Cognitive Modeling (INFOMCM)
A very quick background on R

Author: Chris Janssen (c.p.janssen@uu.nl)

General information

This assignment is meant to give you a short non-exhaustive introduction to R. This assignment does not need to be handed in, and it will not be graded. However, you can ask your lecturers for feedback during the lab sessions. Do not spend more time than 1 class (4 hours) on this – dedicate time outside of class to learning R if you need more time. If you need a lot more time, please contact the lecturers – your ambitions might have been set very high.

You are highly encouraged to also use many of the free available tools on the web to gain more information about R and its use. Please note that you might also learn more about R as you are conducting the main assignments (see Blackboard).

In this assigment I assume that you have some basic background in programming (e.g., taken a course before, or have some experience outside of classes). If you have no programming experience, please contact the lecturer.

Lecturers

Chris Janssen (lab coordinator)
Krista Overvliet
Teaching Assistants

Goals of this lab session

At the end of this lab session, you can:

1. Run code in R and/or software packages that run R (e.g., R-studio)
2. Find more information about R and its functions in various ways
3. Implement your own code in R

The level that we want to achieve is that you are familiar enough with R such that you can work on the main assignments (and through those, learn more about R).

Symbols

Throughout this assignment, three symbols will be used, similar to the main assignment.

<ins>Self-check, hints, tips:</ins> At a self-check, you will be asked a question to check whether you understand what you read and did so far. You can discuss this question with

your partner. For hints and tips, you will get some more information that might help you in your understanding.

Testing: Here we describe concrete tests that you can run to test if your intuition is correct.

Engineering: This is an essential step that you need to do, and which involves some coding.

Section I: Where to find R and information about it?

Engineering: There are multiple alternatives. Read the full information on installing R first before installing it.

- The default R installation is to go to the R website and download the latest version of R: https://www.r-project.org/
  In the menu on the left side, look under "CRAN" and choose a server in the Netherlands. While your downloads run, read the hints and tips below.
- If you use a Windows machine, you might also want to install Rstudio: https://www.rstudio.com/ . This software package highlights various components in your code (e.g., functions and specific statements such as if / else). As a downside, the graphical environment reduces the speed with which your code executes.
- If you use a Mac, the default R package already highlights your code nicely and you don't need Rstudio (some Linux distributions also do this).
- **Multi-threading in R:** In some of the labs (e.g., the lab on processing models), you will run quite intense simulations that will take quite some computing time. One of the reasons is that, at the time of writing this manual, the default R packages do not use multi thread computations. The Microsoft Open R package of R does allow multi thread computing and can therefore be faster. It is compatible with multiple platforms and with R studio. However, it might not be compatible with all R packages. You can find Open R here: https://mran.microsoft.com/open/

Hints: R is used widely in machine learning, statistics, linguistics, and psychology. Therefore, there are lots of (free, online) resources. Here are my top favorites:

- **Online fora**: simply google "R" and then what you look for
- **R short reference card**: install this and place it prominently on your desktop for easy access. It defines some of the most basic and frequently used functions: *ftp://cran.r-project.org/pub/R/doc/contrib/Short-refcard.pdf*
- **Free R manuals**: There are many good manuals on the R website. https://cran.r-project.org/manuals.html I recommend the "An introduction to R website", as it provides a very structured overview of topics (e.g., vectors, lists). Other manuals (including dedicated topics and short manuals) are also available here: https://cran.r-project.org/other-docs.html
- **R help functions:** R helps you in three ways:

  - Within the R environment is a “help” menu (see top menu) in which you can find general information about a topic
  - When you type *help(nameOfFunction)* in the command screen you will open the help file of that function. For example, check *help(par)* or *help(points)* (NB: this only works for predefined functions)
  - When you type *?topic* in the command prompt, you will find an overview of documents that relate to this topic. For example *?plot*
- **Andy Field book:** If you want to learn about statistics as used in social science a good resource is the book “Discovering Statistics Using R” by Andy Field and colleagues. It is lengthy, but explains many topics in great detail.

Section II: The interactive command prompt

<u>Testing</u>: Start R. You will see a “console window” or command prompt. In this prompt, you can interactively type commands behind the “>” sign. When you press enter, that command is executed.

