Cognitive Modeling LC1

Harm Manders

```
## set working direcotry
setwd("/home/harm/Uni/cogmod/LC1/") ##<======= SET THIS TO YOUR LC1 FOLDER!
```

```
Put {r, echo=FALSE} in the header of a code block to exclude the commands and only print output.

Note that for text outside R blocks a double SPACE at the end of a line means NEWLINE, whereas in the co
```

For more one Rmarkdown http://rmarkdown.rstudio.com/authoring_basics.html
and see the cheatsheet

1. basic numerical opertions

```
1+1
2*3
3-4
1e+2
1e-2
2^2
sqrt(4)
log(10)
```

if you want to know more about a function you can type

```
?function_name or help(function_name)
```

in the console. Further info should then appaer on the right in the Help tab.

Q1 Try *?log* to read how to use log function. What are the two ways to calculate the log of 10 using base 10?

(please list all answwers are pasted at the end of the document)

```
log(10, 10)
log10(10)
```

2. Variable assignment

We assign values to variables with the assignment operator "=". Just typing the variable by itself at the prompt will print out the value. We should note that another form of assignment operator "<-" is also in use.

```
x = 10 # by the way this is a commment (using a #)
y <- 11
x
```

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

```
y
```

```
## [1] 11
```

```
x*y
```

```
## [1] 110
```

Note there is also an operator <<- This is normally only used in functions, and cause a search to made through parent environments for an existing definition of the variable being assigned. If such a variable is found (and its binding is not locked) then its value is redefined, otherwise assignment takes place in the global environment.
see ¿'<-" for more info.

! Running lines of code

You can run multiple lines of code by selecting them in the coding window and pressing cmd+enter.
See example below, if you execute this code you will set x and y in the workspace of R and can then be used for further manipulation.

This can be useful if you want to compute in between steps, it is now recommended to only run Knitr code to produce the final report given that it will not use the data stored in the workspace but will always start from scratch.
Make sure that you have tell Rstudio to ouput chunks to the console not inline:

NOTE YOU MAY HAVE TO DO THIS FOR EACH TIME YOU OPEN AN .Rmd FILE FOR THE FIRST TIME FOR EACH SESSION ***

2. Data Types

Numbers

Decimal values are called numerics in R. It is the default computational data type. If we assign a decimal value to a variable x as follows, x will be of numeric type.

```
x = 10.5        # assign a decimal value
x               # print the value of x
```

```
## [1] 10.5
```

```
class(x)        # print the class name of x
```

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

Q2 if you assigning an integer value to a variable (e.g. k = 1), will the data value also be integer ? please check

---

```
k = 1
```

In order to create an integer variable in R, we invoke the as.integer function. We can be assured that k is indeed an integer by applying the is.integer function.

```
k = as.integer(k)
class(k)        # print the class name of y
```

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

```
is.integer(k)  # is y an integer?
```

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

It may be useful to perform arithmetic on logical values. Like the C language, TRUE has the value 1, while FALSE has value 0.

```
as.integer(TRUE)     # the numeric value of TRUE
```

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

```
as.integer(FALSE)    # the numeric value of FALSE
```

```
## [1] 0
```

LOGICAL

A logical value is often created via comparison between variables.

```
x = 1; y = 2
z = x > y       # is x larger than y?
z               # print the logical value
```

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

Standard logical operations are "&" (and), "|" (or), and "!" (negation)

```
u = c(TRUE, TRUE); v = c(FALSE,TRUE)
u & v           # Element-wise logical AND
```

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

```
u && v          # Logical AND
```

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

```
u | v           # Element-wise u OR v
```

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

```
u || v         # logical OR
```

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

```
!u              # negation of u
```

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

Standard relational operations

```
w ="truth" # a string
x = 0
w == "false" # note use double == for identity
```

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

```
x != 3 # x is not 3
```

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

```
x == 0 # note use double == for identity
```

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

```
x <= 0 # smaller or equal
```

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

```
x >= 0 # larger or equal
```

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

Further details and related logical operations can be found in the R documentation.

```
help("&")
```

STRING

String values are simply made by using “”

```
first = "Wouter"
last ="van den Bos"
#Two character values can be concatenated with the paste function.
paste(first, last)
```

```
## [1] "Wouter van den Bos"
```

VECTORS

A vector is a sequence of data elements of the same basic type. Members in a vector are officially called components.

