Jessy Duran Ramirez,(jessyjomary.duranramirez@uzh.ch)

Exercise sheet: Hands-on introduction to R

1 Basics

1.1 This Intro

To execute an R command you can directly type it in the console (lower-left window in R-Studio) after the > sign. However it is good practice to save your code in a script and hence this is what you should do in this course. You can open a script with *File→ New File→ R Script* and save it with *File→ Save*. With CMD+Enter (Mac) or CTRL+r (Windows) you can execute the selected code in your script or, if there is no text selected, the line where the cursor currently is.

In order to follow the flow of this module, you have to type all the code that is in grey boxes like the following

```
# However you don't have to type lines that start with # these are comments ignored by R
# You should however read these lines since they contain comments essential
# for understanding the R commands following or preceding them
THIS WILL BE EXECUTED
```

in your R script and execute them as discussed above. In addition you should type and execute the code corresponding to the **TASKS**.

1.2 Datastructures: Vectors, Matrices, Dataframes

With the assignment operator "<-", we can assign values to a variable:

```
# This assigns the value 3 to variable v1
v1<-3
# For our purposes  v1=3 would do the same thing
```

You can display $v1$ in the console by just typing its name in the command line. Actually, $v1$ is considered by R to be a vector (of length 1). You can join different vectors or elements into a single vector by the function *c()* (for "concatenate")

```
# This concatenates v1 with 1 and 2 and assigns it to  v2 (display v2 to be sure it worked)
v2<-c(v1,1,2)
```

TASK: Using *c()*, create a vector "v3" containing the numbers 1 to 5 and verify that the comand *1:5* does the same.

We can combine different vectors of the same length to a matrix with the commands *cbind()* (considers them as columns) or *rbind()* (considers them as rows).

```
# This creates for example a matrix, whose first row is the vector v2 created before ..
# ...and the second row is the vector 1 to 3
# ...and the third row is the vector 2 to 4  (verify this)
ma1<-rbind(v2,1:3,2:4)
```

TASK: Create a matrix $ma2$ whose first column are the numbers 1 to 5 and whose second column are the numbers 4 to 8.

1.2.1 Selecting subsets from vectors and matrices

R is very powerful when it comes to selecting subsets of vectors and matrices. There are several ways to do this

Select via numerical indices For vectors $v[k]$ selects the k'th element of v, and $v[c(k,l,m)]$ selects the k'th, l'th, and m'th element. For example

```
# This selects the first two elements of v2 and assigns them to v4
v4<-v2[c(1,2)]
```

TASK: Select the first and the third element of v2.

Similarly for matrices, $ma[k,]$ selects the k'th row, $ma[,l]$ selects the l'th columns, and $ma[k,l]$ the element in row k and column l. This can again be extended to more than one row/column, e.g. $ma[c(k1,k2),c(l1,l2)]$ select the elements whose row is either k1 or k2 and whose column is either $l1$ or $l2$.

TASK: Select from $ma1$ the sub-matrix corresponding to the first two rows and the first two columns.

Select via names The elements of a vector can be named. The names can be assigned and viewed via the function *names*

```
#take an example
v5<-1:4
#This vector is not named yet
names(v5)
#Now assign names to its elements
names(v5)<-c("a","b","c","d")
#display the named vector
v5
```

The elements of the vector can now be simply selected by their names

```
#This selects the elements named "a" and "c"
v5[c("a","c")]
# but you can, of course still use the numerical-index-based selection described above,....
# ... in this case you can get the same output with..
v5[c(1,3)]
```

The same strategy applies with small changes to matrices: The names of the rows and columns of a matrix *ma* can be assigned with *rownames(ma)* and *colnames(ma)*, and $ma[rn1,cn1]$ select the elements with row-name "rn1" and column-name "cn1". For example

```
#asign names to ma1
rownames(ma1)<-c("patient1","patient2","patient3")
colnames(ma1)<-c("variable1","variable2","variable3")

#create subset containing only patients 1 and 3
ma3<-ma1[c("patient1","patient3"),]
#create subset containing only variables 1 and 2
ma4<-ma1[,c("variable1","variable2")]
```

TASK: Create a subset containing only variables 1 and 3 from patients 1 and 2.

