ALGONQUIN COLLEGE

SCHOOL OF ADVANCED TECHNOLOGY

ICT - Applications & Programming
Computer Engineering Technology – Computing Science

Numerical Computing – CST8233

A. Kadri

Lab #2 – R Programming part 1

This lab introduces rules for variable names, and how to manipulate vectors, matrices and lists. Also, it introduces functions in R.

Objectives

- Read and follow the instructions, and
- Complete the exercises at the end of this document.

You will need to show your completed work to your lab professor to get your grades.

Grades:

1% of your final course mark

Deadline

During the lab period of Week 4 (May 27th)

PART I

Step 1. Variables

Variables work much as you expect them to, but they are persistent between scripts in the same project. They have very specific rules for their naming.

A variable name may only contain

- Letters,
- Numbers,
- The period symbol, and
- The underscore

A variable name may only start with

- Letters, and
- The period symbol
  - But only when that symbol is **NOT** immediately followed by a digit.

Test yourself on the following. Which of these variable names are valid?

☐ `Integer_1`
☐ `Employee.1`
☐ `Integer%`
☐ `1Integer`

☐ `.Monthly_Salary`
☐ `.4Compound_Rate`
☐ `_Yearly_Rate`

There are three ways to assign a value to a variable:

- Leftward assignment (<-)
- Rightward assignment (->) and
- Equality operator (=)

Examples:

```
# Assignment using equal operator.
vect.1 = c(0,1,2,3)

# Assignment using leftward operator.
vect.2 <- c("Hello","World")

# Assignment using rightward operator.
c(TRUE,1) -> vect.3

print(vect.1)
cat ("vect.1 is ", vect.1 ,"\n")
cat ("vect.2 is ", vect.2 ,"\n")
cat ("vect.3 is ", vect.3 ,"\n")
```

You are already familiar with **print()**. The **cat()** function performs much the same job, but is slightly different. Experiment to see what the difference is.

The **class()** function returns the data type of a given variable. In the above example, determine each variable's type.

A number of functions in R behave similar to Unix commands. For instance, the **ls()** command will list all variables currently being used. Like *ls* in Unix, variables that start with a dot will not be shown. You can show these hidden variables this way:
ls(all.name = TRUE)

Like *rm* in Unix, the **rm()** function will delete a variable.

Step 2. Manipulation Data Structures

Manipulation of matrices and vectors is one of the most common tasks you will do in R.

2A: Manipulating Matrices

Create three matrices using three different methods as follows:

```
mat1 <- matrix(c(1,2,3,4,5,6,7,8,9), nrow = 3)
mat2 <- matrix(1:9, ncol = 3)
vect1 <- 1:9
mat3 <- matrix(vect1, nrow =3)
```

You can add, subtract, multiply (element-wise and algebra), and divide matrices as follows. Also, you can find the transpose of a matrix using the **t()** function.

```r
# adding two matrices
mat1 + mat2
# the product of linear algebra matrix multiplication
mat1 %*% mat2
# element by element multiplication
mat1 * mat2
# to find the transpose of a matrix (switch columns and rows)
t(mat1)
```

To access an element in a matrix, use the indices of the matrix much like you would if it were an array. You may also remove any row, column of any matrix or change the value of an element as shown below:

```r
# to access element in row 1 and column 3
mat1[1,3]
# to access all elements in row 2
mat1[2,]
# to access all elements in column 2
mat1[,2]
# to remove column 1
mat2[,-1]
# to remove row 1
mat3[-1,]
# to remove row 1 and column 2
mat1[-1,-2]
# to change a value of an element in a matrix
mat1[1,2] <- 15
```

Test yourself:

- Change the values of all elements of the first row of mat1 to 15.
- Change the values of all elements of the first and second columns of mat2 to 6.

So far, we have manipulated data based on the indices of the matrices. It is useful to manipulate the element based on its value:

```r
mat1 <- matrix(c(1,2,3,4,5,6,7,8,9), nrow = 3)
# to find all elements that have values less than 6
mat1[mat1 < 6]
# to find all elements that have values greater than or equal 6
mat1[mat1 >= 6]
# to see the Boolean result of each element
mat1 > 4
```

2B: Manipulating Vectors

Just like matrices, elements of vectors can be accessed using the square brackets "[" and "]". Accessing more than one element inside a vector can be done by using another vector. Input the following into RStudio to see this in action:

```
# Accessing vector elements using position.
t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")
u <- t[c(2,3,6)]
print(u)

# Accessing vector elements using logical indexing.
v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]
print(v)

# Accessing vector elements using negative indexing.
x <- t[c(-2,-5)]
print(x)

# Accessing vector elements using 0/1 indexing.
y <- t[c(0,0,0,0,0,0,1)]
print(y)
```

