Introduction to R

by Katja Nowick, University of Leipzig

Interactive R

The easiest way to use R is in its interactive mode. You start R in its interactive mode by simply typing "R" at the command line:

```
R
```

You will see how R starts and welcomes you. Each line begins with a ">". In this mode we can "talk" to R. We can ask it some questions, press "Enter" and R will answer us. For instance, we can use R as a calculator:

```
3+5
13-8
2*(7+2)
```

Or let's do something a little bit more complicated:

```
sqrt(16)
log(4)
sin(90)
```

Ok. That's easy, isn't it?
In the last three lines you used *functions*. Most of the time when you are programming in R you will use functions. You always recognize functions on their parentheses. Functions are operations that always work the same way. In the parentheses you write the arguments with which you want to run the function. In our example above, sqrt() is a function. You want to know the square root of 16, so you stick 16 into your function. Likewise, log() and sin() are functions. Here we calculated the logarithm of 4 and the sine of 90. We are going to use many more functions today.

Variables

A variable is a symbolic name to which a value may be associated. The associated value may be changed. You should always pick a meaningful name for your variables. Never use white space or special characters in a variable name. Never start a variable name with a number. Let's define the variable a:

```
a=2
```

If you forgot what a is, simply type "a" again, and R will tell you:

```
a
```

Note, there are two ways to assign a value to a variable. We just did it by using the "=" sign. You will also often see that people use "<-" instead.:

```
a<-3
a
```

Now you see that a equals 3 instead of 2, because we assigned the value 3 to a. So, what we just did, was overwriting the old value of a and assigning it a new value. Some people prefer to use "<-" to clearly indicate that they want to ASSIGN a value to a variable and not TEST if the variable is equal to a certain value. To check if a variable is equal to a certain value, we use "==":

```
a==3
a==2
```

R answers us with "TRUE" or "FALSE", respectively. "TRUE" and "FALSE" are special values that indicate that a test was passed (TRUE) or not (FALSE).
There are different types of variables. Our variable a is a "numeric" variable. You can test of which type a variable is by asking:

```
class(a)
```

Another type of variable is character. Let's define some character variables:

```
name1="Tina"
name2="Rizky"
```

Note how we always have to use quotes when assigning a character to a variable, otherwise we get an error message.
We can use our variables to do some calculations:

```
a*a
a^2
b=6
b/a
a<b
a==b
```

A list of all operators can be found here: http://cran.r-project.org/doc/manuals/R-lang.html#Operators
What happens if we type:

```
a=b
```

Check what the value of a is now:

```
a
```

What happens if we try to do calculations with our character variables:

```
name1*name2
```

Oops, we can of course not do calculations with characters. But we can ask for instance, if the two character variables are the same:

```
name1==name2
```

Ok. So far, we talked about two types of variables, numeric variables and character variables. Let's move on to some more complicated variable types: vectors and matrices.

Vectors

A vector is a variable that contains more than one element. These can be numbers or characters. Let's say we want to create a vector that contains the numbers from 1 to 9:

```
Numbers=1:9
Numbers
```

Alternative ways to achieve the same are to use the functions "seq" or combine "c":

```
Numbers=seq(1,9)
Numbers

Numbers=c(1,2,3,4,5,6,7,8,9)
Numbers
```

In this example, the first way was of course easier. But "c" is more flexible, and would allow us to assign the numbers in any order we want:

```
Numbers=c(3,6,2,10,2647849,1,999)
Numbers
```

Or to create vectors containing characters:

```
Names=c("Tina","Rizky")
Names
```

To get a certain element of your vector, you have to tell R which element you want. We use the square brackets to tell R the index of the element we want. Important: R always starts counting with 1:

```
Numbers[1]
Numbers[2]
```

```
Numbers[5]
Numbers[7]
Names[2]
```

We can also ask R for more than one element.:

```
Numbers[2:5]
Numbers[6:3]
Numbers[c(1,2,6)]
```

If you only want to get the first or last elements of your vector, use:

```
head(Numbers)
tail(Numbers)
```

This can be useful, if you have very long vectors and want to get an idea what your vector contains. In this case you don't want R to print the whole vector on your screen.
You may have wondered, if the different kinds of brackets mean something. We have already used "()" and "[]". The square brackets are always for *indexing*, for instance if we want to get certain elements from a vector. We will also use the square brackets in the next part to get certain elements from a matrix. The round parentheses are for *functions*. sqrt(), head(), tail(), c(), seq() are functions. R has many, many, many functions. You can even write your own functions - how, you will learn later. Within the parentheses your write the arguments, for instance, on which variable you want the function to be performed, how exactly do you want the function to be performed etc. You will see this, when we do more functions. There is a third type of brackets: {}. We are using these brackets to define *code blocks*, for instance in loops or if-else statements. We will do this later.