Logical indexing Logical indexing is one of the most useful features of R. A logical variable can take two values: T standing for "TRUE" or F standing for "FALSE". If now v is a vector and vL is a vector of logical values of the same length as v, then $v[vL]$ selects only those elements at which vL has the value T. For example

```
v5<-c(-1,2,3,4,-2)
#This selects the first and the third element
v5[c(T,F,T,F,F)]


## [1] -1  3
```

This feature is so useful because we can obtain logical vectors as the result of other operations and then use them to select entries. For example

```
#This creates a logical vector which is true when v5 is smaller 0
v5<0


## [1]  TRUE FALSE FALSE FALSE  TRUE


#This can now be used to select only the negative elements of v5
v5[v5<0]


## [1] -1 -2


# We can for example set those elements to 0
v5[v5<0]<-0
v5


## [1] 0 2 3 4 0


# Similarly, "==" tests equality
v5==0


## [1]  TRUE FALSE FALSE FALSE  TRUE


# and "!=" inequality
v5!=0


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

TASK: Set in $v5$ all elements that are not 2 to 1.

With the same minor changes as seen above for names and numerical indices this concept does also apply to matrices; e.g. $ma[vL,]$ selects the rows at which vL has the value "TRUE".

TASK: Select from the matrix $ma1$ only those rows for which the value in the first column is larger than 1.

1.2.2 Data-frames and lists

A matrix requires that all its elements are of the same data-type. Consider the following example

```
#the vector "age"  gives the ages of the participants of a study ..
# .. (it is hence a numeric vector)
age<-c(20,25,27,18,45)
# the vector "initials", their initials (it is hence of type character)
initials<-c("D.T","H.C","B.S", "C.B","M.P")
#if we want to combine these two vectors to a matrix by cbind
cbind(initials,age)
```

```
##      initials age
## [1,] "D.T"    "20"
## [2,] "H.C"    "25"
## [3,] "B.S"    "27"
## [4,] "C.B"    "18"
## [5,] "M.P"    "45"
```

```
# we see that age is forced to become a character vector
#(as indicated by the quotation marks)
```

This problem is solved by the data-frame structure, which allows different columns to be of different format. In the above example we can create a data-frame by

```
participants<-data.frame(initials,age,stringsAsFactors=F)
#The option stringsAsFactors=F is chosen to avoid that
#the string variable "initials" is  converted into a factor
#(a data-type we will not consider here)
#This is how our data.frame looks like
participants
```

```
##   initials age
## 1      D.T  20
## 2      H.C  25
## 3      B.S  27
## 4      C.B  18
## 5      M.P  45
```

We can select subsets from data.frames in the same way as we have done for matrices (numerical indices, names, logical indices). For example

```
#select the column "age"
participants[,"age"]
```

```
## [1] 20 25 27 18 45
```

```
#select the column "initials"
participants[,"initials"]
```

```
## [1] "D.T" "H.C" "B.S" "C.B" "M.P"
```

```
#select all participants with age under 25
participants[participants[,"age"]<25, ]
```

```
##   initials age
## 1      D.T  20
## 4      C.B  18
```

In data-frames there is a second option to select columns via the "$" sign:

```
#this selects the column "age"
participants$age


## [1] 20 25 27 18 45
```

And a third option with the pipe,"%>%". Pipes are a powerful tool for clearly expressing a sequence of multiple operations. Packages in the *tidyverse* load "%>%"for you automatically, so you don't usually load *magrittr* explicitly.

```
# Load the package tidyverse
library(tidyverse)

# Select the column "age" from the data frame participants.
participants %>%
  select(age)
```

TASK: select all participants with age under 25 similar to above but using "$" and "%>%" to extract the column with age. *Hint: To subset rows with condition using dplyr, use the function filter instead of select.*

One of the most flexible data-structures in R are lists. In a list we can combine data-structures of any type– for example

```
# This generates a list whose first element is a data.frame and
# whose second element is a vector
list1<-list(participants=participants, v2=v2)
```

We can select elements from a list the following way (similar to the above)

```
# This selects the second element of the list
list1[[2]]


