt{z}$
3. If $z < 10$, prints a vector containing the numbers between 1 and z

3.5 Loops

The second way to modify the flow of the program is to write a loop. A loop is simply a series of commands that is repeated a number of times. For example, you want to run the same analysis on all the samples you collected; you want to plot the results contained in a set of files; you want to test your simulation over a number of parameter sets; etc.

`R` provides you with two ways to loop over commands: the `for` loop, and the `while` loop. Let's start with the `for` loop, which is used to iterate over a vector: for each value of the vector, a series of commands will be run, as shown by the following example, which you can type in a script called `forloop.R`.

```
myvec <- 1:10 # vector with numbers from 1 to 10

for (i in myvec){
  a <- i^2
  print(a)
}
```

In the code above, the variable `i` takes the value of each element of `myvec` in sequence. Then, you can use the variable `i` to perform operations.

For loops are used when you know that you want to perform the analysis using a given set of values (e.g., run over all files of a directory, all samples in your data, all sequences of a fasta file, etc.).

The `while` loop is used when the code is repeated until a certain condition is met, as shown by the following example, which you can type in a script called `whileloop.R`:

```
i <- 1

while (i <= 10){
  a <- i^2
  print(a)
  i <- i + 1
}
```

The script performs exactly the same operations as that for the for loop above. Note that you need to update the value of `i`, (using `i <- i + 1`), otherwise the loop will run forever (infinite loop — to terminate click on the stop sign in the top-right corner of the console).

You can break a loop using the command `break`. For example:

```
i <- 1

while (i <= 10){
  if (i > 5){
    break
  }
  a <- i^2
  print(a)
  i <- i + 1
}
```

Exercise 3.2

What does this do? Try to guess what each loop does, and then write a script to confirm your intuition.

1. Code:

```
z <- seq(1, 1000, by = 3)
for (k in z){
  if (k %% 4 == 0){
     print(k)
```

```
  }
}
```

2. Code:

```
z <- readline(prompt = "Enter a number: ")
z <- as.numeric(z)

isthisspecial <- TRUE
i <- 1
while (i < z){
  if (z %% i == 0){
     isthisspecial <- FALSE
     break
  }
}

if (isthisspecial == TRUE){
  print(z)
}
```

3.6 Functions

We have used many built-in fuctions, such as `length`, `dim`, `print`, `seq`, etc. In `R`, you can write your own functions, so that you can keep your code tidy and well-structured. Each function has the same anatomy:

```
name_of_function <- function(arguments of the function){
   body of the function
   ...
   ...
   return(something) # optional
}
```

3.7 Putting it all together

Tutorial 4

Stochastic processes

Tutorial 5

Introduction to R – Advanced

5.1 Manipulating strings

5.2 Plotting

Tutorial 6

Dynamical systems

Tutorial 7

Introduction to UNIX

Tutorial 8

Data Visualization