Another fun way to create vectors is the function "rep". Try this:

```
newNumbers=rep(1:13, 2)
newNumbers

newNumbers=rep(1:13, each=2)
newNumbers

Bambina=rep(c("a", "happy", "sunny", "funny", "bunny"),3)
```

What is the function rep doing? You have probably figured it out already based on the examples above. Let's assume you did not. You can find out what a function is doing by asking R for help. You do this by using the question mark:

```
?rep
```

If you are not sure what the exact name of a function is, try:

```
??rep
```

and R will give you everything that contains something with the term you were asking for. The so-called help pages are always organized the same way: Description, Usage, Arguments, ... and usually give examples in the end. The arguments specify how exactly the function will be performed. If you do not specify anything, the function will be performed with default arguments. The help pages explain you all the different options for the function. The first couple of times you open a help page it might look confusing. But no worries, you will soon get used to it. And you should, because this is the quickest and easiest way to get help! Let's go through the help page for rep together.

We can also change the values of the elements in our vectors. Let's go back to our vector Numbers and do some changes:

```
Numbers

Numbers[2]=24
Numbers

Numbers[5]=231
Numbers
```

We can add elements to our vector:

```
Numbers=c(Numbers, 602, 78,5)
Numbers

Numbers=c(-1,-392,52, Numbers)
Numbers
```

And we can delete elements from our vector. To do this we need to tell R the index of the element we want to delete. For instance, to delete the 3rd element of our vector:

```
Numbers=Numbers[-3]
```

We can also do the following:

```
Numbers=Numbers[2:11]
```

Here we overwrite our vector Numbers, saying that we only want the elements from 2 to 12 in the new vector also called Numbers. The outcome is similar to deleted elements.
Some functions that can come in handy if you have a long vector and quickly want to check some properties of the vector:

```
sum(Numbers)
min(Numbers)
max(Numbers)
mean(Numbers)
sort(Numbers)
```

You can check whether the values in your vector meet some condition. Let's say, you want to know if there is an element in your vector that has the value 3:

```
Numbers
Numbers==3
```

The answer from R is another vector with TRUEs and FALSE. Each element in the result vector corresponds to one element in the Numbers vector. Mostly the elements in the result vector are FALSE, but the 3rd element is TRUE. This means that the 3rd element in Numbers is equal to 3. Let's try some more tests:

```
Numbers==10
Numbers==6
Numbers<6
Numbers>=24
```

Is there any element in your vector that is less than 6, less than -400, greater than 1000?:

```
any(Numbers<6)
any(Numbers<(-400))
any(Numbers>1000)
```

Are all elements less than 10?:

```
all(Numbers<10)
```

Which elements are less than 10?:

```
which(Numbers<10)
```

The function which() gives you the indices of the elements meeting the criteria. What is the value of the elements that are less than 10? Here we need to use indexing to not get the number of the element but the value of the element:

```
Numbers[which(Numbers<10)]
```

Two alternative ways to get the elements that are less than 10 is this:

```
Numbers[Numbers<10]

subset(Numbers, Numbers<10)
```

What happens here internally is that R first creates a vector of TRUEs and FALSE by testing each element of Numbers if it is smaller than 10:

```
Numbers<10
```

This vector of TRUEs and FALSEs is then used for indexing and thus we can retrieve the elements less than 10:

```
Numbers[Numbers<10]
```

By now we already created a lot of variables. Do you still remember all of them? If not, ask R to list them for you:

```
ls()
```

If you want to remove some variables, you can use the function rm(). Often you don't need to remove variables, because you can just assign a new values to them. But some variables might be big and need a lot of memory, e.g. big tables. If you need more free memory, remove some large variables.:

```
rm(a)
```

Exercises I

1. Create a vector that contains the length of various alternative transcripts for a gene. The lengths are in bp: 526,723,1064,821,697,743,1149,489. What is the length of the smallest transcript? Are there any transcripts longer than 1000 bp? Are all transcripts at least 500 bp long? You discovered a new splice variant, that creates a transcript of 762 bp length. Add it to your vector. Sort all the transcripts by size. You sequenced the orthologous gene in the sister species. Most transcripts have the same size. Create a new vector that is a duplicate of the first vector but for the gene of the sister species. In the sister species you discovered that the transcript with 1149 bp contains a stop codon. Remove it from the vector.

