Integrative Bioinformatic R course first lecture

Mohmaed Hamed & Sarah Fischer

12 Oktober 2018

R and R studio installation

Welcome to this R introduction course.

Before we start please download R and RStudio. For downloading Rstudio see https://www.rstudio.com/products/rstudio/download/ and for downloading R either use your shell script via pip or pip3 or visit the website of the R-project https://www.r-project.org/.

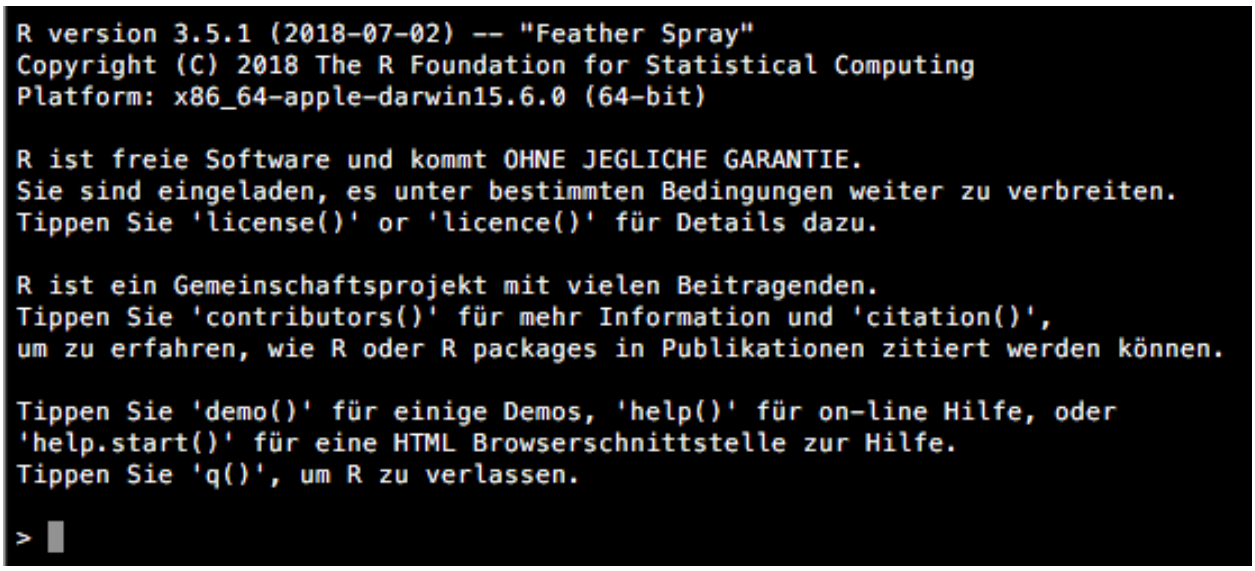
For downloading R you need to choose a CRAN mirror. This mirror will play a further role for downloading R packages. So take a look where your nearest or most trustworthy CRAN mirror is located.

R shell

First to the R programming language in generell. After installing R you can simply type R in your terminal / shell console.

