R- Programming

Dr. Saritha K
Associate Professor & HoD
Dept. of Computer Applications
Muthoot Institute of Technology and Science

Introduction to R

- R is a programming language and software environment for statistical computing and graphics.
- The R language is widely used among statisticians and data miners for developing statistical software.
- R can be downloaded and installed from CRAN repository.
- CRAN stands for Comprehensive R Archive Network

Continue….

- R is an open source maintained and developed by a community of developers.
- The R code repository, as well as compiled binaries (ready-to-install software) available at: http://cran.r-project .org
- R also has a large set of functions which provide a flexible graphical environment for creating various kinds of data presentations.

History

- The R programming language is an offshoot of a programming language called S.
- It was developed by Ross Ihaka and Robert Gentleman from the University of Auckland, New Zealand.
- It was primarily adopted by statisticians and is now the de facto standard for statistical computing.

How to Start With R

- R works on many operating systems including Windows, Macintosh, and Linux.
- Because R is free software it is hosted on many different servers around the world (mirrors) and can be downloaded from any of them.
- A list of all the available download mirrors is available here: http://www.r-project.org/index.html
- Step 1: Download R and install R: Windows installer version of R from R-3.2.2 for Windows (32/64 bit)
- Step 2: Download and install Rstudio **after you install R**

Continue….

- Rstudio is an Integrated Development Environment (IDE) for R
- R comes to ready to loaded with various libraries of functions called packages
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called repositories.
- To install packages using the command: install.packages()

Continue….

- **The working directory (wd)**

  – Like many programs, R has a concept of a working directory (wd)

  – It is the place where R will look for files to execute and where it will save files, by default

  – At the command prompt in the terminal or in RStudio console type:

  – **setwd("R_course/Day_1")**

  – Alternatively in RStudio use the mouse and browse to the directory → Session → Set Working Directory→ Choose Directory...

Continue….

- To remove ALL objects in your workspace:

```
> rm(list=ls())
```

Basic Operations

- The command line can be used as a calculator.

```
> 2+6
    [1] 8
> 20/5
    [1] 4
 > sqrt(25)
     [1] 5
> 3^2
     [1] 9
> sin(pi/2)
      [1] 1
```

- The number in the square brackets is an indicator of the position in the output
- Arithmetic operators are
  - + addition
  - - subtraction
  - * multiplication
  - / division
  - ^ raised to the power

Continue….

- **A variable** is a letter or word which takes (or contains) a value and tell R to store it as an object.
- We use the assignment operator as <- or =

```
x <- 10
 x
  [1] 10
myNumber <- 60
    myNumber
       [1] 60
```

- We can perform arithmetic functions on variables:

```
sqrt(myNumber)
   [1] 7.745967
```

- We can add variables together:

```
x + myNumber
  [1] 70
```

Continue….

- We can change the value of an existing variable:

```
x <- 21
x
  [1] 21
```

- We can set one variable to equal the value of another variable:

```
x <- myNumber
x
  [1] 25
```

- We can modify the contents of a variable:

```
myNumber <- myNumber + sqrt(16)
   [1] 29
```

Continue….

- **Functions** in R perform operations on **arguments** (the input(s) to the function).
- Arguments are always contained in parentheses, i.e. curved brackets **()**, separated by commas.

```
> sum(3, 4, 5, 6)
        [1] 18
> max(3, 4, 5, 6)
        [1] 6
> min(3, 4, 5, 6)
        [1] 3
```

- Arguments can be named or unnamed, but if they are unnamed they must be ordered

```
> sum(3:6)
      [1] 18
```

Continue….

```
> seq(from=2, to=10, by=2)
[1] 2 4 6 8 10
> seq(2, 10, 2)
[1] 2 4 6 8 10
 > round(14.3567,digits = 2)
[1] 14.36
```

Sample code is in sample_1.r

Continue….

Data types

- Logical

```
> x <- T; y <- F
> x; y
[1] TRUE
[1] FALSE
```

- Numerical

```
> a <- 5; b <- sqrt(2)
> a; b
[1] 5
[1] 1.414214
```

- Character

```
> a <- "1"; b <- 1
> a; b
[1] "1"
[1] 1
> a <- "character"
> b <- "a"; c <- a
> a; b; c
[1] "character"
[1] "a"
[1] "character"
```

Scan( ) function:-

- Reading data from the console using the scan function.
- The default separator for the **scan** function is any white space (single space, tab, or new line).
- Because the default is space delimiting, you can enter data on separate lines. When all the data have been entered, just hit the enter key twice which will terminate the scanning.