2. You got a list of genes that are positively selected in your species of interest. Create a vector that contains these genes: BAL, CAM, LSD, FUZ, ZERP, DING, NOP, YIN. Sort the genes alphabetically and save the sorted gene list in a new vector. Have a look at the first and last couple of elements of the sorted gene list vector to see if they are indeed sorted. Which gene is the 3rd one in your sorted gene list? What is the index of DING in the unsorted gene list? Check if the gene "LSD" is in your gene list.

3. Optional: Your collaborator gives you two lists of genes and wants to know if they are among your positively selected genes: 1. NOP, PHO, SEN; 2. dns, hro, lamp, bal, cat, krr, zerp, yin, tir, jog, nop. Try out how "%in%" works. Can you change the small letters to capital letters? Check the help pages on how the function toupper() works.

Matrices and tables

A matrix is a rectangular array of values, similar to a table. You can easily create one from a vector. Let's first make a vector using some functions we learned before:

```
NumberVector=rep(seq(1,9),2)
NumberVector

NumberMatrix=matrix(NumberVector, nrow=6)
NumberMatrix
```

You see that you have to specify the number of rows you want. Alternatively, you could have specified the number of columns:

```
NumberMatrix=matrix(NumberVector, ncol=6)
NumberMatrix
```

You also see that in both cases R fills the matrix column-wise. If you want to change it, you need to set the argument byrow to TRUE (it is FALSE by default):

```
NumberMatrix=matrix(NumberVector, ncol=6, byrow=TRUE)
```

In case you want to check, if a variable you are working with is a vector or a matrix, you can ask:

```
is.vector(NumberVector)
is.vector(NumberMatrix)
is.matrix(NumberMatrix)
```

Most of the time you are going to deal with much bigger matrices. Often they will be tables that you got from somewhere. How do you get them into R? Here we need to learn a new function: read.table(). Have a look at the help page for read.table(). There are a lot of arguments you can set. R is sometimes good in guessing the format of the table you are trying to read. At other times it might mess up your table. Here we are going to read in a table that has column names (header), is tab-separated, has no quotation signs, and we want R to keep strings as strings (and not as factors):

```
Table1=read.table("Matrix1.txt", header=TRUE, sep="\t",
  quote="", stringsAsFactor=FALSE)
```

Always double-check that the table and the format of the table looks like what you expected. You can print the table to your screen:

```
Table1
```

But if it's a big table you don't want to do this. It's wise to check the size (dimensions) of the table first, before you print it:

```
dim(Table1)
```

This gives you the number of 1. rows and 2. columns of your table. You can also ask for the number of rows and columns separately:

```
nrow(Table1)
ncol(Table1)
```

Just like we have seen for vectors before, you can print the first and last part (first/last 6 rows) of the table:

```
head(Table1)
tail(Table1)
```

To get certain elements of your table, we need to use indexing. Remember the square brackets for indexing of vectors? For matrices and tables it is exactly the same, except that we have to give R two indices. R expects the row number to come first and the column number to be specified second:

```
Table1[1,3]
Table1[3,4]
Table1[1,2]
```

You can also get complete columns from your table. To do so, just leave the first index blank, e.g.:

```
Table1[,1]
```

This will give you the complete first column. To get a complete row, leave the second index blank, e.g.:

```
Table1[1,]
```

This will give you the complete first row.
Of course, you can also get other kinds of subsets from your table, e.g.:

```
Table1[1:3,2:3]
Table1[4:10,]
Table1[6,2:4]
Table1[,1:2]
```

Our table has column names (we specified this when reading the table by saying header=TRUE). You might find it easier to work with the column names instead of the index for the column. You need to tell R that you refer to the column name by using the $-symbol:

```
Table1$Expression1
Table1$Gene1[1]
Table1$Gene2[2:8]
```

A table can also have row names. Just specify this when reading the table. If your row names are in column 1, you would use:

```
Table1=read.table("Matrix1.txt", header=TRUE, sep="\t",
  quote="", row.names=1, stringsAsFactor=FALSE)
```

What are the dimensions of the table now?

