

Gabriella Rustici (v1, 23/02/2010)

First steps in R

This tutorial will guide you through your first steps in R. For further reading, we recommend:

- “An introduction to R” by Venables et al., which is available for free from <http://cran.r-project.org>
- “Introductory Statistics with R” by Peter Dalgaard, Springer 2002
- “Bioinformatics and Computational Biology Solutions Using R and Bioconductor” by Gentleman, Carey, Huber, Springer 2005.

1. Preliminaries

1.1 Obtaining and Installing R

If you do not have R installed, you can obtain a copy via <http://www.r-project.org>. It is available for Windows, Mac and most Linux distributions, as well as in source code. You should also get a good text editor with syntax highlighting for R - for a list of recommendations look at the R web site under “Related Projects - R GUIs”.

1.2 Installing R packages

Most packages are available either via CRAN (<http://cran.r-project.org>) and can be installed with the command **install.packages("packagename", dependencies = TRUE)** from within R.

To install Bioconductor (<http://www.bioconductor.org>) packages use these commands:

```
> source("http://www.bioconductor.org/biocLite.R")  
  
> biocLite("packagename")
```

Sometimes you might get a package from the author directly, as a compressed archive. You can install it using the command R CMD INSTALL packagename.tar.gz from the Unix command line or from a zip file in the GUI menu of the Windows version.

1.3 Getting help

There are many ways to access further documentation from within R. Find out what the function **library()** does by using the commands **help(library)** or **?library**. The command **library()** results in a list of R-packages that are already loaded and can be used by you. What happens if you type library without parentheses?

1.4 Navigating the help pages

The command **help.start()** launches a web browser for convenient access to R's manuals and the help pages of all installed packages.

2. Vectors and assignments

Build a vector with entries 1 to 10 and call it **x**. Try these different ways of creating the same vector:

```
> x = 1:10
> assign("x", 1:10)
> x = seq(1, 10, by = 1)
> x = seq(length = 10, from = 1, by = 1)
> x = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

The result, however, is always the same:

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

The operator “=” associates the value of an expression (e.g. “1:10”) on its right hand side with a name on its left hand side (e.g. “x”). The value can then be retrieved at a later point in your program by its name.

If an expression is used without associating it with a name, its value is printed on the screen but not stored for later access.

```
> seq(10, 1, -1)
[1] 10 9 8 7 6 5 4 3 2 1
```

Compare “a = 1:10-1” with “b = 1:(10-1)”.

Type **ls()** or **objects()** to get an overview of the objects in your workspace. Single objects can be removed by **rm(objectname)**. To clean up your whole workspace use **rm(list=ls())**. Remove **a** and **b**.

Let’s have a closer look at the vector **x**. The functions **summary()**, **length()**, **str()**, and **class()**, among others, give you an overview of an object’s properties. The output depends on what type of object it is. For a numeric vector, **summary()** provides information on the distribution of its entries.

```
> summary(x)
  Min.   1st Qu.  Median Mean    3rd Qu.  Max.
 1.00   3.25   5.50   5.50   7.75   10.00
```

```
> length(x)
[1] 10
```

```
> str(x)
num [1:10] 1 2 3 4 5 6 7 8 9 10
```

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

x is a numeric vector of length 10. Its type can be changed:

```
> y = as.character(x)
> y
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
```

```
> class(y)
[1] "character"
```

```
> x = as.numeric(y)
> x
[1] 1 2 3 4 5 6 7 8 9 10
```

3. Vector arithmetic

Build two vectors of length 5 and try some arithmetic operations like +, -, *, /, sum, mean, ^, log, exp, sin, cos, tan, sqrt, abs, max, min, range, prod. These operations work element-wise and can be nested.

```
> x = 5:1
> x
[1] 5 4 3 2 1
```

```
> y = c(4, 7, 3, 7, 6)
> y
[1] 4 7 3 7 6
```

```
> x + y
[1] 9 11 6 9 7
```

```
> x * y
[1] 20 28 9 14 6
```

```
> sin(x) + cos(y)
[1] -1.612567896 -0.002900241 -0.848872489 1.663199681 1.801641271
```

```
> sum(sin(x) + cos(y))
```

```
[1] 1.000500

> sqrt(x)
[1] 2.236068 2.000000 1.732051 1.414214 1.000000

> sqrt(x)^2
[1] 5 4 3 2 1
```

Let's try to calculate a more complex formula. Imagine vector **x** containing your **n** measurements. The sample variance is defined as

```
> sum((x-mean(x))^2)/(length(x)-1)
[1] 2.5

> var(x) # gives the same result!
[1] 2.5
```

The symbol “#” starts a comment in R, meaning the rest of the line will not be evaluated.

4. Vector indexing

Individual elements of a vector can be referenced by giving the name of the vector followed by the subscripts in square brackets. You can also use logical expressions for indexing. Some examples:

```
> x = c(5.6, 5.4, 2, 9, -3.9)
> x
[1] 5.6 5.4 2.0 9.0 -3.9

> x[4]
[1] 9

> x[2:3]
[1] 5.4 2.0

> x[c(1, 3)]
[1] 5.6 2.0

> x[x > 4]
[1] 5.6 5.4 9.0
```

```
> x > 4
[1] TRUE TRUE FALSE TRUE FALSE
```

Negative indices exclude certain elements from the vector, e.g. `x[-3]` is the same as `x` with the third element missing.

```
> x[-3]
[1] 5.6 5.4 9.0 -3.9
```

For "named" vectors, indexing via element names is very convenient.