Vectors may be added, subtracted, multiplied and divided by each other:

```
# Create two vectors.
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)

# Vector addition.
add.result <- v1+v2
print(add.result)

# Vector subtraction.
sub.result <- v1-v2
print(sub.result)

# Vector multiplication.
multi.result <- v1*v2
print(multi.result)

# Vector division.
divi.result <- v1/v2
print(divi.result)
```

The **sort()** function can sort your vector contents. An example of reverse-sorting is included:

```
v <- c(13,8,41,2,0,11,-9)

# Sort the elements of the vector.
sort.result <- sort(v)
print(sort.result)

# Sort the elements in the reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)

# Sorting character vectors.
v <- c("green","Blue","Yellow","violet", "Green")
sort.result <- sort(v)
print(sort.result)

# Sorting character vectors in reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)
```

2C: Manipulating Lists

As shown in Lab 1, lists can contain elements of different types, such as numbers, strings, vectors, and even another list or a matrix. The elements of the list can be named with the **name()** function so they can be accessed by that name. The "$" operator is used to specify the name when accessing the list.

```
# Create a list containing a vector, a matrix and a list.
listA <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
                  list("green",12.3))

# Give names to the elements in the list.
names(listA) <- c("1st Quarter", "A_Matrix", "An_Inner_list")

# Show the list.
print(listA)
# Access the first element of the list.
print(listA[1])
# Access the third element. As it is also a list, all its
# elements will be printed.
print(listA[3])
# Access the list element using the name of the element.
print(listA$An_Inner_list)
```

Part II

Step 1. Built-in Math Functions

Similar to other programming languages, R provides various mathematical functions to perform mathematical calculations.

```
x <- -4.5
y <- c(1.2, 2.5,8.1)

# to return the absolute value of an input
print(abs(x))

# to returns the square root of an input
print(sqrt(abs(x)))

# to return the smallest integer which is larger than or equal to an input
print(ceiling(x))

# to return the largest integer, which is smaller than or equal to an input
print(floor(x))

# to return the truncate value of an input
print(trunc(y))
# to return round value of an input
z <- 3.142857
print(round(z, 2))

# to return cos(x), sin(x) value of an input
print(sin(z))
print(cos(z))
print(tan(z))

# to returns natural logarithm of an input
print(log(abs(x)))

# to returns common logarithm of an input
print(log10(abs(x)))

# to return the exponent of an input
print(exp(x))
```

Step 2. Built-in String Functions

R also provides various string functions to perform common tasks:

```
# to extract sub-strings in a character vector
a <- "Hello World!"
substr(a, 3, 4)

# to search for pattern in an input
st1 <- c('abcd','bdcd','abcdabcd')
pattern<- '^abc'
print(grep(pattern, st1))

# to find pattern in an input and replaces it with replacement (new) text
st1<- "England is beautiful but no the part of EU"
sub("England", "UK", st1)

# to concatenate strings after using "sep" string to separate them
paste('one',2,'three',4,'five')

# to split the elements of character an input vector at split point
a<-"Split all the character"
print(strsplit(a, ""))

# to convert the string into lower case
st1<- "HelLo WorLd"
print(tolower(st1))

# to convert the string into upper case
st1<- "HelLo WorLd"
print(toupper(st1))
```

Step 3. User-Defined Functions

It is possible to make your own functions, and then recycle them for use in other scripts. Both the function script and the other script files are in the same working directory. The structure of a user-defined function is as follows:

```
function.name <- function(arguments)
{
  # computations on the arguments
  # some other code
  # use "return" if the function return a value

  return(variable)
}
```

Attached below is an example script. It calculates the surface area of a circle, accepting the radius as its parameter:

```
Circlesurface <- function (radius)
  return(pi*radius^2)


# calling the function
a <- Circlesurface(8)
```

PART III

Step 1. Decision-Making and Loops

The if statement in R behaves in a way that should be familiar to you. **If()** accepts an expression that can be resolved to Boolean true or false. If true, certain code will be executed, **else** will trigger other code should the expression is false.