Here is a vector containing three numeric values 2, 3 and 5.

```
v1 = c(2, 3, 5)
v1
```

```
## [1] 2 3 5
```

Here is one vector with strings

```
v2 = c("A", "B", "C", "D", "E")
v2
```

```
## [1] "A" "B" "C" "D" "E"
```

```
# or an empty vector can be useful
v3 = c()
```

Combining vectors works with the same function to create them

```
v4 = c(v1,v2,v3)
v4
```

```
## [1] "2" "3" "5" "A" "B" "C" "D" "E"
```

Value Coercion

In the code snippet above, notice how the numeric values are being coerced into character strings when the two vectors are combined. This is necessary so as to maintain the same primitive data type for members in the same vector.

Lets check the number of elements of this new vector

```
length(v3)
```

```
## [1] 0
```

Arithmetic operations of vectors are performed member-by-member, i.e., memberwise.

For example, suppose we have two vectors a and b.

```
a = c(1, 3, 5, 7)
b = c(1, 2, 4, 8)
```

Then, if we multiply a by 5, we would get a vector with each of its members multiplied by 5.

```
5 * a
```

```
## [1]  5 15 25 35
```

And if we add a and b together, the sum would be a vector whose members are the sum of the corresponding members from a and b.

```
a + b
```

```
## [1]  2  5  9 15
```

Similarly for subtraction, multiplication and division, we get new vectors via memberwise operations. Empty vectors can also be created

```
a - b
```

```
## [1]  0  1  1 -1
```

```
a * b
```

```
## [1]  1  6 20 56
```

```
a / b
```

```
## [1] 1.000 1.500 1.250 0.875
```

```
e = c()
```

Q3 the seq function is very useful ot make sequences of numbers, which will be saved as vectors. Look up how seq works and make a sequence starting at 0 ending at 100 in steps of 2?

```
seq(0,100,2)
```

```
##  [1]   0   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32
## [18]  34  36  38  40  42  44  46  48  50  52  54  56  58  60  62  64  66
## [35]  68  70  72  74  76  78  80  82  84  86  88  90  92  94  96  98 100
```

We can also construct a grid by combining to vectors using **expand.grid**
This can be usefull when you want to test a function for a combination of parameters

```
xgrid <- seq(1,3,1) # short for of seq
ygrid <- seq(-2,0,1)
grid <- expand.grid(xgrid, ygrid) # not limited to 2D can be longer list of lists
grid
```

```
##   Var1 Var2
## 1    1   -2
## 2    2   -2
## 3    3   -2
## 4    1   -1
## 5    2   -1
## 6    3   -1
## 7    1    0
## 8    2    0
## 9    3    0
```

expand.grid can also use with more than 2 variables

```
xgrid <- seq(1,3,1) # short for of seq
ygrid <- seq(-2,0,1)
zgrid <- seq(3,5,1)
grid <- expand.grid(xgrid, ygrid,zgrid)
head(grid) # use the head() function to just display the top of the table
```

```
##   Var1 Var2 Var3
## 1    1   -2    3
## 2    2   -2    3
## 3    3   -2    3
## 4    1   -1    3
## 5    2   -1    3
## 6    3   -1    3
```

We retrieve values in a vector by declaring an index inside a single square bracket "[]" operator.

For example, the following shows how to retrieve a vector member. Since the vector index is 1-based, we use the index position 3 for retrieving the third member.

```
v1 = c("A", "B", "C", "D", "E")
v1[3] # get the third member of the vector
```

```
## [1] "C"
```

Unlike other programming languages, the square bracket operator returns more than just individual members. In fact, the result of the square bracket operator is another vector, and s[3] is a vector slice containing a single member "C".

Negative Index

If the index is negative, it would strip the member whose position has the same absolute value as the negative index. For example, the following creates a vector slice with the third member removed.

```
v1[-3] # remove the third member of the vector (display what is left)
```

```
## [1] "A" "B" "D" "E"
```

extracting multiple elements

Vectors can be used to get multiple elements from a vector, even including duplicates:

```
v1[c(2,1,2,1)]
```

```
## [1] "B" "A" "B" "A"
```

To produce a vector slice between two indexes, we can use the colon operator ":". This can be convenient for situations involving large vectors.

```
v1[2:4]
```

```
## [1] "B" "C" "D"
```

Vector operations

min() and **max()**
for numerical vectors we can use **min()** and **max()** to get the min and max value

```
x = seq(1,10,.5)
min(x)
```

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

```
max(x)
```

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

If you want to know where to find the min or max in the vector:

```
which.max(x)
```

```
## [1] 19
```

```
which.min(x)
```

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

Besides **min()** and **max()** there are numerous operations you can peform on a vector. Here is a list of a few useful ones:

```
sum()

mean()

median()

range()

var() # the variance in the vector

sd() # standard devation
```