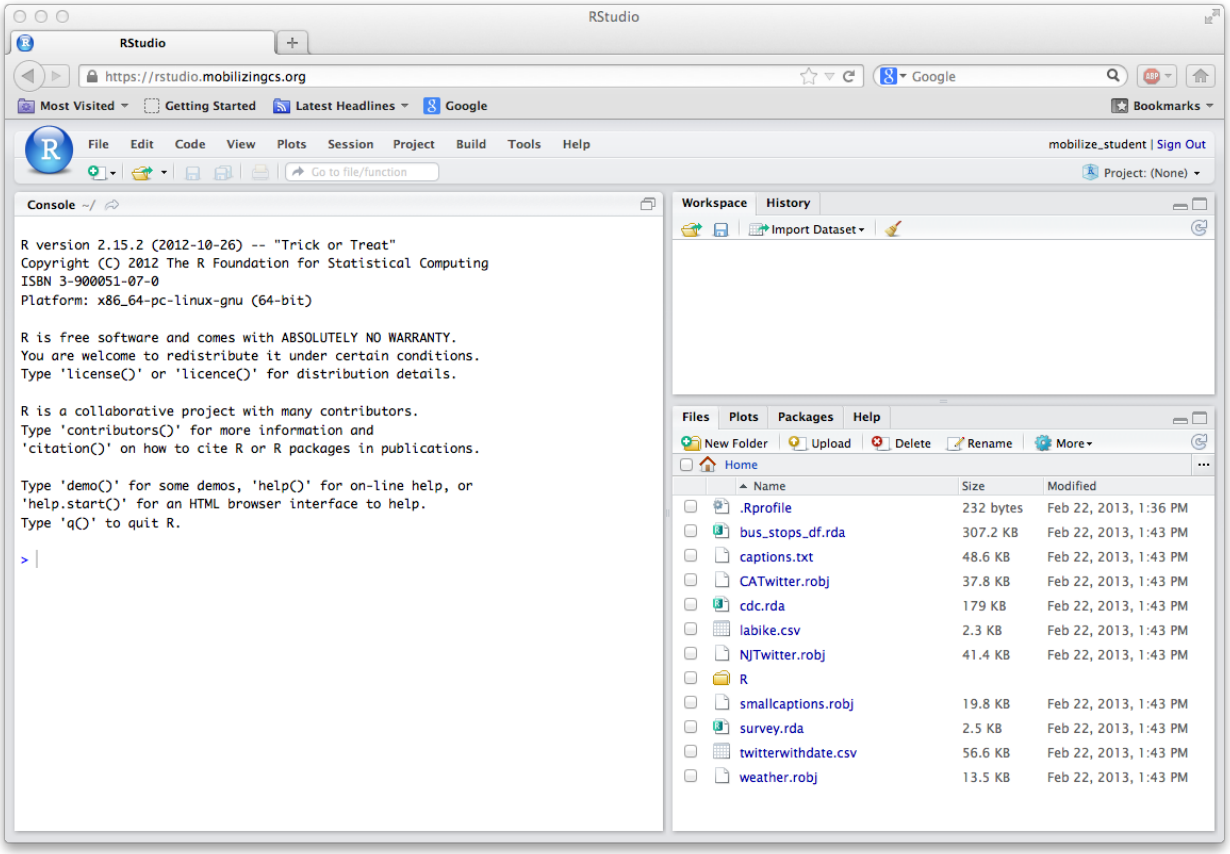
```
R
```

R Shell :



RStudio

The R IDE RStudio is available for all common platforms. Starting with RStudio is fairly easy by just typing your R commands in the Console. But only working directly interactive is not advised. Therefore you need *.R* files.



IDE:

Your Turn: Try out to type a R command directly in the R console.

```
"Hello World"
```

Create a script file

Create a new script file : go to *File-> New Script*.

An *Untitled* file will appear above the console tab.

Run code from script

To run the code in your script either :

- push the run bottom in the header line of the script to run everything

or

- mark the lines and make run line or selection

or

- press Control and R at the same time (MAC: cmd + Enter)

Save / Open a Script

To save the script go to *File -> Save* or *Save as* or use the shortkeys.

To open a saved script go to *File-> Open* or you can browse to it also under the tab "Files".

> Your Turn: Open the R script *Hello.R* located in the data folder of this workshop. Try out the different ways of running this code.

What is R ?

Besides being a free, wide-spread used programming language for statistical analysis R is powerful for bioinformatic analysis. The computational biology community provides a lot of biological data analysis related R packages which simplifies the usage of R for biological data analysis. Additionally R can be combined with other progamming languages or analysis tools. Besides the analysis R also produces graphics of high quality. Getting help from the R user community is easy, there are a lot tutorials, discussion forums and R user mailing lists online or even local meetup-groups.

Basic commands

One important command you need in every project is to find or set your working directory you want to use. If you started a new project then the working directory is the directory your project is saved. If you only started R studio then the working directory is the default setup. Change it to your desired location on your computer by replacing "pathname" by your path e.g. "/Desktop/Bioinformatics_course/example_data/").

The # symbol indicates comments in R.

```
# get the working directory path
getwd()

# set the working directory path
setwd("pathname")

# R is case- sensitive
Getwd

# But also gives you error messages
```

> Your Turn : Type the above comments with a parth to your desired location for your new R script. And set the workingdirectory to the course folder of today.

TAKE CARE : that the paths are specified with a single forward-slash separator.

TAKE CARE : R does not require a semicolon at the end of the statments.

