Basic Computing 1 – Introduction to R

Stefano Allesina

Basic Computing 1

- **Goal:** Introduce the stastical software `R`, and show how it can be used to analyze biological data in an automated, replicable way. Showcase the `RStudio` development environment, illustrate the notion of assignment, present the main data structures available in `R`. Show how to read and write data, how to execute simple programs, and how to modify the stream of execution of a program through conditional branching and looping.
- **Audience:** Biologists with little or no background in programming.
- **Installation:** To complete the tutorial, we will need to install `R` and `RStudio`. To install `R`, follow instructions at `cran.rstudio.com`. Then install `Rstudio` following the instructions at `goo.gl/a42jYE`.

Motivation

When it comes to analyzing data, there are two competing paradigms. First, one could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; second, one could write programs that can be run to perform the analysis of the data, the generation of tables and statistics, and the production of figures automatically.

This latter approach is to be preferred, because it allows for the automation of analysis, it requires a good documentation of the procedures, and is completely replicable.

A few motivating examples:

- You have written code to analyze your data. You receive from your collaborators a new batch of data. With simple modifications of your code, you can update your results, tables and figures automatically.
- A new student joins the laboratory. The new student can read the code and understand the analysis without the need of a lab mate showing the procedure step-by-step.
- The reviewers of your manuscript ask you to slightly alter the analysis. Rather than having to start over, you can modify a few lines of code and satisfy the reviewers.

Here we introduce `R`, which can help you write simple programs to analyze your data, perform statistical analysis, and draw beautiful figures.

What is R?

`R` is a statistical software that is completely programmable. This means that one can write a program (script) containing a series of commands for the analysis of data, and execute them automatically. This approach is especially good as it makes the analysis of data well-documented, and completely replicable.

`R` is free software: anyone can download its source code, modify it, and improve it. The `R` community of users is vast and very active. In particular, scientists have enthusiastically embraced the program, creating thousands of packages to perform specific types of analysis, and adding many new capabilities. You can find

a list of official packages (which have been vetted by R core developers) at goo.gl/SOSDWA; many more are available on GitHub and other websites.

The main hurdle new users face when approaching R is that it is based on a command line interface: when you launch R, you simply open a console with the character > signaling that R is ready to accept an input. When you write a command and press Enter, the command is interpreted by R, and the result is printed immediately after the command. For example,

```
1 + 1
```

```
## [1] 2
```

A little history: R was modeled after the commercial statistical software S by Robert Gentleman and Ross Ihaka. The project was started in 1992, first released in 1994, and the first stable version appeared in 2000. Today, R is managed by the *R Core Team*.

RStudio

For this introduction, we're going to use RStudio, an Integrated Development Environment (IDE) for R. The main advantage is that the environment will look identical irrespective of your computer architecture (Linux, Windows, Mac). Also, RStudio makes writing code much easier by automatically completing commands and file names (simply type the beginning of the name and press Tab), and allowing you to easily inspect data and code.

Typically, an RStudio window contains four panels:

- **Console** This is a panel containing an instance of R. For this tutorial, we will work mainly in this panel.
- **Source code** In this panel, you can write a program, save it to a file pressing Ctrl + S and then execute it by pressing Ctrl + Shift + S.
- **Environment** This panel lists all the variables you created (more on this later); another tab shows you the history of the commands you typed.
- **Plots** This panel shows you all the plots you drew. Other tabs allow you to access the list of packages you have loaded, and the help page for commands (just type help(name_of_command) in the Console) and packages.

How to write an R program

An R program is simply a list of commands, which are executed one after the other. The commands are witten in a text file (with extension .R). When R executes the program, it will start from the beginning of the file and proceed toward the end of the file. Every time R encounters a command, it will execute it. Special commands can modify this basic flow of the program by, for example, executing a series of commands only when a condition is met, or repeating the execution of a series of commands multiple times.

Note that if you were to copy and paste (or type) the code into the **Console** you would obtain exactly the same result. Writing a program is advatageous, however, because it can be automated and shared with other researchers. Moreover, after a while you will have a large code base, so that you can recycle much of your code in several programs.