Examples on how to use the row names:

```
rownames(Table1)
Table1["PTGDR",2]
Table1["PTGDR",]
Table1[c("PTGDR", "EOMES"),]
```

Let's get the old table back:

```
Table1=read.table("Matrix1.txt", header=TRUE, sep="\t",
  quote="", stringsAsFactor=FALSE)
```

Since every column and every row of a matrix/table is a vector, you can use all the functions we have learned for vectors, also for rows and columns of your table. Be careful to only refer to one column or row for doing this, e.g.:

```
max(Table1[,3])
mean(Table1[,4])
sort(Table1[,1])
any(Table1[,2]=="ZNF573")
```

And you could ask for all Expression1 values larger or smaller than a certain value. For instance, to get all Expression1 values (column 3) larger than 10, you would ask for:

```
Table1[,3]>10
```

This gives you a vector of TRUEs and FALSE, depending on if the Expression1 value is larger than 10 or not. Often you would like to see the actual values and not the TRUEs and FALSEs. You can achieve this by using:

```
Table1[Table1[,3]>10,3]
```

Note, that we use a condition to create a Boolean vector to select the rows we want: in this example we only want the rows in which the Expression1 value (column 3) is larger than 10. So we use Table1[,3]>10 to select the rows. And we want to see the Expression1 values, so we select column 3.

If we want to do the same, but see the complete row, we would use:

```
Table1[Table1[,3]>10,]
```

Let's suppose you don't want the Expression1 values larger than 10, but actually the names of the genes that have Expression1 values larger than 10. Just specify which column(s) you want:

```
Table1[Table1[,3]>10,1]
Table1[Table1[,3]>10,1:2]
```

The function subset() also works on tables and matrices. Just specify as the second argument the columns you want to select and your conditions:

```
subset(Table1, Expression1>10)
subset(Table1, Table1[,3]>10,3)
subset(Table1, Gene1 == "POLK")
subset(Table1, Expression1>10 & Gene1 == "POLK")
```

Of course, we can also change the values of elements in the table. Say which element you want to change and what the new value should be. Let's change some expression values:

```
head(Table1)

Table1[5,3]=1
head(Table1)

Table1[3,3:4]=c(10, 5)
head(Table1)
```

You can also add columns or rows to your table. To do so, you need to create a vector, which you can then add as a column using cbind() or row using rbind():

```
newRow=c("NEW1", "NEW2", 3, 5)
Table1=rbind(Table1, newRow)
```

Exercise II

Read in the table Matrix2.txt. How many rows and columns does the table have? What are the column names? Which genes (column 1) have expression values larger than 30 (column 3) Is the gene NSUN6 in the table? How often is NSUN6 in the table? Which Expression2 values belong to NSUN6? Make a second table that contains the first 6 rows of the first table. Make a vector containing the values 1, 2, 3, 4, 5, 6. Add this vector as another column to the newly created table.

Loops

The for-loop

Let's get back to our table. One thing you might want to do is to calculate the mean of Expression1 and Expression2 for each gene pair. I'm sure by now you can figure out yourself how to do this for an individual row:

```
head(Table1)

mean(c(Table1[1,3], Table1[1,4]))
```

But you don't want to do write the same code 35 times until you went through your whole table. That's when you use loops.
One kind of loop is the "for-Loop". With this loop you can do the same code FOR each element (here: gene pair):

```
for (GenePair in 1:35)
{
  print( mean(c(Table1[GenePair,3], Table1[GenePair,4])) )
}
```

Ok, lots of new things in here. Let's go through it step by step. First, all 4 lines together form the for-loop. It's important to consider them as a whole, to see what's happening. The basic structure of for-loops is always the same. The first line starts with the word "for". In the parentheses we say that we want to do the loop 35 times. More specifically, we define a new variable, called GenePair. GenePair will start with the value 1 in the first iteration of the loop. With each iteration, GenePair will increase by 1. Finally, when GenePair is 35, the loop will end.

If you had wanted to calculate the mean only of the GenePairs 5 to 10, you would have said:

```
for (GenePair in 5:10)
{
  print( mean(c(Table1[GenePair,3], Table1[GenePair,4])) )
}
```

After your for-statement, you open curly brackets. Within the curly brackets you write the code that you want to be executed in each iteration. Everything within the curly brackets is called a *"code block"*. The code in this code block is almost the same as the code we used before for one GenePair. Compare:

```
mean(c(Table1[1,3], Table1[1,4]))

mean(c(Table1[GenePair,3], Table1[GenePair,4]))
```

The only difference is that in the loop, each time we go through the loop we consider a different row. Remember, our variable GenePair goes from 1 to 35. So, in the first iteration we get the mean of the elements in row 1, in the second iteration of the elements in row 2 and so on. At the end of the loop, GenePair is 35, so the mean R just printed for you was for row 35.

Within the loop, we need to add the print() function. Otherwise, R would calculate the mean of each GenePair but not tell us what the mean is. You can try the same loop without the print(), in case you want to see what happens.