> Your Turn : To look if this was successful then you should be able to run the *Hello.R* script from your new script with the following command :

```
source("data/Hello.R")
```

Getting Help

During this course the first thing to do if you need help is: Ask the presentor. But maybe also your colleagues to your right or left already managed the task. If you are sitting alone somehwere there are further possibilities to get help directly by R:

```
# General help page of R
help.start()

# For specific commands like print
```

```r
help(print)
?print

# Available for some R packages but not per default
browseVignettes("package_name")

# Try out tab-completion in RStudio (see that print is suggested)
pri
```

Your Turn : Try out the code from above by asking for help with the print function.Try also *??matrix* and look at the difference to *?matrix*.

R as a calculator

First of all R is a full feature programming language. Therefore we start with simple calculations with R. Let's try out this by writing the following calculations in our R script and run them afterwards.

```r
# An addition
5 + 4

# A subtraction
5 - 2

# A multiplication
3 * 7

 # A division
(5 + 4) / 2

# Modulo
28%%6

# Exponentiation
2^5
```

So R works with functions, like in mathematics. So the input (arguments or parameters) in this case are the numbers and the output or the result is the returned value.

Your Turn : Use R to calculate the following.

- Nine to the 3rd power divided by 2 ?
- What is (2/4)x6?

Build in mathematical functions

R provides already some basic mathematical functions for arithmetic calculations. Here is an overview over the most common functions.

Function	Operation
log(x)	Natural log of x
exp(x)	e^x
log(x,n)	log to base n of x
log10(x)	log to base 10 of x

Function	Operation
pi	π (3.141593)
sqrt(x)	Square root of x
factorial(x)	x!
choose(n,x)	Binomial coefficient ("n choose x")
floor(x)	Greatest integer $<$ x
ceiling(x)	Smallest integer $>$ x
round(x,digits=0)	Rounds x to nearest integer
cos(x)	Cosine of x in radioans
sin(x)	Sine of x in radians
tan(x)	Tangent of x in radians
abs(x)	Absolute value of x

Your Turn :

- What is log of 29 ?
- What is the square root of 257 ?
- What is the tangent of 7.25 ?

Variable assignment

The print function is called by writing print and in the paranthesis what has to be printed. So either direct what you want to print or a variable.

```
# print function
print("Hello world!")
```

A **variable** is a symbolic name to which a value may be associated. The associated value may be changed.

TAKE CARE :

- variable name cannot start with a digit
- some digits or names are already used and should be avoided :
  - c,q,t,C,D,F,I,T
- pi for ecample was already mentioned in the list of functions, even if it just a variable with a specific value.

Variable assignment could be with <- or with =

```
# Assign the value 42 to x
x <- 12

# Print out the value of the variable x
x
# This will only print it to the console

# use the print function if it should be stored e.g. in a log file
print(x)

# Modify the variable
x<-10*x +2
print(x)
```

```r
y="Character_variable"
print(y)

weight=55

# Add variables
x+weight

weigth_gain=weight + x
```

Your Turn : Define your own variable and execute some arithmetic calculations:

```r
# Assign the value 42 to your variable
.. <- 42

print(..)

log_transformed<-log(..)

x+..
```

Data types

There are different kind of data types besides numbers

- numerical : real numbers
- character: chain of characters, text
- logical : TRUE or FALSE : Boolean variabels
- factor : characters or numbers associated with a certain categorie
- special values :
  - NA- missing value
  - NULL - "empty object"
  - Inf / -Inf - infinity
  - NaN - not a number

What type is my variable ??

Sometimes there are data type related errors. Therefore you need to check what kind of data type your variable is. This is even more important when you are loading data from sources directly in R.

```r
galaxy_1<-42
galaxy_2<-"universe"
galaxy_3<-FALSE
```

Your Turn : Define the variables like here. Then try to run the logarithm function for all three galaxy variables. Check the class of the three variables by using the function *class(..)*

Detection of data types

There are some more function to check the data type more precisely.

```
v=1
is.character(v)
is.vector(v)

z=as.logical(v)
is.logical(v)
is.logical(z)
is.data.frame(v)

v="1"
is.numeric(v)
```

> Your Turn : Find for the three Galaxy variables what kind of check you have to use to get *TRUE* as result. Change the data type of galaxy_3 to numeric. What is the result? What happens when you try this also with galaxy_2?

TAKE CARE : changing the data types could lead to loss of information.