The most basic operation: assignment

The most basic operation in any programming language is the assignment. In R, assignment is marked by the operator <-. When you type a command in R, it is executed, and the output is printed in the **Console**. For example:

```
sqrt(9)
```

```
## [1] 3
```

If we want to save the result of this operation, we can assign it to a variable. For example:

```
x <- sqrt(9)
x
```

```
## [1] 3
```

What has happened? We wrote a command containing an assignment operator (<-). R has evaluated the right-hand-side of the command (sqrt(9)), and has stored the result (3) in a newly created variable called x. Now we can use x in our commands: every time the command needs to be evaluated, the program will look up which value is associated with the variable x, and substitute it. For example:

```
x * 2
```

```
## [1] 6
```

Types of data

R provides different types of data that can be used in your programs. The basic data types are:

- logical, taking only two possible values: TRUE and FALSE

```
v <- TRUE
class(v)
```

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

- numeric, storing real numbers (actually, their approximations, as computers have limited memory and thus cannot store numbers like π, or even 0.2)

```
v <- 3.77
class(v)
```

```
## [1] "numeric"
```

- integer, storing whole numbers

```r
v <- 23L # the L signals that this should be stored as integer
class(v)
```

```
## [1] "integer"
```

- `complex`, storing complex numbers (i.e., with a real and an imaginary part)

```r
v <- 23 + 5i # the i marks the imaginary part
class(v)
```

```
## [1] "complex"
```

- `character`, for strings, characters and text

```r
v <- 'a string' # you can use single or double quotes
class(v)
```

```
## [1] "character"
```

In `R`, the type of a variable is evaluated at runtime. This means that you can recycle the names of variables. This is very handy, but can make your programs more difficult to read and to debug (i.e., find mistakes). For example:

```r
x <- '2.3' # this is a string
x
```

```
## [1] "2.3"
```

```r
x <- 2.3 # this is numeric
x
```

```
## [1] 2.3
```

Operators and functions

Each data type supports a certain number of operators and functions. For example, numeric variables can be combined with `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `^` (exponentiation). A possibly unfamiliar operator is the modulo (`%%`), calculating the remainder of an integer division:

```r
5 %% 3
```

```
## [1] 2
```

meaning that `5 %/% 3` (5 integer divided by three) is 1 with a remainder of 2. The modulo operator is useful to determine whether a number is divisible for another: if y is divisible by x, then `y %% x` is 0.

Numeric types also support many built-in functions, such as:

- `abs(x)` absolute value
- `sqrt(x)` square root
- `round(x, digits = 3)` round `x` to three decimal digits
- `cos(x)` cosinus (also supported are all the usual trigonometric functions)
- `log(x)` natural logarithm (use `log10` for base 10 logarithms)
- `exp(x)` calculating e^x

Similarly, `character` variables have their own set of functions, such as

- `toupper(x)` make uppercase
- `nchar(x)` count the number of characters in the string
- `paste(x, y, sep = "_")` concatenate strings, joining them using the separator `_`
- `strsplit(x, "_")` separate the string using the separator `_`

Calling a function meant for a certain data type on another will cause errors. If sensible, you can convert a type into another. For example:

```
v <- "2.13"
class(v)
```

```
## [1] "character"
```

```
# if we call v * 2, we get an error.
# to avoid it, we can convert v to numeric:
as.numeric(v) * 2
```

```
## [1] 4.26
```

If sensible, you can use the comparison operators `>` (greater), `<` (lower), `==` (equals), `!=` (differs), `>=` and `<=`, returning a logical value:

```
2 == sqrt(4)
```

```
## [1] TRUE
```

```
2 < sqrt(4)
```

```
## [1] FALSE
```

```
2 <= sqrt(4)
```

```
## [1] TRUE
```

Similarly, you can concatenate several comparison and logical variables using `&` (and), `|` (or), and `!` (not):

```
(2 > 3) & (3 > 1)
```

```
## [1] FALSE
```

```
(2 > 3) | (3 > 1)
```

```
## [1] TRUE
```

Data structures

Besides these simple types, `R` provides structured data types, meant to collect and organize multiple values.

Vectors

The most basic data structure in `R` is the vector, which is an ordered collection of values of the same type. Vectors can be created by concatenating different values with the function `c()` (concatenate):

```
x <- c(2, 3, 5, 27, 31, 13, 17, 19)
x
```

```
## [1]  2  3  5 27 31 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] # what if the element does not exist?
```

```
## [1] NA
```

`NA` stands for Not Available. Other special values are `NaN` (Not a Number, e.g., `0/0`), `Inf` (Infinity, e.g., `1/0`), and `NULL` (variable not set). You can test for special values using `is.na(x)`, `is.infinite(x)`, etc.

Note that in `R` a single number (string, logical) is a vector of length 1 by default. That's why if you type `3` in the console you see `[1] 3` in the output.