RANDOM Genetating random numbers or sequence of random numbers is also very useful for running simulations

```
set.seed(123) # set random seed such that you can replicate results!
rr =runif(100, 0, 1) # generates 100 random numbers between 0 and 1. unif indicates uniform distributio
```

Q4 find the minimum and maximum of the random numbers you just generated, and also the position in the list:

```
set.seed(123) # set random seed such that you can replicate results!
rr =runif(100, 0, 1) # generates 100 random numbers between 0 and 1. unif indicates uniform distributio
max(rr)
```

```
## [1] 0.9942698
```

```
which.max(rr)
```

```
## [1] 24
```

```
min(rr)
```

```
## [1] 0.0006247733
```

```
which.min(rr)
```

```
## [1] 74
```

Q5 try to plot all random numbers using plot() and identify the min and max in it using abline() to add (h)orizontal and (v)ertical lines

(hint: use ?plot and ?abline)

MATRIX A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. The following is an example of a matrix with 2 rows and 3 columns.

```
A = matrix( c(2, 4, 3, 1, 5, 7), nrow=2, ncol=3, byrow = TRUE)        # fill matrix by rows

A                      # print the matrix
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    3
## [2,]    1    5    7
```

An element at the mth row, nth column of A can be accessed by the expression A[m, n].

```
A[2, 3]      # element at 2nd row, 3rd column
```

```
## [1] 7
```

The entire mth row A can be extracted as A[m,].

```
A[2, ]       # the 2nd row
```

```
## [1] 1 5 7
```

Similarly, the entire nth column A can be extracted as A[,n].

```
A[ ,3]       # the 3rd column
```

```
## [1] 3 7
```

We can also extract more than one rows or columns at a time.

```
A[ ,c(1,3)]  # the 1st and 3rd columns
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    1    7
```

Transpose
We construct the transpose of a matrix by interchanging its columns and rows with the function t .

```
t(A)          # transpose of A
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    4    5
## [3,]    3    7
```

Combining Matrices

The columns of two matrices having the same number of rows can be combined into a larger matrix using **cbind**. If they have the same number of colums they can be combined using **rbind**.

Q6 Below are two lists, can you combine them using rbind, if not, why not and how could you change that?

```
L1 = matrix( c(2, 4, 3, 1, 5, 7), nrow=3, ncol=2, byrow = TRUE)
L2  = matrix( c(2, 4, 3, 1, 5, 7), nrow=3, ncol=2, byrow = F)
cbind(L1,L2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    4    2    1
## [2,]    3    1    4    5
## [3,]    5    7    3    7
```

LIST A list is a generic vector containing other objects.

For example, the following variable x is a list containing copies of three vectors n, s, b, and a numeric value 3.

```
n = c(2, 3, 5)
s = c("aa", "bb", "cc", "dd", "ee")
b = c(TRUE, FALSE, TRUE, FALSE, FALSE)
x = list(n, s, b, 3)   # x contains copies of n, s, b
y =list() # create an empty list
```

List Slicing

We retrieve a list slice with the single square bracket "[]" operator. The following is a slice containing the second member of x, which is a copy of s.

```
x[2]
```

```
## [[1]]
## [1] "aa" "bb" "cc" "dd" "ee"
```

Member Reference

In order to reference a list member directly, we have to use the double square bracket "[[]]" operator. The following object x[[2]] is the second member of x. In other words, x[[2]] is a copy of s, but is not a slice containing s or its copy.

```
x[[2]]
```

```
## [1] "aa" "bb" "cc" "dd" "ee"
```

We can even pull out one member of that list, and modify its content directly.

```
x[[2]][1] # get first member of second element in list
```

```
## [1] "aa"
```

```
x[[2]][1] = "ta" # change it
x[[2]]
```

```
## [1] "ta" "bb" "cc" "dd" "ee"
```

Data Frames

subset() This is a very useful function if you only want to use a subset of a dataframe:

check out the built in dataset cars

```
cars
```

```
##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 4      7   22
## 5      8   16
## 6      9   10
## 7     10   18
## 8     10   26
## 9     10   34
## 10    11   17
## 11    11   28
## 12    12   14
## 13    12   20
## 14    12   24
## 15    12   28
## 16    13   26
## 17    13   34
## 18    13   34
## 19    13   46
## 20    14   26
## 21    14   36
## 22    14   60
## 23    14   80
## 24    15   20
## 25    15   26
## 26    15   54
## 27    16   32
## 28    16   40
## 29    17   32
```

```
## 30    17   40
## 31    17   50
## 32    18   42
## 33    18   56
## 34    18   76
## 35    18   84
## 36    19   36
## 37    19   46
## 38    19   68
## 39    20   32
## 40    20   48
## 41    20   52
## 42    20   56
## 43    20   64
## 44    22   66
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

You will get a list of 50 cars wich are associated with two attributes speed and distance (dist). To select all cars that drive at speed 20:

```
twenties = subset(cars, cars$speed==20)
twenties
```

```
##    speed dist
## 39    20   32
## 40    20   48
## 41    20   52
## 42    20   56
## 43    20   64
```

```
test = subset(cars, cars$dist > 30)
test
```

```
##    speed dist
## 9     10   34
## 17    13   34
## 18    13   34
## 19    13   46
## 21    14   36
## 22    14   60
## 23    14   80
## 26    15   54
## 27    16   32
## 28    16   40
## 29    17   32
## 30    17   40
## 31    17   50
## 32    18   42
## 33    18   56
## 34    18   76
## 35    18   84
```

```
## 36    19   36
## 37    19   46
## 38    19   68
## 39    20   32
## 40    20   48
## 41    20   52
## 42    20   56
## 43    20   64
## 44    22   66
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

Q7 can you now select all cars that drove more that 30 kilometers in distance?

CONTROL STRUCTURES (WHILE IF THEN FOR ELSE)

What if....

The syntax of if statement is:

```
if (test_expression) {
   statement
}
```

If the test_expression is TRUE, the statement gets executed. But if it's FALSE, nothing happens.

Here, test_expression can be a logical or numeric vector, but only the first element is taken into consideration.

In the case of numeric vector, zero is taken as FALSE, rest as TRUE.

Example: if statement

```
x = 5
if(x > 0){
   print("Positive number")
}
```

```
## [1] "Positive number"
```

if...else statement The syntax of if...else statement is:

```
x = -3
if(x >= 0){
   print("Non-negative number")
} else {
   print("Negative number")
}
```

and nested ifs...

```r
x = -3
x <- 0
if (x < 0) {
   print("Negative number")
} else if (x > 0) {
   print("Positive number")
} else
   print("Zero")
```

Even more useful is the ifelse statement

```r
ifelse(test_expression,x,y)
```

Q8 can you use ifelse to tell for each element in the following list if it is even or odd ? (hint modulo: x %% y)

```r
a = c(5,7,2,9)
ifelse(a %% 2 == 0, 1, 0)
```

```
## [1] 0 0 1 0
```

For what?

Syntax of for loop

```r
for (val in sequence)
{
    statement
}
```

Here, sequence is a vector and val takes on each of its value during the loop. In each iteration, statement is evaluated.
Example

```r
for (i in 1:5)
{
print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

For a while

Syntax of while loop

```r
while (test_expression)
{
```

```
   statement
}
```

Here, test_expression is evaluated and the body of the loop is entered if the result is TRUE. The statements inside the loop are executed and the flow returns to evaluate the test_expression again. This is repeated each time until test_expression evaluates to FALSE, in which case, the loop exits.

Example:

```
i <- 1

while (i < 6) {
   print(i)
   i = i+1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Example of while Loop

```
i <- 1

while (i < 6) {
   print(i)
   i = i+1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

break statement A break statement is used inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop.

In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

```
x <- 1:5

for (val in x) {
    if (val == 3){
        break # kills the whole loop at 3
    }
    print(val)
}
```

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

I/O

how to read and write files, set and read the working directory

```
## set working direcotry
setwd("~/Dropbox/WOUTER/TEACHING/UvA/BREIN&COG/practicum/LC1/") ##<======= SET THIS TO YOUR LC1 FOLDER!
list.files() # check files
list.dirs() # check dirs
getwd() # cehck working dir
```

```
ebbing_data <- read.delim("Ebbinghaus.txt",header= TRUE, sep = "\t") # open a file, will be stored as d

write.table(ebbing_data,"test.txt") # write as tab delim
write.csv(ebbing_data,"test.csv") # write as comma seperated
x<-read.csv(file.choose()) # open dialog to open a csv file.
```

ANSWER SECTION

Q1 Try *?log* to read how to use log function. What are the two ways to calculate the log of 10 using base 10?

```
?log
log(10, 10)
log10(10)
```

Q2 if you assing an integer value to a variable (e.g. k = 1), will the data value also be integer

No, k will be of class numeric

```
k = 1
class(k)
```

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

Q3 the seq function is very useful ot make sequences of numbers, which will be saved as vectors. Look up how seq works and make a sequence starting at 0 ending at 100 in steps of 2?