Try out statements like the ones below and any additional statement to see how the interactive prompt (and R) works. Make note of when values are stored internally, and when they are not.

```
> 3+ 3      ##### execute some mathematical statement

> “hello”    #### type a string (i.e. set of characters)  (Note: the “ symbol needs to be typed manually and not copied from this assignment in order to work)

> a <- 5      ##### assign (<-) a variable (here: a) with a value (here: 5)

> a      ### show the current value of a variable

> a + 6      ##### show the value of a variable when an operation is done on it

> a      ##### note: that this does not update the variable value

> a <- a + 6      ##### unless you update it explicitly

> myvector <- c(4,5,6)      ##### assign a vector of numbers to a variable

> myvector * 2

> 3 == 4    ### a Boolean (True/False) statement

> 4 <= 3   ### another one

> help(plot)    ### open a help menu
```

Engineering:
By developing a script, you don't need to retype every command every time you want to run it. In Rstudio a "script" window is probably already open from the moment you started the program. In regular R you can use the menu to open a "new document" (or to load an existing document/script). Open a script, and save your file as "myBasicRtest.R".
(continues on next page)

Now write some code in your script. For example, you can write some of the commands that you tested before. Note that in a script file, you do not need to start with the ">" sign.
You can also try out some of the functions that are mentioned in the *R short reference card*, to familiarize yourself with some of the core R functions.

Once you've typed in commands, you can execute your code:

- Select the part of the code that you want to run (or, *select all* code).
- Type ctrl + enter (windows) or apple+enter (mac) to execute the selected code.

Try this yourself.

Section III: Statements, Functions, Packages

In R you can make use of many well-known programming statements such as if, else, while, and for.

Testing: Type the following code in your script editor and test what it does:

```
check <- TRUE
while(check)
{
      for (i in unique(seq(1,20,1)))
      {
            if(i < 12)
            {
                  if (i %% 2 == 0)
                  {
                        print(c(i, i+1))
                  }
            }
            else
            {
                  check <- FALSE
            }
      }
}
```

Do you understand what this code does? Do you understand the if, else, while, and for statements? If you do not understand a step, try to only execute part of the code first (e.g. “seq(1,20,1)”)
Notice my use of two types of brackets: ( ) and { }
Notice how variables get assigned. Notice use of the symbols == , %%, and <

Testing: Now let’s try functions.

You can use functions that are already predefined. For example, like this:

```
plot(seq(1,15,1),seq(1,15,1),col=rep(2,length(seq(1,15,1))),pch=seq(1,15,1))
```

The predefined functions are plot, seq, rep, and length.
(use the help() command to find out more about these functions)

You can also define them yourself, as is done here for *mysteryFunction* and for the *slicerFunction*:

```
mysteryFunction <- function(digit,add)
{
        result <- digit + add
        result
}
slicerFunction <- function(digit,criterion=5)
{
        result <- digit[digit<criterion]
        result
}

test <- c(4,5,6,7,8,7,6,5,4,3,2,1)
slicerFunction(test)
slicerFunction(test,8)

mysteryFunction(test,5)
```

What do these functions do?
Why does only one of the following two commands work?

```
> slicerFunction(test)

> mysteryFunction(test)
```

Engineering:
Now let's try defining your own functions. Define a function "myCounter", which has exactly 1 argument, called "number". "number" can be given the value of 1 number or a vector of numbers. The outcome is either 1 number (if just one value is given), or a vector of numbers in which you give the addition of the number plus its position in the string of digits. Here is some example output:

```
> myCounter(4)
[1] 5
> myCounter(number=4)
[1] 5
> myCounter(c(4,10))
[1]  5 12
> myCounter(1:10)
 [1]  2  4  6  8 10 12 14 16 18 20
> myCounter(10:1)
 [1] 11 11 11 11 11 11 11 11 11 11
```

Tips: R can also load "packages". Very useful!! Packages contain functions that other people have predefined. In (regular) R you can install packages by going to "package installer" in the "Packages & Data" menu at the top. You can search for a package in a specific CRAN (pick one in the Netherlands). Make sure to always select "install dependencies". The procedure might be different in Rstudio.

Install for example "ggplot2", a useful graphical package. Once you installed it, you need to call it in your code. For example, test this:

```
require(ggplot2)
qplot(c(4.3,5.2,6.9),c(10.1,11.2,11.9),xlab="x label", ylab="y label")
```

Tips: When you plot data using R, it plots "layer by layer". The order of commands influences what gets printed in what order. For example, run the following commands one by one to see the order of plotting:

```
plot(seq(0,10,1),seq(0,10,1),pch=20,col=1)

lines(c(0,10),c(10,0),col=3)

points(rnorm(15,5,0.5),rnorm(15,4,0.5),col=2,pch=20)

text(5,9, "test")

plot(5,6)
```

Notice how a new plot command overwrites your previous plots. In order to plot multiple plots next to each other, you want to explicitly call a new plotting window using:

```
quartz()    ### on mac
window()  ### on PC
```

If this is followed by this command, you can specify how many rows and columns you have:

```
par(mfrow=c(2,3))
```

The package ggplot uses a slightly different way of ordering data across various plots.