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

```
x[1:3]
```

```
## [1] 2 3 5
```

```
x[4:7]
```

```
## [1] 27 31 13 17
```

```
x[c(1,3,5)]
```

```
## [1]  2  5 31
```

You can also use a vector of logical variables to extract values from vectors. For example, suppose we have two vectors:

```
sex <- c("M", "M", "F", "M", "F") # sex of Drosophila
weight <- c(0.230, 0.281, 0.228, 0.260, 0.231) # weigth in mg
```

and that we want to extract only the weights for the males.

```
sex == "M"
```

```
## [1]  TRUE  TRUE FALSE  TRUE FALSE
```

returns a vector of logical values, which we can use to subset the data:

```
weight[sex == "M"]
```

```
## [1] 0.230 0.281 0.260
```

Given that `R` was born for statistics, there are many statistical functions you can perform on vectors:

```
length(x)
```

```
## [1] 8
```

```
min(x)
```

```
## [1] 2
```

```
max(x)
```

```
## [1] 31
```

```
sum(x) # sum all elements
```

```
## [1] 117
```

```
prod(x) # multiply all elements
```

```
## [1] 105436890
```

```
median(x) # median value
```

```
## [1] 15
```

```
mean(x) # arithmetic mean
```

```
## [1] 14.625
```

```
var(x) # unbiased sample variance
```

```
## [1] 119.4107
```

```
mean(x ^ 2) - mean(x) ^ 2 # population variance
```

```
## [1] 104.4844
```

```
summary(x) # print a summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00    4.50   15.00   14.62   21.00   31.00
```

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 times, 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:

- Create a vector containing all the even numbers between 2 and 100 (inclusive) and store it in variable `z`.
- Extract all the elements of `z` that are divisible by 12. How many elements match this criterion?
- What is the sum of all the elements of `z`?
- Is it equal to $51 \cdot 50$?
- What is the product of elements 5, 10 and 15 of `z`?
- Create a vector `y` that contains all numbers between 0 and 30 that are divisible by 3. Find the five elements of `y` that are also elements of `z`.
- Does `seq(2, 100, by = 2)` produce the same vector as `(1:50) * 2`?
- 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) # this should return the identity matrix
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
B <- matrix(1, 3, 2) # you can fill the whole matrix with a single number (1)
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) # by default, matrices are filled by column
Z
```

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

To determine the dimensions of a matrix, use `dim`:

```
dim(B)
```

```
## [1] 3 2
```

```
dim(B)[1]
```

```
## [1] 3
```

```
nrow(B)
```

```
## [1] 3
```

```
dim(B)[2]
```

```
## [1] 2
```

```
ncol(B)
```

```
## [1] 2
```

Use indices to 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, ] # first row
```

```
## [1] 1 4 7
```

```
Z[, 2] # second column
```

```
## [1] 4 5 6
```

```
Z [1:2, 2:3] # submatrix with coefficients in first two rows, and second and third column
```

```
##      [,1] [,2]
## [1,]    4    7
## [2,]    5    8
```

```
Z[c(1,3), c(1,3)] # indexing non-adjacent rows/columns
```

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

Some operations use all the elements of the matrix:

```
sum(Z)
```

```
## [1] 45
```

```
mean(Z)
```

```
## [1] 5
```

Arrays

If you need tables with more than two dimensions, 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 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
```

This can be problematic, for example, when your code expects an array and `R` turns your data into a matrix (or you expect a matrix but find a vector). To avoid this behavior, add `drop = FALSE` when subsetting:

```
dim(M[,,1, drop = FALSE])
```

```
## [1] 4 3 1
```

Lists

Vectors are good if each element is of the same type (e.g., numbers, strings). Lists are used when we want to store elements of different types, or more complex objects (e.g., vectors, matrices, even lists of lists). Each element of the list can be referenced 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]] # access first element using index
```

```
## [1] "a" "b" "c" "d"
```

```
mylist[[2]] # access second element by index
```

```
## [1] 1 2 3
```

```
mylist$Names # access second element by label
```

```
## [1] "a" "b" "c" "d"
```

```
mylist[["Names"]] # another way to access by label
```

```
## [1] "a" "b" "c" "d"
```

```
mylist[["Values"]][3]  # access third element in second vector
```

```
## [1] 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 typically represent 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 each column can be of a different type.

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

