BIO 754 - Lecture Week 01

19-02-2017

- R is a programming language for statistical analysis,
- It is easier to manipulate data in R than it is in SPSS or Excel,
- Free as opposed to MATLAB,
- There are many libraries/packages especially for biological analysis (e.g Bioconductor),
- Functions (especially the ones in base or stats packages) have comprehensive help files,
- Enables object-oriented programming,
- Nice, flexible graphical outputs,
- We can execute commands on command line so we can go step by step. Compilation is not necessary.

R also has some limitations. For example, it can be very slow for very large datasets. It cannot process files line by line as opposed to python or perl. E.g. for processing high throughput sequencing data output you should prefer one of these languages.

Some important notes before starting: * Saving your code in a text file is a good practice. You should write your code not in the command line but in a text editor and save it. * Here we will be using an Integrated Development Environment (IDE) for R called RStudio, which helps writing code and saving output. You will open a new R Script file (File–>New File–>R Script), write your code in this file, and send it to the console (command line) by using Command+Enter or Cntrl+Enter. * Your homework will be prepared using RMarkdown.

The most simple use of R is using it as a command line calculator:

```
2 + 100
```

```
## [1] 102
```

```
2 + 100 * 3
```

```
## [1] 302
```

As you notice the second command includes two functions and R has a built-in hierarchy giving priority to multiplication. This you can change using parantheses:

```
(2 + 100) * 3
```

```
## [1] 306
```

Variables

The next step is to create variables, a type of computer **object**. This is a piece of data you store in the working memory that can be retrieved and manipulated. Every variable has a **name** (identifier) and a **value**, stored at a certain location (which we do not deal with).

In R you can use `<-` or `=` to assign values to variables (which are equivalent in most cases). For example, let's create a variable called `g`:

```r
g = 2
g
```

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

So the value is stored in memory. What if we searched for a variable called `h`?

```r
h
```

```
## Error in eval(expr, envir, enclos): object 'h' not found
```

We can do mathematical calculations on numeric variables. `g` is a numeric variable:

```r
g + 2
```

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

Still, after this calculation when we retrieve `g`, we get the original value:

```r
g
```

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

But if we assign new value to variable `g` and overwrite it, we cannot retrieve the old value:

```r
g = 5  # g is no longer 2
g
```

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

```r
g * g  # the square root of new 5
```

```
## [1] 25
```

R Syntax

The code chunk above includes the `#` in it, which commands R to ignore what is written after. It is generally used to take notes for people reading our code.

```r
# Nobody cares what I write here. But not in the below line:
Nobody cares
```

```
## Error: <text>:2:8: unexpected symbol
## 1: # Nobody cares what I write here. But not in the below line:
## 2: Nobody cares
##           ^
```

R also ignores spaces and new lines. Let us examine the commands below:

```
g+5
```

```
## [1] 10
```

```
g + 5
```

```
## [1] 10
```

```
g     +
  5
```

```
## [1] 10
```

But there are rules. For example, small and capital case letters are not equivalent. And all variable names have to start with a letter (not a number):

```
G
1g = 5
```

```
## Error: <text>:2:2: unexpected symbol
## 1: G
## 2: 1g
##     ^
```

Meanwhile, variable names can *contain* numbers, points and underscore (but not commas):

```
g1.b_2017 = 10
g1.b_2017
```

```
## [1] 10
```

You can also run multiple commands in the same line by separating them with semi colons:

```
g2.b_2017 = 12 ;  g2.b_2017 = 20
g2.b_2017
```

```
## [1] 20
```

The first command created `g2.b_2017` with value `12`, but the second command *overwrote* that `12` by `20`, and the latter is what you retrieved.

Two small notes: We suggested writing your code in the R Script editor and sending it into the R Console. But if you're in the Console, you can use up or down arrow keys to navigate through your last commands.

If we have written a half-code and sent it to the Console but you decided you do not want to execute anymore, you can skip that line by pressing ESC in RStudio and in the Windows R Console (and Ctrl+C in UNIX-based R Console).

Built-in functions

In R in addition to simple mathematical functions there are built-in functions that allow you to manipulate objects, calculate many useful statistics, and produce graphics. Let's learn a very simple one: `sum`:

```
sum(3, 5, -10.08)
```

```
## [1] -2.08
```

Observe that functions in R work receive their arguments (the input data) within **parantheses**, and the arguments are always separated by **commas**. You can input data directly, as above, or use variables that contain the data:

```
j = 3; k = -10.08
sum(g, j, k)  # this also works
```

```
## [1] -2.08
```

```
mean(g, j  k) # this will not work
```

```
## Error: <text>:1:12: unexpected symbol
## 1: mean(g, j  k
##                ^
```