Section IV: Vectors, data frames, data summaries

In R you define a vector using the c() function. However, some other functions can also give a vector as output.

Testing: Try out these commands and make sure you understand what they do.

```
c(4, 5, 6)    ### simple vector

c()    ### empty vector

a <- rep(5,10)

b <- seq(0,9,1)

c <- c(4, 5, 6)

a * b

a + c

a + 1
```

Engineering:
A useful format in R is a so-called "data frame". Develop the following data frame, called "myFrame" in your script window:

```
a <- 0:9
b <- seq(0,0.9,0.1)
ind <- rep(c("A","B"),length(a)/2)
indAlt <- c(rep("C",length(a)/2),rep("D",length(a)/2))
myFrame <- data.frame(index1=ind,index2=indAlt,a,b)
```

Testing:
Typically, you only want to look at a so-called "subset" of your data: either specific rows or specific columns. Try out these commands and see whether you understand what they do:

```
names(myFrame)
myFrame$index1
myFrame$index2          ### what does the $ sign mean?
dim(myFrame)

myFrame[1:5,]
myFrame[,1:2]           ### what does the "," sign do?

myFrame[myFrame$index1 == "A",]
myFrame[myFrame$index1 == "A",]$b
```

Make sure you understand how the "," and the "$" are used to look at specific parts of the data frame.

You can also add new columns to R yourself. For example:

```
myFrame$c <- "test"
myFrame$d <- myFrame$a %%2
myFrame$diffa <- c(0, diff(myFrame$a))
myFrame$diffb <- c(0, diff(myFrame$b))
myFrame$diffd <- c(0, diff(myFrame$d))
```

What does the diff function do?

Tips: You can save a data frame to your computer as either a ".Rdata" file, or as for example a comma separated value (CSV) file. This is for example useful when you want to store data that is generated by a model. If you want to find out more, read about the following functions (e.g., in R short reference file and the R help files):

- save()
- load()
- read.csv()
- setwd() a function to load a specific directory as your local working directory

<u>Testing:</u>
When you have a data frame, you might want to get specific information out of it. Test the following ways of extracting data from data frames:

```
mean(myFrame$a)

summary(myFrame$a)

summary(myFrame)

levels(myFrame$index1)
```

Sometimes we also want to get specific summaries for specific <u>*subsets*</u> of the data. The functions *tapply* and *aggregate* are useful. These functions take multiple arguments and if you use a data frame, you typically need to define multiple times what part of the data you look at. In this respect the command *with* is useful. Once you've defined a (subset of your) data frame using *with*, you can later use just the column names instead of the full definition using the (subset of the) data frame.

Here are some examples of commands:

```
with(myFrame,tapply(a,list(index1,index2),median))

with(myFrame,aggregate(a,list(l1=index1,l2=index2),median))


with(myFrame[myFrame$index2 ==
"C",],aggregate(a,list(l1=index1,l2=index2),median))
```

Notice how the use of "list" is slightly different in aggregate compared to tapply. Notice that aggregate gives a data frame as output. Notice also what the column headers are. Generating a data frame as output is very useful when you want to do some further calculations yourself!
If you want to understand why "*with*" is such a useful function, try to rewrite the function without the use of "*with"*.

Section V: Simple statistics

R is developed for number crunching and statistics. For example, let's imagine that we have gathered data on reaction times for two groups. For one group the data follows the following distribution:

```
fast <- rnorm(20,200,10)
```

for the other group the data follows the following distribution:

```
slow <- rnorm(20,400,10)
```

You can perform a t-test to see if these two groups differ. This is the simplest way (if the test is unpaired and we do a two-sided test):

```
t.test(a,b)
```

<u>Testing:</u>

As you might recall from your statistics classes, whether you can detect a difference between two groups depends on many factors, including: how many observations you have, how far the means are apart, and what the distribution of values is (is it a narrow or a wide distribution). You can try this with the above code by systematically changing the parameters *n*, *mean*, and *sd* of the rnorm function. See whether this has the expected effect on the ability to detect a difference between groups.

Of course there are many more ways in which you can do statistics. A good resource to check is the Andy Field book "Discovering statistics using R". On the last page of this book an overview is given of what type of test might be appropriate, given the type of data you have collected.