Readline() function

sample code is in read_keyboard.r

- **R- Objects**
  - Vectors
  - Lists
  - Matrices
  - Arrays
  - Factors
  - Data Frames

- Vectors
  - Ordered collection of data of the same data type
  - In R, single number is a vector of length 1
  - There are at least three ways to create vectors in R:

    (a) sequence function,
```
> x<-seq(from=2, to=10, by=2)
[1] 2 4 6 8 10
```
    (b) concatenation function,
```
> y<-c(2,4,6,8,10,12,14)
```
    (c) scan function.
```
> x<-scan()
```

```
a)   > x<-c(1,2,3,5)
     > length(x)
                 [1] 4
b)   > s<-rep(5,10)
                  [1] 5 5 5 5 5 5 5 5 5 5
c) > rep(c(5,8),3)
                 [1] 5 8 5 8 5 8
d) > v1 = c(1, 3, 2, -8.1)
                  [1] 1.0 3.0 2.0 -8.1
e) > apple<-c("red","green","yellow“)
                  [1] "red" "green" "yellow"
     > class(apple)
                 [1] "character"
```

Sample code is in vector_1.r, summary_vect.r

- ‘mode’ is a mutually exclusive classification of objects according to their basic structure.
- The ‘atomic’ modes are **numeric, complex, character ,logical, lists** .
- ‘class’ is a property assigned to an object that determines how to operate with it.
- If an object has no specific class assigned to it, such as a simple numeric vector, it’s class is usually the same as its mode, by convention.

- Lists
  - contain many different types of elements

```
> list1 <- list(c(2,5,3),21.3,”apple”)
    [[1]]
            [1] 2 5 3
    [[2]]
            [1] 21.3
    [[3]]
            [1] "apple"
```

Sample code is in list_1.r

- Matrices
  - A matrix is a two-dimensional rectangular data set.

```
     > M = matrix( c('a','a','b','c','b','a'),
nrow=2,ncol=3,byrow = TRUE)
          [,1] [,2] [,3]
     [1,] "a" "a" "b“
     [2,] "c" "b" "a"
```

  - Transpose of this matix is

```
     > t(M)
          [,1] [,2]
     [1,]  "a" "c"
     [2,]  "a" "b"
     [3,]  "b" "a"
```

- Sample code is in matrix_1.r

- Arrays
  - Arrays can be of any number of dimensions.
  - The array function takes a dim attribute which creates the required number of dimension.
    - > a <- array(c('green','yellow'),dim=c(3,3,2))
    - >print(a)

```
, , 1
 [,1] [,2] [,3]
 [1,] "green" "yellow" "green"
[2,] "yellow" "green" "yellow"
[3,] "green" "yellow" "green"
, , 2 [,1] [,2] [,3]
[1,] "yellow" "green" "yellow"
[2,] "green" "yellow" "green"
[3,] "yellow" "green" "yellow“
```

Sample code is in array_1.r

- Factors
  - It stores the vector along with the distinct values of the elements in the vector as labels.
    - apple_colors <- c('green','green','yellow','red','red','red','green')
    - factor_apple <- factor(apple_colors) print(factor_apple)
  - Know the number of distinct values using
    - print(nlevels(factor_apple))

Sample code is in factor_1.r

- Data Frames
  - Data frames are tabular data objects.
  - Unlike a matrix in data frame each column can contain different modes of data.
  - The first column can be numeric while the second column can be character and third column can be logical.
  - It is a list of vectors of equal length.
  - Data Frames are created using the data.frame() function.

-
  ```
  BMI <- data.frame(
                      gender = c("Male", "Male","Female"),
                      height = c(152, 171.5, 165),
                      weight = c(81,93, 78),
                      Age =c(42,38,26)
                       )
      print(BMI)
  ```

```
    gender height weight Age
  1 Male     152.0    81     42
  2 Male     171.5    93     38
  3 Female   165.0    78     26
```

- The statistical summary and nature of the data can be obtained by applying summary() function.
- The internal structure of R object is obtained using the function str().

Sample code – **data_frame_1.r and data_frame_2.r**

table()

- Table function in R performs a tabulation of categorical variable and gives its frequency as output. It is further useful to create conditional frequency table and Proportional frequency table.