Finally, you certainly noticed the indentation of the code within the curly brackets. This makes your code more readable. It organizes blocks of code, so that it is easy to see what belongs together. You should make it a habit to use indentations. It will make your life and the life of your co-workers much easier if you or they ever try to read and understand any of your code. There are two common ways of indentation usage, which one you use is a matter of taste. Here are the two ways:

```
for (GenePair in 1:35)
{
  print( mean(c(Table1[GenePair,3], Table1[GenePair,4])) )
}
```

and:

```
for (GenePair in 1:35){
  print( mean(c(Table1[GenePair,3], Table1[GenePair,4])) )
      }
```

If you don't want the means to be printed to the screen, but rather want to store them in a variable to do something with the means later in your script, you need to declare this variable before you start the loop:

```
Result=numeric(35)

for (GenePair in 1:35)
{
  Result[GenePair] = mean(c(Table1[GenePair,3],
    Table1[GenePair,4]))
}
```

You can also do multiple lines of code within the loop:

```
meanResult=numeric(35)
maxResult=numeric(35)

for (GenePair in 1:35)
{
  print ("Hello")
  meanResult[GenePair] = mean(c(Table1[GenePair,3],
    Table1[GenePair,4]))
  maxResult[GenePair] = max(c(Table1[GenePair,3],
    Table1[GenePair,4]))
  print (maxResult[GenePair])
  print (GenePair)
}
```

The while-loop

Another very common loop is the while loop. It is useful, if you want a certain code only be executed while some condition is fulfilled (TRUE). For instance, you want to loop through your table for as long Expression1 of the gene pair is smaller than 1000:

```
row=1
while(Table1[row,3]<1000)
{
  print (Table1[row,3])
  print ("Is still smaller than 1000")
  row=row+1
}
```

Or you want to loop through your table until you get to a gene called EOMES (in other words: for as long as the gene is not called EOMES):

```
row=1
while(Table1[row,1] != "EOMES")
{
  print (Table1[row,1])
  print ("This is still not EOMES")
  row=row+1
}
```

Exercise III

1. Create a vector containing the numbers from 1 to 10. For each element of the vector you want to calculate the sine. Use a for-loop and create a vector with the results of the sine function for each element of the vector.

2. Let's imagine you have 10 bacteria cells. You put your bacteria into media and let them grow. Every time unit, each bacterium produces an amount of 6 metabolites and divides once. You let the bacteria grow until the amount of metabolites is more than 1000000. How many bacteria do you have when this happens? Hint: use a while loop in which you in each iteration double the number of bacteria and calculate the amount of metabolites.

3. Optional: In Exercise II you created a table of 6 rows to which you added a vector with the numbers 1 to 6 as additional column. Calculate for each row the sum of the numbers in the columns 3 to 5.

Apply

The function apply() and its derivatives are useful to apply the same function to each element of a vector or matrix. It's usually faster than looping through your vector or matrix.
For vectors use sapply(). It returns a vector:

```
x=1:10

sapply(x, sqrt)
```

For matrices use apply(). It returns a matrix:

```
m=matrix(x, nrow=2)
m

apply(m,1,sqrt)
apply(m,2,sqrt)
```

The second argument (called *margin*) of the function apply() specifies if the function should be executed by rows (margin = 1) or columns (margin = 2). This is important if calculating something across an entire row or column:

```
apply(m,1,sum)
apply(m,2,sum)
apply(m,1,mean)
apply(m,2, mean)
```

Exercise IV

1. Is there another way to do the same calculation as in Exercise III 1. without using a for-loop?

2. Use the table that contains Matrix2 (see Exercise II). Assign the values of column 3 to a new vector. Make one more vector in that each element is the element of column 3 minus 1 and combine the two vectors to a matrix. Log-Transform all elements in the matrix (use logarithm of base 2) using apply(). Calculate for each row in the matrix the sum of the two values.

Conditional Execution

Sometimes you want to execute a command only under some condition. Then you need an*if-statement*. Put your condition in brackets and the command in curly brackets. R will evaluate if the condition is TRUE, and only if it is true it will execute the command:

```
x=4

if(x==5) {x=x+1}

x
```

A better way to write it, especially if you want to execute more than one command within the curly brackets:

```
if(x==5)
{
  x=x+1
}
```

Check what the value of x is. The command was not executed, because x wasn't 5. Now set x=5 and try again:

```
x=5

if(x==5)
{
  x=x+1
}

x
```

The value of x is now 6. If you call the same if statement again, it won't be executed.

