UNIT-II

Data Types of R-Vectors, Matrices, Arrays, Lists, Factors, Data Frame-Basic Expressions Vectors – sequences, lengths, names, indexing vectors, vector recycling and repetition, Matrices and Arrays – creating arrays and matrices, rows, columns, dimensions, indexing arrays, Arithmetic. Decision-Making Structures – Loops, User Defined Functions, User-Defined Package, Reports using R markdown, Conditional – if and else, vectorized if, multiple selection, Loops – repeat loops, while loops, for loops, Advanced looping – replication, looping over lists, looping over arrays, Multiple – input apply, instant vectorization.

R VECTORS

R operates on named data structures. The simplest such structure is the vector, which is a single entity consisting of an ordered collection of different atomic vectors, also termed as six classes of vectors.

* The category of the vector is determined by using class function.
* A vector of multiple elements can be created normally using six different ways
* Elements of a Vector can be accessed by the process named indexing.
* Different functions that can be used on multiple elements of the vector include sorting, ranking etc.
* Different types of operations on vectors include: Mathematical functions, Arithmetic, Relational and Logical Operators.

Matrices

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout.

* A matrix in R can be directly created by specifying the data values and number of rows and columns, through vectors, by using cbind or rbind function for vectors etc.
* Elements of a matrix are stored using the column and row index of the element. Thus x[i,j] will denote element belonging to j column of i row.
* Different operations are performed on matrices in R, which are generally performed in mathematics including addition and subtraction of matrices, calculation of transpose, multiplication of matrices, addition of matrices, inverse of a matrix etc.

Arrays

Arrays are the R data objects which can store data in more than two dimensions unlike matrix. For example - If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns.

* An array is created using the array() function. It takes vectors as input and uses the values in the dim parameter to create an array. We can give names to the rows, columns and matrices in the array by using the dim names parameter.
* An array can be considered as a multiple subscripted collection of data entries. The element of an array is represented as x[i,j,k] where i denotes the row number, j denotes the column number and k denotes the matrix number. The elements can be accessed using proper indices.
* An array can also be considered as a collection of matrices. Hence, a matrix can be directly created from an array using proper subscripts.
* We can do calculations across the elements in an array using the apply()function.

Lists

R list is an object consisting of an ordered collection of objects known as its components. All the components are numbered and it is not necessary that components are of the same data type. It can even have list inside it.

* A list can be created by using the list function with different data objects.
* List elements / Components of lists may be given a name and can be accessed using these names. Accessing of lists in R can be done using three ways:. [[operator can be used with computed indices ; named indices and the $ operator can only be used with literal names.
* Various operations can be applied on the elements of list like data manipulation and concatenation of different lists. List can also be converted to a matrix or to any other data object in R.

Factors

A factor is a vector object used to specify a discrete classification (grouping) of the components of other vectors of the same length. Factors **r**epresent categorical variables and are used as grouping indicators.

* There are different ways of creating a factor which include creating a factor from a vector,gl() function and ordered() function
* As discussed above, the levels (distinct values) of the factor are displayed after the values are printed. We can print the number of distinct values in the vector using nlevel() function.
* Different functions can be used on factors which include table(), unclass() etc.

Data Frame

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column. Unlike a matrix in data frame each column can contain different modes of data.

* Data frames are generally created by reading in a dataset using importing of different files from other software’s. However, data frames can also be created by using data.frame function or by joining columns and rows using cbind() and rbind().
* We can extract specific column from a data frame using $ sign succeeded by column name or by a column number, while a row can be extracted by specifying the row numbers only.
* Different functions like str, col names, row names, summary etc. can be used on a data frame to view the details.

VECTOR BASIC EXPRESSIONS

# Vector of strings  
fruits <- c("banana", "apple", "orange")  
  
# Print fruits  
fruits

# Vector of numerical values  
numbers <- c(1, 2, 3)  
  
# Print numbers  
numbers

To create a vector with numerical values in a sequence, use the : operator:

# Vector with numerical values in a sequence  
numbers <- 1:10  
  
numbers

You can also create numerical values with decimals in a sequence, but note that if the last element does not belong to the sequence, it is not used:

# Vector with numerical decimals in a sequence  
numbers1 <- 1.5:6.5  
numbers1  
  
# Vector with numerical decimals in a sequence where the last element is not used  
numbers2 <- 1.5:6.3  
numbers2

In the example below, we create a vector of logical values:

# Vector of logical values  
log\_values <- c(TRUE, FALSE, TRUE, FALSE)  
  
log\_values

To find out how many items a vector has, use the length() function:

fruits <- c("banana", "apple", "orange")  
  
length(fruits)

o sort items in a vector alphabetically or numerically, use the sort() function:

fruits <- c("banana", "apple", "orange", "mango", "lemon")  
numbers <- c(13, 3, 5, 7, 20, 2)  
  
sort(fruits)  # Sort a string  
sort(numbers) # Sort numbers

## Access Vectors

You can access the vector items by referring to its index number inside brackets []. The first item has index 1, the second item has index 2, and so on:

fruits <- c("banana", "apple", "orange")  
  