To start a web browser for built-in help pages you can use `?`:

```
?mean
?sum
```

Variable classes

The variable `g` was a number. We can also create variables that contain text strings. E.g. let's create a variable called `x` that contains the word `nurbahar`.

```
x = nurbahar
```

```
## Error in eval(expr, envir, enclos): object 'nurbahar' not found
```

This will not work, as R seeks for an object called nurbahar In order to create a character string, we need quotation marks (notice how the color of the text changes in the R Script editor):

```
x = "nurbahar"
x
```

```
## [1] "nurbahar"
```

Can we apply the same functions we did on `g` on `x` as well?

```
x * x
```

```
## Error in x * x: non-numeric argument to binary operator
```

```r
x + 2
```

```
## Error in x + 2: non-numeric argument to binary operator
```

There is an R function called `class`, which gives the class (type) of an object.

```r
class(g)
```

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

```r
class(x)
```

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

```r
class(k)
```

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

Functions like summation are defined for only data of certain classes and will not work on others. As you see class of `x` is `character` and you cannot use mathematical operations on it.

Meanwhile some functions can be used on both such as `nchar`, which counts the number of characters in a word or number:

```r
x
```

```
## [1] "nurbahar"
```

```r
nchar(x)
```

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

```r
k
```

```
## [1] -10.08
```

```r
nchar(k)
```

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

Workspace and List of objects

To close R you can use the `quit()` or its `q()` alias. If you say y as an answer, it will save the current workspace and it will be opened in your next R session. In RStudio you can also do this by clicking on the quit button.

The `getwd()` function helps you find the current working directory. This is the directory is where your variables, files, etc. are saved unless you provide an alternative path while saving them. The w.d. you can change using the `setwd()` function.

```
getwd()
```

```
## [1] "/Users/msomel/Documents/misc/metu/ders/2380754_comp_2017"
```

```
setwd("/Users/msomel/Music/")
```

You can also see the files in your w.d. using the `list.files()` function:

```
list.files()
```

```
##  [1] "BIO754 Student list.docx"      "BIOL 754 Section 1 Grades.ods"
##  [3] "Empirical_Rule.png"            "GSE17274_ReadCountPerLane.txt"
##  [5] "attendance BIOL 754.ods"       "homework_week_01.Rmd"
##  [7] "homework_week_01.html"         "homework_week_02.Rmd"
##  [9] "homework_week_02.html"         "homework_week_03.Rmd"
## [11] "homework_week_03.html"         "lecture_week_01.Rmd"
## [13] "lecture_week_01.html"          "lecture_week_01.pdf"
## [15] "lecture_week_02.Rmd"           "lecture_week_02.html"
## [17] "lecture_week_02.pdf"           "lecture_week_03.Rmd"
## [19] "lecture_week_03.html"          "lecture_week_04.Rmd"
## [21] "lecture_week_04.html"          "lecturenotes"
## [23] "r code week 1.R"               "stat test.docx"
## [25] "stat test.pdf"                 "syllabus_2380754_2017.docx"
## [27] "syllabus_2380754_2017.pdf"
```

Meanwhile, to list the objects present in the R workspace (or global environment):

```
ls()
```

```
## [1] "g"         "g1.b_2017" "g2.b_2017" "j"         "k"         "x"
```

```
xx = "myNewObject"
ls()
```

```
## [1] "g"         "g1.b_2017" "g2.b_2017" "j"         "k"         "x"
## [7] "xx"
```

Another useful function is `rm()` or `remove()` used to remove objects (it actually does not delete the object from memory but removes the pointer so that the memory allocated for that object can be overwritten and used for other objects):

```
rm(xx)
ls()
```

```
## [1] "g"         "g1.b_2017" "g2.b_2017" "j"         "k"         "x"
```

Creating and combining vectors using colon and c()

There are several ways to create a vector, which is an object that contains a list of values. The simplest way is using `:` for creating an integer vector:

```r
4:10
```

```
## [1]  4  5  6  7  8  9 10
```

You can also create the vector and save it in memory under a name:

```r
k = 4:10   # now the value of k above is also overwritten
class(k)
```

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

And check its length or calculate its mean:

```r
length(x)
```

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

```r
mean(x)
```

```
## Warning in mean.default(x): argument is not numeric or logical: returning
## NA
```

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

```r
length(g)
```

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

```r
mean(g)
```

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

When you apply a function on a vector R applies the function to every element, one by one:

```r
k + 2
```

```
## [1]  6  7  8  9 10 11 12
```

```r
k * k
```

```
## [1]  16  25  36  49  64  81 100
```

A common way to create a vector is using the c() function, we can combine different objects into a vector:

```r
z = c(5, 10, 2, 1, 10)
z
```

```
## [1]  5 10  2  1 10
```