What if you want to do something under some condition, but otherwise something else? Then you need *else* to tell R what should be executed if the first condition is not TRUE:

```
x=4

if(x==5)
{
  x=x+1
}     else {x=x*2}

x
```

The same thing just written differently:

```
x=4

if(x==5) {x=x+1} else {x=x*2}

x
```

x should be 8 now.
Other examples for conditions:

```
if(x!=5) {x=x+1}

if(x>=5) {x=x+1}
```

Write your own function

We have learned about functions already. And R has many many functions available to you. But from time to time you might miss something and want to write your own function. This especially makes sense if you have some operation you want to execute more than once, potentially with different input data.

To write your own function you need four things: 1. Make up a name for your function. Often people use the prefix "my" to indicate that they made the function themselves. 2. Decide which arguments you want to give to your function. They will go into the brackets when you call the function. Usually these are your input data. 3. The actual commands/calculation your function is supposed to do. They go into the curly brackets. 4. Decide what you want your function to return. This is the result of the calculation done by your function.

Let's make a function that calculates the standard error:

```
myStError=function(x)
{
  StError=sd(x)/sqrt(length(x))
  return(StError)
}
```

So, we first gave our function a name (myStError) and said that it is a function. Our function takes one element, x, as input. In case of this function, this needs to be a vector for which we want the standard error to be calculated. Then we calculate the standard error within the curly brackets and store the result in the variable StError. And we define that the variable StError is to be returned.

The return is important, because we have no access to the variables that are calculated within the function. We only have access after the calculation to the variables returned by the function. If you forget to specify the return, R would still do the calculation specified by your function, but you would get no output from your function.

To run your function, you just need to call your function the same way you do it for other functions, i.e. the name of your function and the argument(s) in brackets:

```
x=1:10

myStError(x)
```

Or:

```
myStError(1:10)
```

Your functions can also take more than one argument.

Exercise V

1. An ant colony sends a certain number of foragers to find food. 70 percent of foragers come back with one food item. Write a function that calculates how many food items arrive at the nest. Hint: The number of foragers is your argument. Run your function for foragers = 50, 100, 300.
You study your ant colony and realize that the amount of food they bring home depends on the temperature. The 70 percent is true for a day with 25 degree Celsius. But with every degree lower temperature, the percentage of ants bringing food home drops by 2 percent. Change your function, so that it accepts temperature as an additional argument. And calculate the amount of food for 24, 20, and 8 degree Celsius.

2. What happens if your want to run your myStError function with the argument "Vladi" (myStError("Vladi")? How can you test within the function if the argument is of the right variable type? Add a command to the function that produces an error, in case the argument is not of the right variable type. Hint: check the help page for the function stop().

3. Optional: If you want to return more than one value you need to return a *list*. Change the function such that it returns mean, median, and standard error.

Graphics

R is also very powerful in terms of graphical data presentation. Talking about graphics can easily fill a complete course day. Here are just a few examples to give you an idea.

Let's plot Expression1 versus Expression2:

```
x=Table1[,3]
y=Table1[,4]
plot(x,y)
```

To add axis labels, use:

```
plot(x,y, xlab="Expression1", ylab="Expression2")
```

If you want red dots instead of black open circles, you can also specify this:

```
plot(x,y, xlab="Expression1", ylab="Expression2", type="p",
  pch=20, col="red")
```

There are many more ways to specify size, shape, color of dots, draw lines, label your dots etc. Start by checking out the help pages for plot() and then follow the links to get more information.

There are also many other types of plots that can easily be created with R. For instance a box plot:

```
boxplot(y)

boxplot(y, xlab="Gene 2", ylab="Expression level")
```

Some more graphics examples will follow in other modules.

You can also plot functions. In this case you have to specify the range in which you want to plot the function using the arguments *from* and *to*. Let's, as an example, plot a normal distribution and a sine function:

```
plot(dnorm, from=-5, to=5)

plot(sin, from = -2*pi, to = 2*pi)
```

For publications you sometimes want to have a figure consisting of multiple panels with plots. To create for instance a figure with 4 plots, you first need to set up your plotting parameters, e.g. specifying that you want them to be plotted in a 2x2 layout. You use the function par() for this:

```
par(mfrow = c(2,2), pty = "s")
```

The pty argument just specifies that the plotting should be squared and independent of the device size. Then you can draw your four plots into this area:

```
plot(x,y, xlab="Expression1", ylab="Expression2", type="p",
  pch=20, col="red")

boxplot(y, xlab="Gene 2", ylab="Expression level")

plot(sin, from = -2*pi, to = 2*pi)

plot(1:10, 1:10)
```

Very fancy stuff can be done with ggplot. Have a look at it when you have time: http://docs.ggplot2.org/current/

Exercise VI

1. Some people prefer to have slides with a black background. Then it would look nicer, if also your plot had a black background and if your axes and axis labels would be in white. Hint: First, to change the color of your graphics background use par(bg="black"). Then check the help pages for par() to see how you can change the color for your axes and axis labels.