# Access the first item (banana)  
fruits[1]

access multiple elements by referring to different index positions with the c() function:

fruits <- c("banana", "apple", "orange", "mango", "lemon")  
  
# Access the first and third item (banana and orange)  
fruits[c(1, 3)]

use negative index numbers to access all items except the ones specified:

fruits <- c("banana", "apple", "orange", "mango", "lemon")  
  
# Access all items except for the first item  
fruits[c(-1)]

To change the value of a specific item, refer to the index number:

fruits <- c("banana", "apple", "orange", "mango", "lemon")  
  
# Change "banana" to "pear"  
fruits[1] <- "pear"  
  
# Print fruits  
fruits

To repeat vectors, use the rep() function:

repeat\_each <- rep(c(1,2,3), each = 3)  
  
repeat\_each

Repeat the sequence of the vector:

repeat\_times <- rep(c(1,2,3), times = 3)  
  
repeat\_times

Repeat each value independently:

repeat\_indepent <- rep(c(1,2,3), times = c(5,2,1))  
  
repeat\_indepent

## Generating Sequenced Vectors

numbers <- 1:10  
  
numbers

To make bigger or smaller steps in a sequence, use the seq() function:

numbers <- seq(from = 0, to = 100, by = 20)  
  
numbers

Matrix

A matrix is a two dimensional data set with columns and rows.

A column is a vertical representation of data, while a row is a horizontal representation of data.

A matrix can be created with the matrix() function. Specify the nrow and ncol parameters to get the amount of rows and columns:

# Create a matrix  
thismatrix <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)  
  
# Print the matrix  
thismatrix

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
thismatrix

[,1] [,2]

[1,] "apple" "cherry"

[2,] "banana" "orange"

## Access Matrix Items

You can access the items by using [ ] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
**thismatrix[1, 2]**

The whole row can be accessed if you specify a comma **after** the number in the bracket:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

**thismatrix[2,]**

"banana" "orange"

The whole column can be accessed if you specify a comma **before** the number in the bracket:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
**thismatrix[,2]**

"cherry" "orange"

## Access More Than One Row

More than one row can be accessed if you use the c() function:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
thismatrix[c(1,2),]

[,1] [,2] [,3]

[1,] "apple" "orange" "pear"

[2,] "banana" "grape" "melon"

## Access More Than One Column

More than one column can be accessed if you use the c() function:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
thismatrix[, c(1,2)]

## Add Rows and Columns

Use the cbind() function to add additional columns in a Matrix:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
newmatrix <- cbind(thismatrix, c("strawberry", "blueberry", "raspberry"))

[,1] [,2] [,3] [,4]

[1,] "apple" "orange" "pear" "strawberry"

[2,] "banana" "grape" "melon" "blueberry"

[3,] "cherry" "pineapple" "fig" "raspberry"

# Print the new matrix  
newmatrix

Use the rbind() function to add additional rows in a Matrix:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
newmatrix <- rbind(thismatrix, c("strawberry", "blueberry", "raspberry"))  
  
# Print the new matrix  
newmatrix

[,1] [,2] [,3]

[1,] "apple" "orange" "pear"

[2,] "banana" "grape" "melon"

[3,] "cherry" "pineapple" "fig"

[4,] "strawberry" "blueberry" "raspberry"

## Remove Rows and Columns

Use the c() function to remove rows and columns in a Matrix:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange", "mango", "pineapple"), nrow = 3, ncol =2)  
  
#Remove the first row and the first column  
thismatrix <- thismatrix[-c(1), -c(1)]  
  
thismatrix

## Number of Rows and Columns

Use the dim() function to find the number of rows and columns in a Matrix:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
dim(thismatrix)

## Matrix Length

Use the length() function to find the dimension of a Matrix:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
length(thismatrix)

## Loop Through a Matrix

You can loop through a Matrix using a for loop. The loop will start at the first row, moving right:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
for (rows in 1:nrow(thismatrix)) {  
  for (columns in 1:ncol(thismatrix)) {  
    print(thismatrix[rows, columns])  
  }  
}

## Combine two Matrices

Again, you can use the rbind() or cbind() function to combine two or more matrices together:

# Combine matrices  
Matrix1 <- matrix(c("apple", "banana", "cherry", "grape"), nrow = 2, ncol = 2)  
Matrix2 <- matrix(c("orange", "mango", "pineapple", "watermelon"), nrow = 2, ncol = 2)  
  
# Adding it as a rows  
Matrix\_Combined <- rbind(Matrix1, Matrix2)  
Matrix\_Combined  
  
# Adding it as a columns  
Matrix\_Combined <- cbind(Matrix1, Matrix2)  
Matrix\_Combined

## Arrays

Compared to matrices, arrays can have more than two dimensions.