You can also combine already existing vectors:

```
z2 = c(z, k, z)
z2
```

```
##  [1]  5 10  2  1 10  4  5  6  7  8  9 10  5 10  2  1 10
```

```
length(z2)
```

```
## [1] 17
```

```
class(z2)
```

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

How about combining variables of different classes?

```
z3 = c(z2, x)
z3
```

```
##  [1] "5"         "10"        "2"         "1"         "10"        "4"
##  [7] "5"         "6"         "7"         "8"         "9"         "10"
## [13] "5"         "10"        "2"         "1"         "10"        "nurbahar"
```

```
class(z3)
```

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

All numbers are converted into character.

```
z3 * 10   # doesn't work
```

```
## Error in z3 * 10: non-numeric argument to binary operator
```

The numbers can be back-converted using the function `as.numeric`:

```
as.integer(z3)
```

```
## Warning: NAs introduced by coercion
```

```
##  [1]  5 10  2  1 10  4  5  6  7  8  9 10  5 10  2  1 10 NA
```

But then you get an `NA` for the strings which cannot be converted. `NA`s are special variables:

```
z4 = as.integer(z3)
```

```
## Warning: NAs introduced by coercion
```

```r
z4 * 2
```

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

```r
NA + 100
```

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

Creating vectors using rep() and seq():

An alternative way to create a vector is using the repeat function; `rep(a,b)`, which is used for creating a vector consisting of `a`s repeated for `b` times.

```r
rep(1, 5)
```

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

```r
rep("masallah", 41)
```

```
##  [1] "masallah" "masallah" "masallah" "masallah" "masallah" "masallah"
##  [7] "masallah" "masallah" "masallah" "masallah" "masallah" "masallah"
## [13] "masallah" "masallah" "masallah" "masallah" "masallah" "masallah"
## [19] "masallah" "masallah" "masallah" "masallah" "masallah" "masallah"
## [25] "masallah" "masallah" "masallah" "masallah" "masallah" "masallah"
## [31] "masallah" "masallah" "masallah" "masallah" "masallah" "masallah"
## [37] "masallah" "masallah" "masallah" "masallah" "masallah"
```

You can also use variable names as arguments:

```r
rep(x, 2)
```

```
## [1] "nurbahar" "nurbahar"
```

```r
rep(x, j)   # j was defined as 3 above
```

```
## [1] "nurbahar" "nurbahar" "nurbahar"
```

```r
rep(k, 10)
```

```
##  [1]  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5
## [24]  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7
## [47]  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9
## [70] 10
```

Check the help page:

```r
?rep
```

As shown in the examples at the end of the `rep` help page, you can use the function in different ways by its different arguments:

```r
rep(k, times=10)
```

```
##  [1]  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5
## [24]  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7
## [47]  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4  5  6  7  8  9
## [70] 10
```

```r
rep(k, each=10)
```

```
##  [1]  4  4  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5  5  5  6  6  6
## [24]  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  7  8  8  8  8  8  8
## [47]  8  8  8  8  9  9  9  9  9  9  9  9  9  9 10 10 10 10 10 10 10 10 10
## [70] 10
```

```r
rep(k, length.out=15)
```

```
##  [1]  4  5  6  7  8  9 10  4  5  6  7  8  9 10  4
```

The `times` argument of `rep` can itself accept a vector as input:

```r
rep(1:3, times=c(5,4,3))
```

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

```r
rep(z, times=1:5)
```

```
##  [1]  5 10 10  2  2  2  1  1  1  1 10 10 10 10 10
```

But the two vectors have to be the same length:

```r
rep(z, times=1:4)
```

```
## Error in rep(z, times = 1:4): invalid 'times' argument
```

Exercise

Now try to create a vector that repeats "nurbahar" 3 times, "ayla" 2 times, and "ozge" once (total length should be 6), which should look as follows:

```
## [1] "nurbahar" "nurbahar" "nurbahar" "ayla"     "ayla"     "ozge"
```

Solution

```r
toREPEAT = c(x, "ayla", "ozge")
toREPEAT
```

```
## [1] "nurbahar" "ayla"     "ozge"
```

```r
rep(toREPEAT, times=c(3,2,1))
```

```
## [1] "nurbahar" "nurbahar" "nurbahar" "ayla"     "ayla"     "ozge"
```

```r
x3 = rep(toREPEAT, times=3:1)
class(x3)
```

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

Another function in R that can be used to create numeric vectors is is called `seq` for sequence.