Vectors

A **vector** is an ordered group of elements of the same type with a single dimension. This is the elementary data structure of R.

Therefore the above check for *is.vector(v)* is *TRUE* , even if it is only one number.

Creating vector

The character "c" is used to concatenate the numbers into one vector. The vector input is separated by a comma. For character vectors quotations around the name is required.

Naming vector

Vector naming is the same as variable assignment. Use <- or = to assign the name to the vector

```
num_vector <- c(1, 10, 49, 60 ,99)
char_vector <- c("a", "b", "c" ,"d")
logic_vector <-c(TRUE,FALSE,TRUE,TRUE)


# sales per day from Monday to Friday
sales_store_A <- c(100, -50, 10, -20, 240)
sales_store_B <-  c(-24,-50,100,-350,10)

# Assigning days to the vectors
names(sales_store_A) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

# Try it a second time with days_vector
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(sales_store_A) <-   days_vector
```

Your Turn : Look at the *sales_store_A* vector before and after the assignment of the names Do the same for sales_store_B. Name the sales_store_B vector by the days_vector.

Addtionally you can generate a vector of regularly spaced numbers by using the sequence function or the replication function

```
seq(0, 1, length=11)

# or
1:10

rep(1:13,2)

coinflip=rep(0:1,15)

days=rep(1:7,4,each=2)
```

Your Turn : Take a look at the defined vectors. Make a vector reflecting the number of days in a month.Additionally make a vector reflecting the days of three months sorted ascending, without using a sorting method (e.g. 1,1,1,2,2,2,3,3,3,...., Consider 2 Months with 30 days and one with 31 days).

TAKE CARE: Often there are multiple solution to make some calulations and frames in R.

ANSWER:

```
month=seq(1,31,length=31)
month=c(1:31)
three_months=c(rep(1:30,each=3),31)
```

Calculation with vectors

Calculation could also be performed direclty on vectors, because most mathematical functions pre-implemented in R work element-wise.

```
height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
height^2

growth=height+0.05

difference =growth - height
```

Your Turn : Perform three pre-implemented functions on the height vector. Make another weight vector and calculate the bmi for it. Try out what happens when the vectors have different number of entries in either one, the weight or height vector. (Remember bmi= weight/(height^2))

ANSWER:

```
weigth= c(50,60,70,80,90,100,110,120,130)

height= c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
length(height)
length(weigth)

bmi <- weigth/height^2
```

Look at the warning message and then at the output of bmi

```
bmi
```

A particular useful function for this is the length function, returning the length of a vector, i.e. the number of elements it contains.

```
length(bmi)
length(height)
length(weigth)
```

Vector selection

To acess specific elements in a vector, use single square brackets *[]* with either only one number or multiple numbers.

```
height[3]

weigth[c(2,4,6)]

weight[c(-1,-4)]
```

> Your Turn : Get the index of the humans which have a weight smaller or equal to 1.80 from above by directly adressing them.

Converting between data types

There are some functions to convert types of vectors.Instead of only variables.

```
v=c(1,0,1)
as.character(v)

as.logical(v)

v=c("1","0","100")
as.numeric(v)
```

> Your Turn : Create a sequence of the first 20 triangular numbers. Triangular number is given by (n(n+1))/ 2.

ANSWER

```
seq(1,20)
sequence= seq(1,20)
triangular= (sequence*(sequence +1) ) /2
triangular
```

Logical values

For making comparisons or conditional statemente with smaller or bigger than, we also need corresponding operators. The output value of a comparison is a logical value either *TRUE* or *FALSE*.

- equal : ==
- not equal : !=
- greater than : >

- less than : <
- greater than or equal : >=
- less than or equal : <=

```
# Try out some comparisons between numbers and characters

1==1
1!=4
2>3
4<2.5
"ah">="b"
"a"<"b"
```

Additionally there are the logical operators

- AND : &
  - returns TRUE if both comparisons return TRUE
- OR : |
  - returns TRUE if at least one comparison return TRUE
- NOT: !
  - returns the negotiation (the opposite) of a logical vector

```
# AND
2>1 & 6>5

# OR
4==4| 3!=5

# NOT
!(6>3)
```

Conditional statements

Conditional statements are statments that are only performed if certain conditions are met.