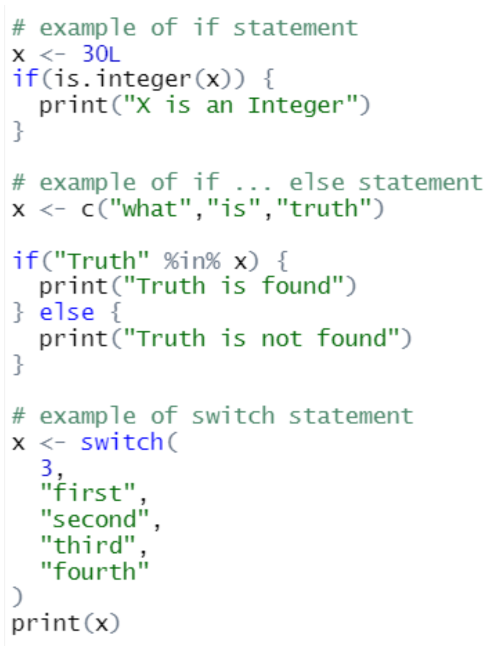
```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
}
```

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
} else {
   // statement(s) will execute if the boolean expression is false.
}
```

switch() behaves in much the same way as in other languages, but the logic is arranged a little differently:

```
switch(expression, case1, case2, case3....)
```

Attached below are some runnable examples of **if()** and **switch()** in operation:



```
# example of if statement
x <- 30L
if(is.integer(x)) {
  print("X is an Integer")
}

# example of if ... else statement
x <- c("what","is","truth")

if("Truth" %in% x) {
  print("Truth is found")
} else {
  print("Truth is not found")
}

# example of switch statement
x <- switch(
  3,
  "first",
  "second",
  "third",
  "fourth"
)
print(x)
```

There are three different ways to make repeating logic in R: **repeat**, **while()** and **for()**:

```
repeat {
   commands
   if(condition) {
      break
   }
}
```

```
while (test_expression) {
   statement
}
```

```
for (value in vector) {
   statements
}
```

Be mindful of your exit condition when using **repeat**. Attached below are some examples of the above loops:

```
# example of repeat loop
v <- c("Hello","loop")
cnt <- 2

repeat {
  print(v)
  cnt <- cnt+1

  if(cnt > 5) {
    break
  }
}

# example of while loop
v <- c("Hello","while loop")
cnt <- 2

while (cnt < 7) {
  print(v)
  cnt = cnt + 1
}

# example of for loop
v <- LETTERS[1:4]
for ( i in v) {
  print(i)
}
```

Part IV – Exercises

Write a new script for each of the following exercises.

1. Create a vector named **`cVec`** that stores the values of the following function:

   $$f(x) = 0.1\, e^{x} \cos x + 2\ln|x|$$

   at $x = 3, 3.1, 3.2, \dots, 6$. Find its summation and print it. The output of the print function must look like:

   "`The sum of this vector is: 256.6346`"

   Plot the vector using `plot()` function and add the title of this plot: "`My First Plot`".

   Note: the function $\cos()$ takes angels in radians.

2. Calculate the following:

   $$\sum_{i=1}^{25}\left(\frac{2^i}{i}+\frac{3^i}{i^2}\right)$$

   Print the value of this summation as follows:

   "`The sum of this summation is: 2129170437`"

3. Create two vectors, **`Vec1`** & **`Vec2`**, of random integers which are chosen with replacement from the range of $0, 1, \dots, 999$. The length of each vector is 100. Set the random seed to 75.

   a. Extract the values in **`Vec2`** which are greater than 600. Save the vector as **`Vec2a`**.

   b. What are the index positions in **`Vec2`** of these values. Save the vector as **`Vec2b`.** *Hint: you can use `which()` function for this step.*

   c. What are the values in **`Vec1`** which correspond to the values in **`Vec2`** which are greater than 600 (correspond means the same index positions). Save the vector as **`Vec1c`**.

   d. How many numbers in **`Vec1`** are divisible by 2.

4. Consider the continuous function

$$f(x) = \begin{cases} x^2 + 2x + 3 & if\ x < 0 \\ x + 3 & if\ 0 \leq x < 2 \\ x^2 + 4x - 7 & if\ 2 \leq x \end{cases}$$

Write a function called **`myFun`** which takes a single argument **`Vec1`**. The function should return the vector of values of the function $f(x)$ evaluated at the values of **`Vec1`**. Plot the function for $-4 \leq x < 4$.

When you have finished all three tasks, demo these to your professor.

> **"Programming isn't about what you know; it's about what you can figure out."** - *Chris Pine*