## [1] 3 1 2


# This selects the element with name "v2"
list1[["v2"]]


## [1] 3 1 2


# The same but using a dollar sign:
list1$v2


## [1] 3 1 2
```

Due to the flexibility of the list structure, many available R functions give their output in the form of lists as we will see below.

2 Datasets

We will use for the remainder of this exercise two data-sets

1. *chsiCourse.csv* (CHSI stands Community Health Status Indicators) contains data on the incidence of death by different causes (given as the number of deaths/100000 inhabitants per year) together with behavioural, socio-economic, and life-style data for all the US counties.
   Check *CHSI-Data_Sources_Definitions_And_Notes.pdf* for details about this data-set.
2. *esophCourse.csv* contains the data from a case-control study on esophageal cancer.

In order to read these two data-sets into *R*, we first change the work-directory (i.e. the directory where *R* looks for and writes files), to the directory that you use for this exercise. You can do this with

```
setwd(PATH OF THE DIRECTORY)
```

Then you can read the *.csv* file with the following command

```
chsi<-read.csv("chsiCourse.csv",stringsAsFactors=F)
#this assigns the data-frame to the variable chsi
#(note that this does not need to be the same name as your filename)
#the last option prevents strings to be converted into factors
```

TASK: Do the same with the *esophCourse.csv* dataset (assign this data to the variable *esophCourse*).

TASK: The data are read as a data-frame (i.e. the variable chsi is a data-frame). Check this with the function *is.data.frame*.

You can view data either by typing the name of the variable of the data-frame in the command-line (this is only an option for small data-sets) or with

```
### This shows the first lines of the data-set
head(chsi)
```

or with

```
#This shows the data as a spread sheet in a seprate window
View(chsi)
```

TASK: Discuss with your neighbour the content of the *chsiCourse.csv* and *esophCourse.csv* data-sets.

TASK: Generate a subset *chsiCalifornia* containing only counties from California (use the appropriate sub-setting approach introduced above).

3 Descriptive Statistics

R offers several useful tools for exploring and describing data.
For the distribution of individual variables:

- Histograms

```
#This makes a histogram of the Lung_Cancer incidence
hist(chsi$Lung_Cancer)
```

- Box-plots

```
#This generates a box plot for different variables
boxplot(chsi$Lung_Cancer,chsi$Col_Cancer,chsi$Brst_Cancer)
```

- Tables

```
#This shows how many counties there are from each state
table(chsi$CHSI_State_Name.x)
```

- The function *summary* provides a set of useful summary statistics for an individual variable or an entire data-set

```
#This summarises the chsi data set
summary(chsi)
```

Similarly for the relation between two variables:

- Scatter plots

```
# Plots Lung-Cancer incidence as a function of the percentage of smokers
# (one data-point-> one dot)
plot(chsi$Smoker,chsi$Lung_Cancer)
```

Now we take a look at the second data frame esophCourse. This frame contains three variables (age group, alcohol group, tobaco group) which might be related to the binary outcome of esophagal cancer.

- Cross-tabulation (or contingency table)

```
# This displays the number of data-points for each combination of ..
# ..the first and the second variable
table(esophCourse$esophBn,esophCourse$tobgp)
```

These absolute numbers are often not very intuitive. Showing frequencies is more intuitive. One very useful and flexible function is *CrossTable*:

```
#First we have to load the library where this function is from
# if you have not installed the package gmodels yet, you can do this with
#install.packages("gmodels")
library(gmodels)
#Now we can use Cross table the same way as table. Note the legend in the first rows.
CrossTable(esophCourse$esophBn,esophCourse$tobgp)
```

- Mosaicplots

```
#This is a graphical illustration of a contingency table / cross-tabulation
ta<-table(esophCourse$esophBn,esophCourse$tobgp)
mosaicplot(t(ta),col=TRUE)
```

Task: Discuss with your neighbour what each of these plots show/"tell" us.

Task: Using the help-function *?* (e.g. *?hist()* for the help for *histogram*), try to make these plots more beautiful (specify main title, label the axes etc).

4 Analysing the data

We start with the CHSI data and assess the association between lung-cancer and smoking:

```
# The command lm(y~x,data=dataX) fits a straight line to the variables y and x...
# ... from the data-set dataX. In our example:
mo<-lm(Lung_Cancer~Smoker,data=chsi)
# the fitted model is assigned to the variable mo
#  The function summary(mo) gives a summary of this  model
summary(mo)
##with the following commands we can add  the model fit to the scatter plot:
# plot(), produces the scatter plot :
plot(Lung_Cancer~Smoker,data=chsi)
#...and abline() adds the model-line to the scatter
abline(mo,col="blue")
```

Now we want to gain a more systematic understanding of this and to this end, we will consider for-loops and functions

4.1 functions

A Function in R has the following structure

```
#To specify a function you have to provide 4 types of items with the follwoing syntax
1_NAME_OF_THE_FUNCTION<-function(2_NAME_OF_THE_INPUT_VARIABLES){
  3_SEQUENCE OF R COMMANDS
  4_NAME_OF_THE_OUTPUT_VARIABLE
  }
#COMMENTS
#1_NAME_OF_THE_FUNCTION:  can be freely chosen
#2_NAME_OF_THE_INPUT_VARIABLES: in the case of more than one input variable,...
# ...these have to be separated by commas
# 3_SEQUENCE OF R COMMANDS: typically process the input and will be executed sequentially
# 4_NAME_OF_THE_OUTPUT_VARIABLE: the last variable will be returned as the output
```

Consider as an example

```
multiply<-function(x,y){
  product<-x*y
  return(product)
  }
multiply(3,4)


## [1] 12


# HERE
#1_NAME_OF_THE_FUNCTION= multiply
#2_NAME_OF_THE_INPUT_VARIABLES: x,y
# 3_SEQUENCE OF R COMMANDS: product<-x*y
# 4_NAME_OF_THE_OUTPUT_VARIABLE: product
```

TASK: Write a function "circleSurface" in which you enter the radius r of a circle and it returns its surface.

Functions are especially useful for tasks that have to be performed repeatedly with small variations. Take as an example the linear fit we have considered above. Assume that we want to extract the p-value for the effect of smoking on lung-cancer and the R-squared of the corresponding model (the latter tells us what fraction of the variance of Lung-cancer incidence is explained by smoking).These two values are actually hidden in the output of *summary(mo)*

```
# summary(mo) produces a list as output --- we assign this to the variable modSum
modSum<-summary(mo)
#with names(modSum) we see the names of these elements
names(modSum)
# the p-values are contained in the element "coefficients"
modSum$coefficients
#and correspond there to the column named "Pr(>|t|)"
modSum$coefficients[,"Pr(>|t|)"]
# we are typically not interested in the p-value for the intercept,
# hence remove this (remove first row)
modSum$coefficients[-1,"Pr(>|t|)"]
# note that a negative index removes the corresponding element/row/column:
# e.g. v[-1] returns the vector v without the first element;..
#...ma[-1,] returns the matrix ma without the first row, etc

# similarly the R-squared corresponds to the element "r.squared" and can be extracted with
modSum$r.squared
```

Suppose we want to repeatedly extract these summary statistics from many models (which we will actually do below). We can simplify this task by writing two functions that do this for us

```
#for the p values
getPvalues<-function(mo){ #this function takes a model as produced by lm() as an input
  #make the model summary
  modSum<-summary(mo)
  #extract the p values
  pvalues<-modSum$coefficients[-1,"Pr(>|t|)"]
  # return the p values as output
  return(pvalues)
  }
# for the r squared
getR2<-function(mo){ #this function takes a model as produced by lm() as an input
  #make the model summary
  modSum<-summary(mo)
  #extract the R-squared
  r2<-modSum$r.squared
  # return the  R-squared
  return(r2)
  }
```

We can now get these values from any model *mo* simply by

```
#for the p values
getPvalues(mo)
# for the r squared
getR2(mo)
```

As a next application of the concept of a function we will use it to obtain a better understanding of what a linear fit does.

In order to do this, we first have to define a function that produces a straight line (characterised by its intercept and slope) a measure of how well it fits the *Lung_Cancer* vs. *Smoking* data. This is done by the following function