You now them from other programmming languages already as if/else statments. In general first the statment in the if parentesis is evalutated. If it is *TRUE* the {} brackets are executed. If the statement is *FALSE*, either nothing happens or the else brackets are executed So the output variable of the statement has always to be a logical value. The general syntx looks like this:

```
if("statement1"){
  "execution"
}

if("statemet2") {
  "exectuion1"
} else {
  "execution2"
}
```

The **statement** could also be a logical or numeric vector , but only the first element is taken into consideration.

> Your Turn : Make a coinflip vector and let you printout head or tail dependent if the first flip was a head or a tail. Make it three time, once with two seperated if statments, once with if else and once without using logical operations in the statement(the last one is tricky).

ANSWER Length of the coinflip vector is not important also not if they make it with numbers or with characters

```
coinflip=rep(0:1,15)
coinflip_2=rep(c("head","tail"),5)

if(coinflip== 0){
  print("head")
}
if(coinflip== 1){
  print("tail")
}

if(coinflip_2=="head") {
  print("head")
} else {
  print("tail")
}

# They have to remember than 0 and 1 are also considered as logical values It appeared in the data conv

if(!coinflip[1]){
  print("head")
}
if(coinflip[1]){
  print("tail")
}
```

Loops

To perforem some conditional statments on a whole vector, we need to have loops.

A **loop** is a sequence of instructions that are continually repeated until a certain condition is reached. There are two types of loops:

- loops execute instructions for a precise number of times as controlled by a counter : **FOR**
- loops based on the verification of a logical condition. The condition is tested at the start or the end of the loop construct. **WHILE**

FOR loop

A for loop iterates over a vector.

```
for (val in sequence){
instructions
}
```

The sequence variable is a vector and val takes on each of its values during the loop The instructions are repeated and contained within curly brackets.

```
a<-c()
for (i in 1:10){
a[i] <- i
}
a
```

WHILE loop

The while loop is made of an epression that evaluates to a logical value, TRUE or FALSE.

```
while (test_expression){
instructions
}
```

As long as the test_expression is *TRUE* the instructions are executed and this execution stops once the test_expression evaluates to *FALSE*. This change of the test expression can take place in the loop.

```
cnt <- 2
while (cnt < 7) {
print("Hello, I am a while loop!")
cnt <- cnt + 1
}
```

> Your Turn : Without using max or min, create an R script that returns the maximum value out of the elements of a numeric vector x of any length.

ANSWER

```
numeric_vector<-c(3,4,5,9,10,32,21,3,1)
max_num<-0
for (i in numeric_vector){
  print(i)
  if(i>max_num){
   max_num<-i
  }
}
print(max_num)
```

Matrix

A **matrix** is a multidimensional collection of data entries of the same type. matrices have two dimensions It has rownames and colnames.

- A matrix with m rows and n columns has order mxn ; with dimensions(m,n).
- In R , matrices can be created in several ways : by the function **matrix**, by combining vectors or by tranfering it to a matrix.

Matrix construction

```
A <- matrix(nrow=2, ncol=3, data=c(1,2,3,4,5,6))
A

dim(A)

A<-rbind(c(1,3,5), c(2,4,8))
A


x<-c(5,7,8)
y<-c(6,3,4)
z<-cbind(x,y)
```

```
dim(z)
z
```

Your Turn : Execute the above matrix construction calls.

Naming matrices

```
rownames(A)<-c("x","y")
colnames(A)<-c("1st","2nd","3rd")

A
```

Matrix analyzation

Data access in matrices is similar to the access with vectors. The first number (x) represents the row, the second (y) the column matrix[x,y]. Additionally when the matrix has names you can acess the columns by name.

```
z[c(1,2),]

z[,-1]

z[2,]

z[2,2]

A[,"1st"]
```

Combining matrices

For combining data with matrices, you can use rbind or cbind depending if you want to add rows or columns.

Your Turn : Try out both rbind and cbind with the two matrices z and A. Look at the results if it works like you wanted it to be.