```
seq(0,100,2)
```

```
##  [1]   0   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32
## [18]  34  36  38  40  42  44  46  48  50  52  54  56  58  60  62  64  66
## [35]  68  70  72  74  76  78  80  82  84  86  88  90  92  94  96  98 100
```

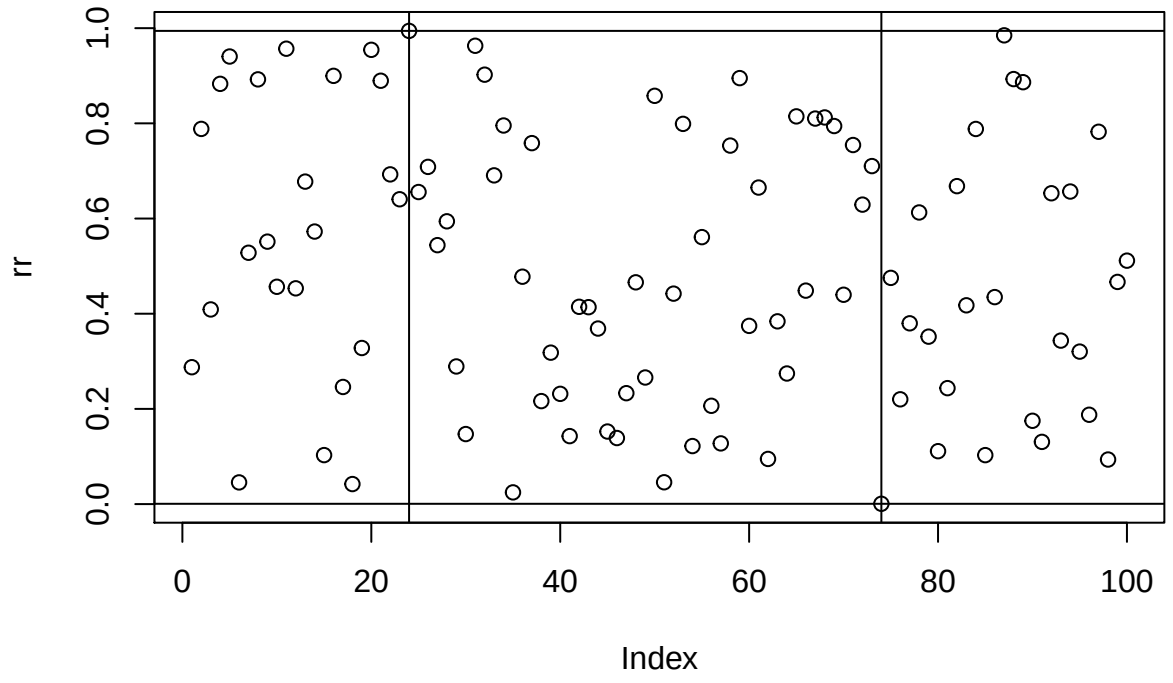
Q4 find the minimum and maximum of the random numbers you just generated, and also the position in the list:

```
set.seed(123) # set random seed such that you can replicate results!
rr =runif(100, 0, 1) # generates 100 random numbers between 0 and 1. unif indicates uniform distributio
max(rr)
which.max(rr)
min(rr)
which.min(rr)
```

Q5 try to plot all random numbers using plot() and identify the min and max in it using abline() to add (h)orizontal and (v)ertical lines

(hint: use ?plot and ?abline)

```
plot(rr)
abline(h = max(rr), v = which.max(rr))
abline(h = min(rr), v = which.min(rr))
```



Q6 Below are two lists, can you combine them using rbind, if not, why not and how could you change that?

No, for *rbind* the number of columns must be the same. Thus change ncol to same value

```
L1 = matrix( c(2, 4, 3, 1, 5, 7), nrow=3, ncol=2, byrow = TRUE)
L2  = matrix( c(2, 4, 3, 1, 5, 7), nrow=2, ncol=2, byrow = F)
rbind(L1,L2)
```

```
##      [,1] [,2]
## [1,]    2    4
## [2,]    3    1
## [3,]    5    7
## [4,]    2    3
## [5,]    4    1
```

Q7 can you now select all cars that drove more that 30 kilometers in distance?

```
test = subset(cars, cars$dist > 30)
test
```

Q8 can you use ifelse to tell for each element in the following list if it is even or odd ? (hint modulo: x %% y)

```
a = c(5,7,2,9)
ifelse(a %% 2 == 0, 1, 0)
```

```
## [1] 0 0 1 0
```