```
getSSQ<-function(parm){
  # We assume that slope and intercept are entered into the function as a vector ..
  #.. of the form parm=c(intercept,slope)
  # extract slope and intercept from this vector
  intercept<-parm[1]
  slope<-parm[2]

  #calculate the lung-cancer incidence ...
  #...that the linear model with those parameters would predict
  yPredicted<-intercept+slope*chsi$Smoker

  #NOTE: as a side effect, we make this function plot the predicted lung-cancer incidence
  lines(chsi$Smoker,yPredicted,col=hsv(runif(1)),lwd=0.25)

  #as a measure of how well the predicted incidence corresponds to the real incidence...
  # ..we first calculate for each data-point the squared difference between predicted ..
  # ..and real lung-cancer incidence
  squaredDifference<-(yPredicted-chsi$Lung_Cancer)^2

  #and then take the sum over all data-points
  ssq<-sum(squaredDifference,na.rm=T)
  # (Use the help function to find out what the option na.rm in sum means.)

  #this is then returned as a measure of how well...
  # ...the straight line characterized by parm fits the data
  return(ssq)
  }
```

The measure produced by this function is called the sum of squares (SSQ) and given that this is a measure of how far away the prediction is from the observation, a smaller SSQ indicates a better prediction. Thus we try to find the linear line with the minimal SSQ. Let's start with the linear model obtained by *lm()*. The intercept and slope of the linear model specified above can be extracted as

```
summary(mo)$coefficients[,"Estimate"]
```

and we can estimate the SSQ by using the above function as

```
getSSQ(summary(mo)$coefficients[,"Estimate"])
```

Task: Test a couple of alternative values for intercept and slope and show that they give larger SSQs than estimate from lm().

The best fit of the data is thus provided by the straight line which minimises the SSQ. We can obtain this line by using the R function optim:

```
# This is just a preliminary step to plot again the original datapoints ..
#... (in order to visualize what optim does)
plot(Lung_Cancer~Smoker,data=chsi)

#optim searches for the parameters that minimise a given function, ..
#as a first argument the starting values for those parameters have to be provided..
#  ... and as a second argument the name of the function that should be minimized
```

```
optimalSSQ<-optim(c(0,0),getSSQ )

#The output produced by optim is a list containing (amongst others)...
# ... the optimal parameter values given by
optimalSSQ$par
# .. and the minimum of the function getSSQ reached at those values given by
optimalSSQ$value
```

Task: Verify that our manual approach with optim and *lm()* yield the same straight line as the best fit of the data.
Task: Fit instead of a straight line an alternative function to the data; e.g. a quadratic curve.

4.2 For-loops

A for-loop in R has the following structure

```
for(COUNTER in VECTOR){
  BODY
  }
```

which means that the COUNTER goes sequentially through each value in the VECTOR and every time the code in the BODY is executed. For example

```
for(k in 1:5){
  print(k)
  }
#this goes through the numbers 1 to 5 and prints each of them
```

Loops can also be nested inside each other

```
for(k in 1:5){
  for(kk in 1:5){
    print(c(k,kk))
    }
  }
```

We can also combine for-loops with if/else statements

```
v6<-c(-1,2,-2,4,5,6,-4)
v6positive<-c(); v6negative<-c()
for(k in v6){
  #if the statement inside the brackets () is TRUE ..
  # then the code inside the curly brackers {} is exectuted
  if(k>=0) {
    v6positive<-c(v6positive,k)
    }
  #if the condition of previous if statment is  false then the
  #the code after "else" is executed
  #note: an if-statment does not have to be follwoed by an else statement
  else{
      v6negative<-c(v6negative,k)
    }
  }
```

TASK: Recode/rewrite the previous for-loop using logical indices

Now we want to use these for-loops to systematically assess the association of potential exposures with the incidence of health-related outcomes in the CHSI data. To do this check first the column-names of the chsi data set

```
colnames(chsi)
```

you will see that columns 1 to 5 correspond to health-related outcomes and columns 6 to 9 to potential exposures. We will create a matrix which contains for each combination of outcome and exposure the R-squared, i.e. how much of the variance of the outcome (incidence) is explained by the exposure.