```
> names(x) = c("first", "second", "third", "fourth")
```

```
> x
first second third fourth <NA>
5.6    5.4    2.0    9.0 -    3.9
```

```
> y = x[c("second", "fourth")]
```

```
> y
second fourth
5.4    9.0
```

Note that `y` inherits the corresponding names from `x`.

5. Matrices

Matrices (or more generally arrays) are multi-dimensional generalizations of vectors. In fact, they are vectors that can be indexed by two or more indices.

```
> M = 1:20
> dim(M) = c(4, 5)
> M
[,1] [,2] [,3] [,4] [,5]
[1,]  1   5   9  13  17
[2,]  2   6  10  14  18
[3,]  3   7  11  15  19
[4,]  4   8  12  16  20
```

The second command assigns a dimension attribute to the vector `M`. This results in `M` being treated as a 4x5-matrix (a matrix with 4 rows and 5 columns) henceforth. We could create the same matrix by

```
> M = matrix(1:20, nrow = 4, ncol = 5)
```

We can index the elements of **M** in the same way we used for vectors. The only difference is that we need two indices in the square brackets, because **M** is two-dimensional. The first index corresponds to the rows, the second to the columns.

```
> M[1,2] # first row, second column of M
[1] 5

> M[1:2,1:2] # the upper left corner of M
[,1] [,2]
[1,] 1 5
[2,] 2 6

> M[1:2,-3] # first two rows but 3rd column missing
[,1] [,2] [,3] [,4]
[1,] 1 5 13 17
[2,] 2 6 14 18

> dim(M[1:2,-3]) # result has 2 rows and 4 columns
[1] 2 4

> M[2,] # second row, all columns
[1] 2 6 10 14 18
```

6. Lists

A list is an object consisting of an ordered collection of other objects known as its components. Components in lists can be of different classes and types (e.g. a character vector, a logical value and a matrix). Components are accessed by double square brackets (in the form **listname[[i]]**), where **i** can be either the numerical index of the element or a character string with its name. As a shortcut, you can also use the **\$** operator, which takes its argument literally and does not evaluate it.

```
> Data = list(measurements = matrix(rnorm(50),10,5),
+ tumor.type = factor(c("ER+","ER-","ER-","ER-","ER+")),
+ differential.genes = c("gene2","gene7","gene9")
+ )

> summary(Data)

Length Class Mode
measurements 50 -none- numeric
tumor.type 5 factor numeric
differential.genes 3 -none- character
```

```
> Data[[3]]
[1] "gene2" "gene7" "gene9"

> Data[["tumor.type"]]
[1] ER+ ER- ER- ER- ER+
Levels: ER- ER+

> Data$differential.genes
[1] "gene2" "gene7" "gene9"
```

What does **Data[[3]][2]** do? Give a command that shows the second row of the measurement matrix in your data without the last entry.

Partial matching can be convenient in an interactive session, but it can also lead to hard to track down bugs in bigger programs - be careful!

```
> Data$dif
[1] "gene2" "gene7" "gene9"
```

7. for-loops and apply

```
> set.seed(0) # initialize the random number generator for reproducibility
> M = matrix(rnorm(15),nrow=3,ncol=5)
```

This results in a 3×5 matrix filled with normal-distributed random numbers. Imagine we need the sum over each row of this matrix. Idea: go through the matrix row by row and compute the sum in each step.

```
for (i in 1:3) {
+ print(sum(M[i, ]))
+ }
```

```
[1] 2.863813
[1] 0.2678195
[1] -1.314142
```

Collect the results of each step in a vector called results:

```
> results = numeric(3)
> for (i in 1:3) {
+ results[i] = sum(M[i, ])
+ }
```

```
> results
[1] 2.8638130 0.2678195 -1.3141423
```

In R, it is often more convenient and elegant to use functions from the `apply` family to perform loops that iterate over every element of a vector or list (**`lapply`** or **`sapply`**), or over every row or column of a matrix:

```
> results2 = apply(M, 1, sum)
```

The second argument of the function **`apply()`** corresponds to the dimensions of the matrix: 1 for the rows and 2 for the columns. Try **`apply(M, 2, sum)`**.

8. Functions

You can easily define your own functions in R. As an example, we are going to write a function that computes the one-sample t-statistic for the testing the null hypothesis that a sample had been drawn from a standard normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$. For a sample vector $x = (x_1, x_2, \dots, x_n)$, this statistic is defined as

$$t = \frac{\bar{x}}{\sqrt{\frac{1}{n(n-1)} \sum_{i=1}^n (x_i - \bar{x})^2}}$$

where $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$. Translate this formula into an R function:

```
> tvalue = function(x) {
+ nx = length(x)
+ tx = mean(x)/sqrt(1/(nx * (nx - 1)) * sum((x - mean(x))^2))
+ return(tx)
+ }
```

`tvalue` is the name of the function and the argument **`x`** is a placeholder for the object that is passed to the function (call by value). The object that will be returned by the function is specified by using **`return()`** (or alternatively the value of the last expression evaluated within the function is being returned). You see that it is possible to call any previously defined function, such as **`mean()`**, within functions.

Once we have defined it, we can use the function **`tvalue`** in R.

```
> s = c(1, 2, 0, 1, -1, 3)
> tvalue(s)
[1] 1.732051
```

Note that a function for computing a one-sample t-test is naturally already available in a statistical programming suite and would not have to be written by the user. Compare the result of our function to the output of `t.test(s)`.

Using `apply`, it is a one-line task to compute the t-statistic for each row of a matrix:

```
> apply(M, 1, tvalue)
[1] 0.8295005 0.2374489 -0.5527173
```

This concludes our short introduction to R.

Don't be put off by its complexity, it is well worth the learning effort.