```
data(trees) # Girth, height and volume of cherry trees
str(trees) # structure of data frame
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

```
ncol(trees)
```

```
## [1] 3
```

```
nrow(trees)
```

```
## [1] 31
```

```
head(trees) # print the first few rows
```

```
##   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 # select column by name
```

```
##  [1]  8.3  8.6  8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3 11.4 11.4 11.7
## [15] 12.0 12.9 12.9 13.3 13.7 13.8 14.0 14.2 14.5 16.0 16.3 17.3 17.5 17.9
## [29] 18.0 18.0 20.6
```

```
trees$Height[1:5] # select column by name; return first five elements
```

```
## [1] 70 65 63 72 81
```

```
trees[1:3, ] #select rows 1 through 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 # select rows 1 through 3; return column Volume
```

```
## [1] 10.3 10.3 10.2
```

```
trees <- rbind(trees, c(13.25, 76, 30.17)) # add a row
trees_double <- cbind(trees, trees) # combine columns
colnames(trees) <- c("Circumference", "Height", "Volume") # change column names
```

Exercise:

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

Reading and writing data

In most cases, you will not generate your data in `R`, but import it from a file. By far, the best option is to have your data in a comma separated value text file or in a tab separated file. Then, you can use the function `read.csv` (or `read.table`) to import your data. The syntax of the functions is as follows:

```
read.csv("MyFile.csv") # read the file MyFile.csv
read.csv("MyFile.csv", header = TRUE) # The file has a header.
read.csv("MyFile.csv", sep = ';') # Specify the column separator.
read.csv("MyFile.csv", skip = 5) # Skip the first 5 lines.
```

Note that columns containing strings are typically converted to *factors* (categorical values, useful when performing regressions). To avoid this behavior, you can specify `stringsAsFactors = FALSE` when calling the function.

Similarly, you can save your data frames using `write.table` or `write.csv`. Suppose you want to save the data frame `MyDF`:

```
write.csv(MyDF, "MyFile.csv")
write.csv(MyDF, "MyFile.csv", append = TRUE) # Append to the end of the file.
write.csv(MyDF, "MyFile.csv", row.names = TRUE) # Include the row names.
write.csv(MyDF, "MyFile.csv", col.names = FALSE) # Do not include column names.
```

Let's look at an example: Read a file containing data on the 6th chromosome for a number of Europeans (Data adapted from Stanford HGDP SNP Genotyping Data by John Novembre):

```
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 43,141 single nucleotide polymorphisms (SNPs) obtained by sequencing European individuals:

- `CHR` The chromosome (6 in this case)
- `SNP` The identifier of 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:

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

Conditional branching

Now we turn to writing actual programs in the **Source code** panel. To start a new `R` program, press `Ctrl + Shift + N`. This will open an `Untitled` script. Save the script by pressing `Ctrl + S`: save it as `conditional.R` in the directory `basic_computing_1/code/`. To make sure you're working in the directory where the script is contained, on the menu on the top choose `Session -> Set Working Directory -> To Source File Location`.

Now type the following script:

```
print("Hello world!")
x <- 4
print(x)
```

and execute the script by pressing `Ctrl + Shift + S`. You should see `Hello World!` and `4` printed in your console.

As you saw in this simple example, when `R` executes the program, it starts from the top and proceeds toward the end of the file. Every time it encounters a command (for example, `print(x)`, printing the value of `x` into the console), it executes it.

When we want a certain block of code to be executed only when a certain condition is met, we can write a conditional branching point. The syntax is as follows:

```
if (condition is met){
  # Execute this block of code
} else {
  # Execute this other block of code
}
```

For example, add these lines to the script `conditional.R`, and run it again:

```
print("Hello world!")
x <- 4
print(x)
if (x %% 2 == 0){
  my_message <- paste(x, "is even")
} else {
  my_message <- paste(x, "is odd")
}
print(my_message)
```

We have created a conditional branching point, so that the value of `my_message` changes depending on whether `x` is even (and thus the remainder of the integer division by 2 is 0), or odd. Change the line `x <- 4` to `x <- 131` and run it again.

Exercise: What does this do?