2. Optional: Have a look at the help pages for plot() and par() and play with your graphics to modify your plots a bit more, e.g. by changing colors, fonts etc. You can also have a look at ggplot to get an idea of what else is possible with R:
http://docs.ggplot2.org/current/

Reading/writing files

We already used the read.table() function.

Just as you can read tables you can also write tables. Let's say, you want to save the Table1 with the meanResult and maxResult we produced before added as additional columns as a file. You can do this by using the function write.table().

```
Table1_mod=cbind(Table1, meanResult, maxResult)

write.table(Table1_mod, "modifiedTable.txt")
```

If you now open the file you just saved, it looks a bit ugly. The default usage of the function produced an extra first column and puts quotation marks around everything. We can change this. We might also like a tab-separated file instead. As usual, have a look at the help pages to see the options for writing files.

```
write.table(Table1_mod, "modifiedTable.txt", quote=FALSE,
  sep="\t", row.names=FALSE)
```

If we didn't have column names yet, we could have just added them here while saving the table:

```
write.table(Table1_mod, "modifiedTable.txt", quote=FALSE,
  sep="\t", row.names=FALSE, col.names=c("Gene1", "Gene2",
  "Expression1", "Expression2", "meanResult", "maxResult"))
```

What if you don't want to write the file in the directory you are currently working in? You have two options. 1. Simply provide the path to where you want your file to be located, e.g.:

```
write.table(Table1_mod, "R-Lecture/modifiedTable.txt",
  quote=FALSE, sep="\t", row.names=FALSE)

write.table(Table1_mod, "~/Evop/R-Lecture/modifiedTable.txt",
  quote=FALSE, sep="\t", row.names=FALSE)
```

Alternatively you can change the working directory within R. The function for this is setwd() (Set working directory). Then your file will be saved in the new working directory:

```
setwd("~/Evop/R-Lecture")
```

You forgot in which directory you are currently working in? Just ask R:

```
getwd()
```

Writing scripts

So far we used R in the interactive mode. This is good to quickly check something or to try out pieces of code. But once it works, you might want to just run everything automatically. You are almost there!

Let's make a script out of the lines of code we produced in our section on loops and the few additions we just did.

First, you need a text editor. Open a new document. Copy the following lines of code into the document:

```
Table1=read.table("Matrix1.txt", header=TRUE, sep="\t",
  quote="", stringsAsFactor=FALSE)

meanResult=numeric(35)
maxResult=numeric(35)

for (GenePair in 1:35)
{
    print ("Hello")
    meanResult[GenePair] = mean(c(Table1[GenePair,3],
      Table1[GenePair,4]))
    maxResult[GenePair] = max(c(Table1[GenePair,3],
      Table1[GenePair,4]))
    print (maxResult[GenePair])
    print (GenePair)
}

Table1_mod=cbind(Table1, meanResult, maxResult)

write.table(Table1_mod, "modifiedTable.txt", quote=FALSE,
  sep="\t", row.names=FALSE)
```

Save the file as my_first_Rscript.R.
You are now ready to run your script. From the Linux command line call:

```
R --vanilla < my_first_Rscript.R
```

Great. Well done! Have a look at the output file you created.

But now we would like to do the same mean and max calculation also with another input file, e.g. Matrix2.txt. Basically, you want to run the same script, but with an input file that has a different name. Also the outfile needs to get another name than you had before to not overwrite the previous file. So we need to change our script, so that it is more flexible and can run with any file you want.

To achieve this flexibility we cannot hard-code the input and output file name into the script (here when we used the functions read.table() and write.table()). And we need to start R from the command line by giving it the names of the input and output file as arguments. So, we need to make some changes: We add to the beginning of our script some lines that allow us to get arguments from the command line. It's a good thing to print the arguments, so that you see how many you have and in which order:

```
Args=commandArgs()
print (Args)
```

Then start your R script from the command line, giving it the names of your input and desired output files as arguments:

```
R --vanilla --args Matrix2.txt modifiedTable2.txt <
  my_first_Rscript.R
```

Have a look at the arguments R printed. It is actually a vector of arguments. The names for our input file and output file are the elements 4 and 5.