- **Sort a Data Frame using Order()**
  - order() can sort vector, matrix, and also a data frame can be sorted in ascending and descending order.
  - The syntax of order() is shown below:

order(x, decreasing = TRUE or FALSE, na.last = TRUE or FLASE, method = c("auto", "shell", "quick", "radix"))

```
y = c(4,12,6,7,2,9,5)
order(y)
```

The above code gives the following output:

```
5 1 7 3 4 6 2
```

- Here the order() will sort the given numbers according to its index in the ascending order.
- Since number 2 is the smallest, which has an index as five and number 4 is index 1, and similarly, the process moves forward in the same pattern.

- y[order(y)]
  - The above code gives the following output:
  - 2 4 5 6 7 9 12
  - Here the indexing of order is done where the actual values are printed in the ascending order.

R Dplyr

- dplyr is a powerful R-package to transform and summarize tabular data with rows and columns.
- Hadley Wickham’s package dplyr has an optimized set of functions designed to work efficiently with data frames.
- One important contribution of the dplyr package is that it provides a “grammar” (in particular, verbs) for data manipulation and for operating on data frames.

- The package contains a set of functions that perform common data manipulation operations such as filtering for rows, selecting specific columns, re-ordering rows, adding new columns and summarizing data.
- In addition, dplyr contains a useful function to perform another common task which is the “split-apply-combine” concept.

- To install dplyr
  - install.packages("dplyr")
- To load dplyr
  - library(dplyr)

- **Important Verb Functions**
  - dplyr package provides various important function that can be used for Data Manipulation.
  - Some of the key “verbs” provided by the dplyr package are
    - select(): return a subset of the columns of a data frame, using a flexible notation
    - filter: extract a subset of rows from a data frame based on logical conditions
    - arrange( ): reorder rows of a data frame
    - rename(): rename variables in a data frame
    - mutate(): add new variables/columns or transform existing variables
    - summarise() : generate summary statistics of different variables in the data frame, possibly within strata

1. select()

- The select() function can be used to select columns of a data frame that you want to focus on.

2. filter() Function:

- The filter() function is used to extract subsets of rows from a data frame

3. arrange():

- The arrange() function is used to reorder rows of a data frame according to one of the variables or columns.

4. rename()

- Renaming a variable in a data frame in R is hard to do.
- The rename() function is designed to make this process easier.

5. mutate()

- The mutate() function exists to compute transformations of variables in a data frame.
- Often, we want to create new variables that are derived from existing variables and mutate() provides a clean interface for doing that.

6. summarise()

- The summarise() function will create summary statistics for a given column in the data frame such as finding the mean.

Sample code is- dplyr-example.r

Control Structures

- Control structures in R allow you to control the flow of execution of a series of R expressions.
- Basically, control structures allow us to put some “logic” into our R code, rather than just always executing the same R code every time.

Commonly used control structures are

- if and else: testing a condition and acting on it
- for: execute a loop a fixed number of times
- while: execute a loop while a condition is true
- repeat: execute an infinite loop (must break out of it to stop)
- break: break the execution of a loop
- next: skip an iteration of a loop

- **If statements**
  - Use an if statement for any kind of condition testing
  - Different outcomes can be selected based on a condition within brackets.

```
if( condition)
{ ## do something }
## Continue with rest of code
```

```
if(condition)
 { ## do something }
else
{ ## do something else }
```

```
if(condition 1)
   { ## do something }
else if(condition 2)
   { ## do something different }
else
   { ## do something different }
```

- **ifelse() Function**
  - **This is a shorthand function to the traditional if…else statement.**
  - **Syntax of ifelse() function**

    ifelse(test_expression, x, y)
  - Here, test_expression must be a logical vector.
  - The return value is a vector with the same length as test_expression.