```
x <- 36
if (x > 20){
  x <- sqrt(x)
} else {
  x <- x ^ 2
}
if (x > 7) {
  print(x)
} else if (x %% 2 == 1){
  print(x + 1)
}
```

Looping

Another way to change the flow of the program is to write a loop. A loop is simply a series of commands that are repeated a number of times. For example, you want to run the same analysis on different data sets that 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 blocks of commands: the `for` loop, and the `while` loop. Let's start with the `for` loop, which is used to iterate over a vector (or a list): 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 new 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. Inside the block defined by the `for` loop, you can use the variable `i` to perform operations.

The anatomy of the `for` statement:

```
for (variable in list_or_vector) {
  execute these commands
} # automatically moves to the next value
```

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 commands need to be repeated while a certain condition is true, 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 we wrote 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 button in the top-right corner of the console). The anatomy of the `while` statement:

```
while (condition is met) {
  execute these commands
} # beware of infinite loops: remember to update the condition!
```

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

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

```
z <- readline(prompt = "Enter a number: ")
z <- as.numeric(z)
isthisspecial <- TRUE
i <- 2
while (i < z) {
  if (z %% i == 0) {
     isthisspecial <- FALSE
     break
  }
  i <- i + 1
}
if (isthisspecial == TRUE) {
  print(z)
}
```

Useful Functions

We conclude with a list of useful functions that will help you write your programs:

- `range(x)`: minimum and maximum of a vector `x`
- `sort(x)`: sort a vector `x`
- `unique(x)`: remove duplicate entries from vector `x`
- `which(x == a)`: returns a vector of the indices of `x` having value `a`
- `list.files("path_to_directory")`: list the files in a directory (current directory if not specified)
- `table(x)` build a table of frequencies

Exercises: What does this code do? For each snippet of code, first try to guess what will happen. Then, write a script and run it to confirm your intuition.

```
v <- c(1,3,5,5,3,1,2,4,6,4,2)
v <- sort(unique(v))
for (i in v){
  if (i > 2){
    print(i)
  }
  if (i > 4){
    break
  }
}
```

```
x <- 1:100
x <- x[which(x %% 7 == 0)]
```

```
my_files <- sort(list.files("../data/Saavedra2013/", full.names = TRUE))
for (f in my_files){
  M <- read.table(f)
  print(paste("The file", basename(f), "contains a matrix with", nrow(M),
  "rows and ", ncol(M), "columns. There are", sum(M == 1),
  "coefficients that are 1 and", sum(M == 0), "that are 0."))
}
```

```
my_amount <- 10
while (my_amount > 0){
  my_color <- NA
  while(is.na(my_color)){
    tmp <- readline(prompt="Do you want to bet on black or red? ")
    tmp <- tolower(tmp)
    if (tmp == "black") my_color <- "black"
    if (tmp == "red") my_color <- "red"
    if (is.na(my_color)) print("Please enter either red or black")
  }
  my_bet <- NA
  while(is.na(my_bet)){
    tmp <- readline(prompt="How much do you want to bet? ")
    tmp <- as.numeric(tmp)
    if (is.numeric(tmp) == FALSE){
      print("Please enter a number")
    } else {
      if (tmp > my_amount){
        print("You don't have enough money!")
      } else {
        my_bet <- tmp
```

```
        my_amount <- my_amount - tmp
      }
    }
  }
  lady_luck <- sample(c("red", "black"), 1)
  if (lady_luck == my_color){
    my_amount <- my_amount + 2 * my_bet
    print(paste("You won!! Now you have", my_amount, "gold dobloons"))
  } else {
    print(paste("You lost!! Now you have", my_amount, "gold dobloons"))
  }
}
print("I told you this was not a good idea...")
```

Exercises in groups

Nobel nominations

The file `../data/nobel_nominations.csv` contains the nominations to the Nobel prize from 1901 to 1964. There are three columns (the file has no header): a) the field (e.g. `Phy` for physics), b) the year of the nomination, c) the id and name of the nominee.

- Take your favorite field. Who received most nominations?
- Who received nominations in more than one field?
- Take the field of physics. Which year had the largest number of nominees?
- What is the average number of nominees for each field? Calculate the average number of nominee for each field across years.

Hints

- You will need to subset the data. To make operations clearer, you can give names to the columns. For example, suppose you stored the data in in a data frame called `nobel`. Then `colnames(nobel) <- c("Field", "Year", "Nominee")` will do the trick.
- The simplest way to obtain a count from a vector is to use the command `table`. The command `sort(table(my_vector))` produces a table of the occurrences in `my_vector` sorted from few to many counts.
- You can build a table using more than one vector. For example, store the Nobel nominations in the data frame `nobel`, and name the columns as suggested above. The command `head(table(nobel$Nominee, nobel$Field))` will build a table with the number of nominations in each field (column) for each nominee (rows).