So we can add two more lines to our script right after we printed the arguments:

```
infile=Args[4]
outfile=Args[5]
```

When we call the function read.table() we can now use our variable infile to specify the file name. Likewise, when we call write.table(), we can now use the variable outfile instead of the file name. So, let's change these two lines in our script:

```
Table1=read.table(infile, header=TRUE, sep="\t", quote="",
  stringsAsFactor=FALSE)

write.table(Table1_mod, outfile, quote=FALSE, sep="\t",
  row.names=FALSE)
```

Ok, let's run the script again:

```
R --vanilla --args Matrix2.txt modifiedTable2.txt <
  my_first_Rscript.R
```

Oops, we are getting error message. Why is this? When trying to do cbind(), R tells us that the number of rows is different. Of course, our Matrix2.txt file has just 28 rows and instead of 35. But you don't always want to check first how many rows you have and then change your script to the right number of rows. That doesn't sound very efficient.

Remember how you can ask R for the number of rows your table has? Right:

```
rows=nrow(Table1)
```

And now we can change the 35 into our variable rows. Then, no matter how many rows our next input file has, it will always work.
There are three lines in our script we have to change:

```
meanResult=numeric(rows)
maxResult=numeric(rows)
for (GenePair in 1:rows)
```

Ok, let's see if it works:

```
R --vanilla --args Matrix2.txt modifiedTable2.txt <
  my_first_Rscript.R
```

And, indeed, we got a new output file with the right number of rows.

Documenting source code

An important aspect of writing a script is to document your script. This means that you explain what your script is supposed to do, what input files it expects, what output it produces etc. This is important if you ever want to give your script to somebody else.

But comments will also help you. When you look at your script again after half a year has passed by and after having written 10 other scripts, you will probably have forgotten what exactly each line in your script is doing and which arguments it expects in which order. In short, depending on how complicated your script is, it can be difficult to understand it again.

That's why: Always document your script!

You document by writing comments. A comment always starts with a #. This is the sign for R, that it can ignore everything that stands in this line. A good habit is to write a little text at the beginning of your script about what the general purpose of the script is. You may comment every line of code you write. However, with some experience, you will probably not comment every line anymore. But you should comment each function or *code block* that solves a particular task. As a rule of thumb, insert a comment every 10 or so lines of code. Also, whenever it took you some effort of thinking about a line of code or if you did something unusual, comment it!

Here is an example of how to document the script we just wrote:

```
# Purpose: Calculating the mean and maximum of expression
            values of two genes for each row/gene
# Input: Table with column names and 4 columns
#        1st and second column: the two gene names
#        Columns 3 and 4: expression values belonging to the
          first and second gene, respectively
# Output:  Table containing the same content as the input
            tables plus two additional comlumns
#          1st column: mean of the expression values of the
            two genes
#          2nd column: maximum of the expression values of the
            two genes
# Assumptions: If your script makes any assumptions about the
                input data, they should be documented here.#
# Arguments: 1 - Name of the input file;
#            2 - Name of the output file
```

```
# Getting the arguments from the command line
Args=commandArgs()
print (Args)

# The name of the infile is taken from argument 4
infile=Args[4]

# The name of the outfile is taken from argument 5
outfile=Args[5]

# Reading in the infile
Table1=read.table(infile, header=TRUE, sep="\t", quote="",
  stringsAsFactor=FALSE)

# Determining the number of rows (features) in infile
rows=nrow(Table1)

# This vector will store the result of the calculation of the
# mean expression values
meanResult=numeric(rows)

# This vector will store the result of the calculation of the
# maximum expression values
maxResult=numeric(rows)

# Performing the mean and maximum calculation for all rows
# (i.e. pairs of genes) in the infile
# The loop will also print "Hello" and the result of the
# calculations for each row
for (GenePair in 1:rows)
{
    print ("Hello")
    meanResult[GenePair] = mean(c(Table1[GenePair,3],
      Table1[GenePair,4]))
    maxResult[GenePair] = max(c(Table1[GenePair,3],
      Table1[GenePair,4]))
    print (maxResult[GenePair])
    print (GenePair)
}

# Combining the the results of the mean and maximum
# calculation with the input table
Table1_mod=cbind(Table1, meanResult, maxResult)
```

```
# Writing output file
write.table(Table1_mod, outfile, quote=FALSE, sep="\t",
  row.names=FALSE)
```

Congratulations, you just wrote your first R script!

Exercise VII

1. Change your first R script so that it prints the two genes of each row in alphabetical order to an output file. You need to write a loop again. In each iteration you sort the two genes of each row alphabetically. Can you store the result of each iteration in a vector? Save the result as a file. Document your source code.

2. Optional: Make an R script out of the function that calculates standard errors (see Exercise V 2.). You want to input your vector of numbers from the command line and get an output printed to the screen. Document your source code.

For further reading and reference have a look at the language definition of R, which can be found here: http://cran.r-project.org/doc/manuals/R-lang.html