Sample code- if-example.r

- **for Loops**
  - syntax is

    ```
    for (variable in sequence)
    { expression }
    ```

  - Example

    ```
    for (i in vector)
            { Exp }
    ```

  - The expression can be a single R command - or several lines of commands wrapped in curly brackets:

Sample code- for-example.r

while loop

- The syntax is

```
while (condition)
     { expression }
```

Functions in R

- Functions are defined using the function() directive and are stored as R objects just like anything else.
- In particular, they are R objects of class “function”.
- Here’s a simple function that takes no arguments and does nothing.

```
f <- function()
        {  ## This is an empty function  }
```

An Example

- The following function takes a single input value and computes its square.

```
square <- function(x)
          {       x * x    }
```

- ➢ The function is created and then assigned the name square.
- ➢ The variable, x, is a formal parameter of the function.
- ➢ When the function is called it is passed an argument that provides a value for the formal parameter.

```
square(1:5)
          [1] 1 4 9 16 25
```

Lazy Evaluation of Function

- R function arguments are not evaluated until the value of the argument is needed.
- Example 1

```
 f <- function(a, b)
{  print(a)
 print(b)
 }
  f(45)
[1] 45
 Error in print(b): argument "b" is missing, with no
default
```

- In this example, the function f() has two arguments: a and b.

```
f <- function(a, b)
          { a^2  }
f(2)
    [1] 4
```

- This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

Sample code- fun-example.r

Loop Functions in R

- R has some functions which implement looping in a compact form.
  - apply(): Apply a function over the margins of an array
  - lapply(): Loop over a list and evaluate a function on each element
  - sapply(): Same as lapply but try to simplify the result
  - tapply(): Apply a function over subsets of a vector

- apply()

  apply(X, margin, FUN)

  - X is the variable you want to apply the **function** to.
  - **margin** specifies if you want to apply by row (margin = 1), by column (margin = 2), or for each element (margin = 1:2). Margin can be even greater than 2, if we work with variables of dimension greater than two.
  - FUN is a function to be applied

- lapply():

  lapply(X, FUN)

  - X: A vector or an object
  - FUN: Function applied to each element of x
- l in lapply() stands for list.
- The difference between lapply() and apply() lies between the output return.
- The output of lapply() is a list. lapply() can be used for other objects like data frames and lists.
- lapply() function does not need MARGIN.

- sapply():
  - **sapply()** function takes list, vector or data frame as input and gives output in vector or matrix.
  - It is useful for operations on list objects and returns a list object of same length of original set.
  - Sapply() function in R does the same job as lapply() function but returns a vector.

sapply(X, FUN)

- X: A vector or an object
- FUN: Function applied to each element of x
- The result of lapply() was a list where each element had length 1, sapply() collapsed the output into a numeric vector, which is often more useful than a list

- tapply():
  - tapply() is used to apply a function over subsets of a vector.
  - The “t” in tapply() refers to “table”.

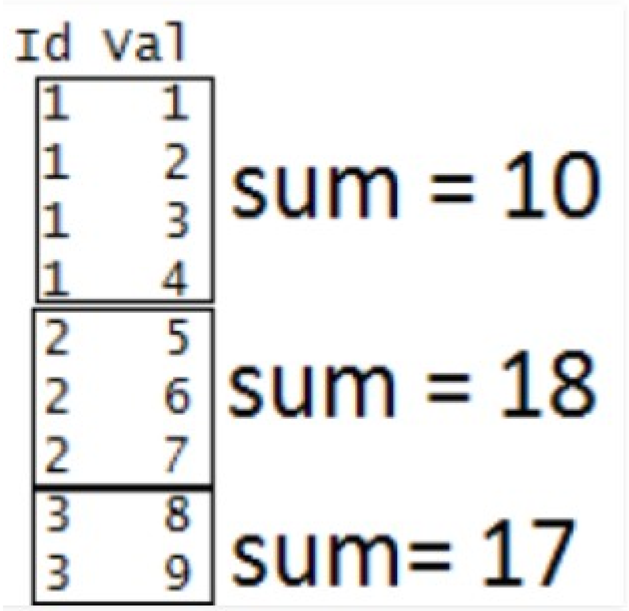
tapply(X, INDEX, FUN = NULL)

- Arguments: -
  - X: An object, usually a vector
  - INDEX: A list containing factor -FUN: Function applied to each element of x

- # Creating a factor
- Id = c(1, 1, 1, 1, 2, 2, 2, 3, 3)

- # Creating a vector
- val = c(1, 2, 3, 4, 5, 6, 7, 8, 9)

- # applying tapply()
- result = tapply(val, Id, sum)
- print(result)



Id	Val	
1	1	sum = 10
1	2	
1	3	
1	4	
2	5	sum = 18
2	6	
2	7	
3	8	sum= 17
3	9	

Output

```
1        2        3
10       18       17
```

Sample code- loop_fun.r

THANK YOU