ANSWER Important they have different dimensions. And that is the trick by combining it.

```
new_matrix=cbind(A,z[1,])
new_matrix_2=rbind(A,z[,1])

combine_not_fit_together=cbind(A,z[,1])
combine_not_fit_together_2=rbind(A,z[1,])
```

Calculation with matrices

There are additional colculation specific for matrices. There are specific calculations for / with matrices, so for example the matrix multiplication.

```
matrix_C <- matrix(c(1,2,3,4,5,6,7,8,9),nrow=3)
matrix_D <- matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,byrow=T)
matrix_E <- matrix(c(1,2,3,4,5,6),nrow=2)
matrix_F <- matrix(c(1,2,3,4,5,6),nrow=3)
```

```r
# Matrix Multiplication %*%

matrix_C%*%matrix_D

# Inner product of a vector
b <- matrix(c(1,2,3),nrow=3)
bt <- matrix(c(1,2,3),nrow=1)

bt%*%b
b%*%bt


# Matrix transpose
t(matrix_E)
```

Additionally there are multiple other matrix facilities.

> Your Turn : Figure out how basic calculations (+,-,*,/) and logical operators on matrixes work with

- plain numbers
- vectors
- other matrices

ANSWER Want them to find out for themselves how the matrices are using it, that they have to keep track of the dimensions etc.

```r
number=5
vector= c(1,2,3)
matrix_E <- matrix(c(1,2,3,4,5,6),nrow=2)
matrix_F <- matrix(c(1,2,3,4,5,6),nrow=3)


matrix_E+number
matrix_E-number
matrix_E/number
matrix_E*number


matrix_E+vector
matrix_E-vector
matrix_E/vector
matrix_E*vector


matrix_F+vector
matrix_F-vector
matrix_F/vector
matrix_F*vector

matrix_F+matrix_E
matrix_F-matrix_E
matrix_F/matrix_E
matrix_F*matrix_E
```

```
######### logical

matrix_E<number
matrix_E>number
matrix_E==number
matrix_E!=number


matrix_E<vector
matrix_E>vector
matrix_E==vector
matrix_E!=vector


matrix_F<vector
matrix_F>vector
matrix_F==vector
matrix_F!=vector

matrix_F<matrix_E
matrix_F>matrix_E
matrix_F==matrix_E
matrix_F!=matrix_E
```

Factors

Factors are variables in R which take on a limited number of different values; such variables are often refered to as categorical variables. It is an important us in statistical modeling, to be treated differently than continous variables. Factors are stored as vector of integer values with a corresponding set of character values to use when the factor is displayed. With the levels command you can see the factor's level which are always character values.

```
data = c(1,2,2,3,1,2,3,3,1,2,3,3,1)
fdata = factor(data)
fdata


rdata = factor(data,labels=c("I","II","III"))
rdata
```

> Your Turn : Try out the levels command on the factors. Try out simple calculations with factor data.

There are further possibilities to use ordered factors.

```
fert = c(10,20,20,50,10,20,10,50,20)
fert = factor(fert,levels=c(10,20,50),ordered=TRUE)
fert
```

TAKE CARE : Factors do not work with even simple arithmetic operations and factors may be appearing unexpected as output of already implemented R functions.

Data frame

A **data frame** is essentially a matrix where the columns can have different data types.

A data frame contrains multiple rows and columns. Columns can be of different kind of data type (i.e. numeric, character, logic, etc.).The columns can be named and accessed by its name with the *$* sign

```
a<- c(4,8,16)
b<- c(3.5,1.2,2)
c<- c("M","F","M")

# Syntax of creating a data frame
exampledataframe<-data.frame(a,b,c)
names(exampledataframe)

names(exampledataframe)<-c("Age","Grades","Gender")

exampledataframe$Gender
```

Structure of data frames

You can acess the structure of the data frame with the corresponding data types of the entries by using the function **str**. The functions names, rownames and colnames could be used equivalently.

```
str(exampledataframe)
colnames(exampledataframe)
names(exampledataframe)
rownames(exampledataframe)
```

Data frames are a special kind of lists.

List

A **list** is a special type of vector . Each element can beof a different type.