We can use the array() function to create an array, and the dim parameter to specify the dimensions:

# An array with one dimension with values ranging from 1 to 24  
thisarray <- c(1:24)  
thisarray  
  
# An array with more than one dimension  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
multiarray

## Access Array Items

You can access the array elements by referring to the index position. You can use the [] brackets to access the desired elements from an array:

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
multiarray[2, 3, 2]

You can also access the whole row or column from a matrix in an array, by using the c() function:

thisarray <- c(1:24)  
  
# Access all the items from the first row from matrix one  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
multiarray[c(1),,1]  
  
# Access all the items from the first column from matrix one  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
multiarray[,c(1),1]

## Check if an Item Exists

To find out if a specified item is present in an array, use the %in% operator:

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
2 %in% multiarray

## Amount of Rows and Columns

Use the dim() function to find the amount of rows and columns in an array:

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
dim(multiarray)

## Array Length

Use the length() function to find the dimension of an array:

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
length(multiarray)

## Loop Through an Array

You can loop through the array items by using a for loop:

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
for(x in multiarray){  
  print(x)  
}

3.1 Decision Making Structures

They require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

3.1.1 If Structure

It is the general form of a typical decision making structure found in most of the programming languages. When we need to execute a set of statements based on a condition then we need to use if structure. There are four types of if statements that we can use in R programming based on the requirement:

* **a) If statement**
* **b) If…else statement**
* **c) Nested-if**
* **d) if-else-if ladder**

3.1.2 Switch StatementA switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case. We can have any number of case statements within a switch.

3.2 Loops

A loop statement allows us to execute a statement or group of statements multiple times. Repetitive commands are executed by loops. There are three kinds of loops including for loop, while loop and repeat loop. R loops are particularly flexible and they are not limited to integers. We can also use loops for character vectors, logical vectors, lists or expressions.

3.2.1 For Loop

If the number of repetitions is known in advance then a for() loop can be used. For executing for loop, we need to specify a variable and a vector. The variable specified is sequentially set to all the values, which are contained in the vector and all operations/commands are executed for all these values.

**Syntax :** for (var in n) {commands to be executed}

3.2.2 While Loop

The While loop executes the same code again and again until a stop condition is met. A condition is tested in starting of a while loop.

**Syntax:** while (condition) {commands to be executed as long as condition is TRUE }

3.2.3 Repeat Loop

The repeat loop doesn’t test any condition in contrast to the while() loop before entering the loop and also not during the execution of the loop for the termination of the loop. The programmer terminates the loop after the appropriate number of iterations using break command.

**Syntax:** repeat{commands to be executed}

3.3 User Defined Functions

A function is a set of statements organized together to perform a specific task. Functions are very much useful when a block of statements has to be written / executed again and again. Thus, the function is a standard unit of reuse. Functions are useful when the program size is too large or complex. Functions are called to perform each task sequentially from the main program. It is like a top-down modular programming technique to solve a problem. Functions are also used to reduce the difficulties during debugging a program. In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions. The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

3.3.1 Function without Arguments

A function can be created with without any arguments :

Example:

>myfunc2<- function(){

+ print("Hello,world!")

+ }

3.3.2 Function with Arguments

It has been observed that user defined function with no argument does not allow the user to change the input given to the function. But, generally, the user need to pass the argument, which change with the requirement. It is necessary that number of arguments while creating a function should be equal to number of arguments passed while calling function. We can pass any number of arguments inside the function

>myfunction <- function(arg1) { for (i in 1:arg1)

+ { result <- i\*3

+ print(result) } }

3.3.3 Nesting of Functions

We can define one function within other function. Due to simple recursive rules, inside function is itself invisible outside of its immediately enclosing function, but can see (access) all local objects (data, functions, types, etc.) of its immediately enclosing function as well as of any function(s) which, in turn, encloses that function. Function inside other function is theoretically possible to unlimited depth, although only a few levels are normally used in practical programs.

**3.4 User Defined Package**

R also provides an extensive facility to develop our own package and upload on the CRAN to facilitate other users with the defined data and functions. Before uploading a new package the user needs to create a user defined package which should contain some functions along with data file.

3.5 Reports using Rmarkdown

A report is created using render unction which is available in rmarkdown package which can be either downloaded from CRAN directory as a zip file or installed. It should be noted that if the file is downloaded from CARN directory, the zip file should be pasted in r-library directory. click the library function to get the details of the path of library. If not supportive, we can install the new package rprojroot. This package will help us to compile a file with extension rmd (rmarkdown) and tex. If successful compilation takes place then it can help us to render to either a html file or pdf file or word document. Rendering can be done in two ways: Direct Rendering and Indirect Rendering.