```
#First create the matrix filled with NAs
R2matrix<-matrix(NA,nrow=5,ncol=4)

#Appropriately name rows and column (Check whether this is the case)
rownames(R2matrix)<-colnames(chsi)[1:5]
colnames(R2matrix)<-colnames(chsi)[6:9]

##now calculate for each combination of exposure and outcome the R2
for(k in 1:5){
  for(j in 6:9){
    # note here the use of the function "paste": paste(a,b,c,d....) ...
    # ..joins the strings a,b,c,d into one single string
    # e.g. paste("I"," like","this course")="I like this course"
    mo<-lm(paste((colnames(chsi)[k]),"~",(colnames(chsi)[j])),data=chsi)
    R2matrix[k,j-5]<-getR2(mo)
    }
  }

#We can plot the matrix
heatmap(R2matrix,col = gray.colors(start=0.9, end=0.2,256),scale = "none",Rowv =NA,Colv=NA)
```

TASK: Do the same to produce a matrix with the p values for each association.*Hint: Use log p values for plotting to make the differences more visibile*

TASK: We have now seen several significant statistical associations between outcomes and exposures. How strong is the evidence for a causal role of those exposures? What are the caveats when interpreting the statistical associations as evidence for a causal role? What could be done to address these issues?

TASK: If you have come this far within one day, you are sufficiently advanced to think on your own about how to appropriately analyse the esophageal cancer. Discuss this with your partner and Teaching-Assistants and quantify the association between esophageal cancer and the potential exposures. (*Hint: Logistic regression*). Alternatively if you have had enough of statistics today, you can consider the next section.

5 Population Dynamics

As a final application of for-loops and functions we will consider simulating the dynamics of biological populations. Here we consider a classical population-dynamics model, the logistic map. Consider the population size of an animal species with discrete generations (e.g. a seasonal breader): at low population densities, the population size in generation $t+1$ will be a multiple of the population at generation t. However at high population densities, the population in generation t will use up so many resources that it jeopardizes its reproductive

sucess, thereby reducing the population growth or even the population size in generation $t+1$. Such a behavior is called negative-density dependent growth and can be captured by the updating rule

$$x(t+1) = r * x(t) * (1 - x(t)) \tag{1}$$

called the logistic map.

The following function simulates, starting from a specified initial population size, the dynamics of a population growing according to the above rule

```
#As input we provide the intial population density (xInitial)
# the growth Rate, the number of generations that the population is simulated..
#.. and optionally the length of the burn-in phase ..
# .. (i.e. the length of the initial phase which is discarded for the final ouput)
logisticMapDynamics<-function(xInitial,growthRate,nGenerations,nBurnIn){
  dynamics<-rep(NA,nGenerations)
  dynamics[1]<-xInitial
  for(k in 1:(nGenerations-1)){
    dynamics[k+1]<-growthRate*dynamics[k]*(1-dynamics[k])
    }
  dynamics[nBurnIn:nGenerations]
  }

#here we plot  the dynamics for one choice of growth rate
plot(logisticMapDynamics(0.01,1.5,100,10) ,type="l",ylim=c(0,1))
#& here we add the dynamics of another growth rate
lines(logisticMapDynamics(0.01,2.5,100,10),col=3)
```

TASK: Discuss with your partner what the function *logisticMapDynamics* does?

TASK: Plot the dynamics for a range of growth-rates between 1 and 4 and discuss the different qualitative behaviors of the system?

To obtain an overview of the dynamic behavior of the system over an entire range of parameters, we can use the following approach

```
res<-c()
for(r in seq(1.3,3.9,by=0.005)){
  res<-cbind(res,rbind(r, logisticMapDynamics(0.01,r,500,100)))
  }
plot(res[1,],res[2,],pch=".")
```

TASK: Discuss with your partner what the above for-loop does and interpret the plot that is generated (this type of plot is called a bifurcation diagram. why?). Discuss what this pattern means for our understanding of complexity in biology.

An alternative to the logistic map (that follows a similar biological intuition of negative density dependent growth but has some mathematical advantages) is given by

$$x(t+1) = x(t) * exp(r * (1 - x(t))) \tag{2}$$

TASK: Repeat the above analysis (exploring dynamics, bifurcation diagram) for this version of negative density dependent growth.