Generating lists

Lists can be createy by using the list function or by coercing other objects using as.list().

```
example_list<-list(c(1,2,3),c("a","b","z","f"))
example_list

example2 <- as.list(c(1, "a character", TRUE, 1 + (0+4i)));
example2
```

```
L <- list(one = 1, two = c(1, 2), five = seq(1, 4, length = 5), list(string = "Hello World"))
L

names(L)
L$five +10
L[[3]]+10
L[["two"]]
```

Accessing the elements of a list

Elements of a list can be accessed by writing the name of the list and the $ character, followed by the name of the element or the double square bracket"[[]]".

```
example_list<-list(c(1,2,3),c("a","b","z","f"))
example_list[[1]]

example_list_2<-list(numbers=c(1,2,3), words="Testing")
example_list_2$words

is.numeric(example_list_2$numbers)
example_list_2[["words"]]
```

Additional vector functions

Additional important functions for vectors are the following three:

- *sum* calculates the sum of the elements of the vector
- *unique* returns a vector without duplicated elements
- *which* returns the location of the statement

```
sum(height)

unique(c(4,5,6,4,3,5,6) )

which(height> 1.8)
```

The which function

With the *which* function we are able to split vectors(and other data types) depending on a specific criterion. Additional there are combinations of simple statistical functions with which.

```
height [which(height> 1.8)]
which.min(height)
which.max(height)
```

> Your Turn : Check the variable *coinflip* from the vector section. Get the indices where the vector has the number 0 and isolate them as variable coinflip_number. Do the same for the ones with the number 1 and call the corresponding vector coinflip_head.

ANSWER

```
# Recap coinflip
coinflip=rep(0:1,15)

coinflip_head=which(coinflip>0)
coinflip[coinflip_head]

coinflip_number=which(coinflip==0)
coinflip[coinflip_number]

coinflip_number=coinflip[-coinflip_head]
```

Saving and loading Workspace

The current workspace can be saved and loaded again in the next session. This could be the whole workspace or single (multiple) objects. The loading function works the same for objects and for workspaces.

```
save(object, file="objectfile.RData")

save.image("myworspsace.RData")

load("myworspsace.RData")
```

> Your Turn : Save the current Workspace, close it and load it again. Select some of the vectors, matrices or dataframes you have created during this workshop and same them in a *.RData* object.

Importing data in R

There are different reading functions available in R. The most used one is *read.table*, but there are also *read.csv* , *read.delim* , etc. The default assumption is that the file is tab-or space-delimited file with a header row in the first line (important is the length, one instance shorter than subsequent lines). For specific formats the characteristics could be adapted.

```
importeddata<-read.table(file ="data/example.txt")

# adding to read the header
data_with_header<- read.table(file="data/example.txt",header=TRUE)
```

In this example there is no need to add the header argument because the data is already in the format that it will throw an error without the header. *TAKE CARE* Do not assume that this will always work out automatically right.

> Your Turn : How many read.xy functions are already implemented in the default R and look at the help pages to detect the difference between these functions.

TAKE CARE : By reading a dataframe, strings are automatically coded as factors if you do not specify it correctly.

Here are some examples of reading from an outside source. Look at the data types of patients.

```
# Reading directly from some source
patients <- read.csv("http://www-huber.embl.de/users/klaus/BasicR/Patients.csv")

# packages provide further reading options
install.packages("readr")
library("readr")

pat <- read_csv("http://www-huber.embl.de/users/klaus/BasicR/Patients.csv")

pat

str(patients)
str(pat)

pat[2, c(1, 2)]
pat[2, c("PatientId", "Height")]
```

Take care what functions you are using and what the output looks like especially if you have loaded packages. In this case the method took care of the conversion from factor to character.

Your Turn : Read in the example.xlsx file as well as the example.csv file. Detect the data type of the column Protein_c. Use for the .csv file *read.csv* and for excel install and load according to above the library *readxl* and run the command *read_excel*.

ANSWER

See the warning and also see that there could be something weired in the data. It is loading so that the user always has to check it.E.g. That for the csv. file there are factors in the Protein_c.

Just trying out different reading files.

```
csv_file=read.csv2("data/example.csv")
str(csv_file)

# Directly convert NA values to 0
csv_file_NA=read.csv2("data/example.csv",na.strings = 0)

# avoid factorising
csv_file_strings=read.csv2("data/example.csv",stringsAsFactors = FALSE)
str(csv_file_strings)


install.packages("readxl")
library("readxl")

# interactive possibility
my_data <- read_excel(file.choose())
#
# exl_file=read.delim(file="data/example.xlsx", sep="\t",header=TRUE)
# Gives warnings and not really an appropriate output

## for reading excel files the best practice is to use functions of already existing packages.


# Specify sheet by its name
genes <- read_excel("data/example.xlsx", sheet = 1)

# Specify sheet by its index
proteins <- read_excel("data/example.xlsx", sheet = 2)
```