Exercise

Check the help page of `seq` and use it to create a vector that runs from 19 to 4, in steps of 3:

```
## [1] 19 16 13 10  7  4
```

Solution

```r
?seq
seq(from = 19, to = 4, by = -3)
```

```
## [1] 19 16 13 10  7  4
```

```r
# and this is an alternative solution:
seq(from = 19, to = 4, length.out = 6)
```

```
## [1] 19 16 13 10  7  4
```

```r
# or:
seq(from = 19, to = 4, length.out = ((19 - 4)/3)+1)
```

```
## [1] 19 16 13 10  7  4
```

Note that, as long as you specify which argument you write explicitly, you can change the order:

```r
seq(to = 4, length.out = 6, from = 19)  # this does the same thing
```

```
## [1] 19 16 13 10  7  4
```

```r
seq(4, 6, 19)  # but not this
```

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

Functions on vectors

Let's now create a vector named `f1` that ranges from 0 to 1 and separates the range into 12 steps:

```r
f1 = seq(0, 1, length.out = 5)
```

And add numbers 1 to 5 to `f1`:

```r
f1 + 1:5
```

```
## [1] 1.00 2.25 3.50 4.75 6.00
```

Add numbers 1 to 3 to `f1` - will it work?

```r
f1 + 1:3
```

```
## Warning in f1 + 1:3: longer object length is not a multiple of shorter
## object length
```

```
## [1] 1.00 2.25 3.50 1.75 3.00
```

When we execute this R runs the command without an error but gives a **warning**. This time, the lengths of the vectors are not multiples of each other. So R recycles the shorter one. It adds the first 3 elements, then adds 4th element of f1 to the first element of the second vector etc. recursively, as if:

```r
f1 + c(1:3, 1:2)
```

```
## [1] 1.00 2.25 3.50 1.75 3.00
```

Different functions can be applied on vectors:

```r
f1/3
```

```
## [1] 0.00000000 0.08333333 0.16666667 0.25000000 0.33333333
```

```r
f1^0.5
```

```
## [1] 0.0000000 0.5000000 0.7071068 0.8660254 1.0000000
```

```r
log(f1, base = 2)
```

```
## [1]       -Inf -2.0000000 -1.0000000 -0.4150375  0.0000000
```

```r
sin(f1)
```

```
## [1] 0.0000000 0.2474040 0.4794255 0.6816388 0.8414710
```

Subsetting vectors

Now we will learn how to obtain and manipulate subsets of vectors. In R subsetting involves square brackets, e.g. the 3rd element of `f1`:

```r
f1[3]
```

```
## [1] 0.5
```

Here 3 is the **index**. In R all indices are 1-based (as opposed to 0-based as in python and some other languages).

Try now obtaining the 2nd and 4th element:

```r
f1[2,4]
```

```
## Error in f1[2, 4]: incorrect number of dimensions
```

This will not work. The square brackets expect a vector as input (unless it is a single number):

```r
f1[c(2,4)]
```

```
## [1] 0.25 0.75
```

Now let's retrieve the 5th element 7 times, in different ways:

```r
f1[c(5,5,5,5,5,5,5)]
```

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

```r
# which is the same as:
rep(f1[5], 7)
```

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

Note that here R first retrieves the value of the 5th element, then uses this as input into `rep`.

You could even run this:

```r
el = 5
tim = 7
rep(f1[el], tim)
```

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

In R syntax using a negative index means the same vector without those elements:

```r
f1[-3]
```

```
## [1] 0.00 0.25 0.75 1.00
```

```r
f1[-c(2,4)]
```

```
## [1] 0.0 0.5 1.0
```

R's interesting behaviour:

```r
f1
```

```
## [1] 0.00 0.25 0.50 0.75 1.00
```

```r
f1[2.1]
```

```
## [1] 0.25
```

```r
f1[2.8]
```

```
## [1] 0.25
```

```r
f1[6]
```

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

```r
f1[6] = 100
f1
```

```
## [1]   0.00   0.25   0.50   0.75   1.00 100.00
```

Logical vectors

Another class of variables in R are logical (Boolean), which are `TRUE` or `FALSE` (`T` and `F`) in short:

```r
m = c(T, F, T)
m
```

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

```r
class(m)
```

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

```r
f2 = f1 > 0.3
f2
```

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

```r
class(f2)
```

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

A logical vector can be used to subset another vector where `TRUE`s will pass. E.g. to retrive the elements of `f1` that have values above `0.3`:

```r
f1[f2]
```

```
## [1]   0.50   0.75   1.00 100.00
```

The `which` function retrieves the indices of the `TRUE`s.

```r
which(f2)
```

```
## [1] 3 4 5 6
```

Meanwhile, `sum` and other mathematical functions can be applied on logical vectors. In this case, `TRUE`s are converted to `1` and `FALSE`s to `0`s, making `sum` and efficient way to count the number of `TRUE`s in a vector:

```r
sum(m)
```

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

```r
sum(f2)
```

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