**Programming with R**

**Module 2: Data Management**

**Data Structures**

We will look at five basic data structures that are available in R, namely, vector, factors, matrix, data frame and list.

**Vectors and assignment**

It is the simplest of structure. A vector is collection of values that all have the same data type. If the elements of a vector are all numbers, it is a numeric vector, or if all are character values, it is a character vector. A vector cannot be of mixed type. A vector can be created using the assignment statement using the function c(). The ‘c’ stands for combine.

**#** Vectors

register\_no <- c(1001:1005)

participant\_name <- c("Anil", "Badri", "Chetna", "Dinesh", "Elisa")

test\_1\_marks <- c(35,42,47,23,37)

test\_2\_marks <- c(43,32,40,37,35)

**Vector operations**

Vectors can be used in arithmetic expressions in which case the operations are performed element by element.

**#** Vector operations

test\_1\_marks + 3

sqrt(test\_1\_marks)

total\_marks <- test\_1\_marks + test\_2\_marks

print(total\_marks)

average\_marks <- total\_marks/2 # average marks of each student from two tests

mean(total\_marks) # average mark of all the students

max(test\_1\_marks)

sort(test\_2\_marks)

range(test\_2\_marks)

test\_1\_marks > 50

any(test\_2\_marks > 50)

all(total\_marks < 100)

test\_2\_marks > test\_1\_marks

length(register\_no)

nchar(participant\_name)

is.na(test\_2\_marks)

anyNA(test\_1\_marks)

participant\_income <- c(60000, 100000, 45000, NA, 70000)

participant\_income

**Accessing vectors**

Vector elements are accessed using indexing vectors. We can access an individual element of a vector by its position (or "index"), indicated using square brackets. In R, the first element has an index of 1.

**#** Accesing vector elements

test\_1\_marks[1]

participant\_name[1:3]

register\_no[c(1,3)]

participant\_name[-1]

**Factor Vectors**

Factors are like vectors, and can mostly be treated as such, but they have another tier of information. A factor keeps track of all the distinct values in that vector and notes the positions in the vector where each distinct value can be found. Factors are R's preferred way of storing categorical data. The set of distinct values are called levels. R provides both ordered and unordered factors.

**#** Factor Vectors

participant\_sport <- c("Hockey","Cricket","Basketball","Hockey","Basketball")

sport\_factor <- as.factor(participant\_sport)

sport\_factor

participant\_qualification <- factor(c("UG","PhD","UG","PG","UG"),

levels = c("UG","PG","PhD"), ordered = TRUE)

participant\_qualification

**Matrices**

In R, a matrix is a collection of elements of the same data type (numeric, character, or logical) arranged into a fixed number of rows and columns. Since we are only working with rows and columns, a matrix is called two-dimensional. In order to access elements of matrix, we must index two positions, one for row and the other for column. They also act like vectors with element-by-element addition, multiplication, subtraction, division, and equality. A very common way of assembling a matrix is by combining vectors or matrices. They can be stacked by row or column.

**#** Matrices

mat\_A <- matrix(1:10, nrow = 5)

mat\_A

mat\_B <- matrix(1:10, ncol = 5)

mat\_B

mat\_C <- matrix(11:20, 5)

mat\_C

mat\_D <- matrix(11:20, , 5)

mat\_D

add\_mat <- mat\_A + mat\_C

rbind(mat\_A, mat\_C)

cbind(mat\_A, mat\_C)

mul\_mat <- mat\_A %\*% mat\_B # size of two mat should be mxn and nxm to multiply

mul\_mat

mat\_A == mat\_C

mat\_A %\*% t(mat\_C) # t is used for transpose

colnames(mat\_A) <- c("Level 1", "Level 2")

rownames(mat\_A) <- c("A", "B", "C", "D", "E")

print(mat\_A)

mat\_B <- matrix(1:10, ncol=5,

dimnames = list(c("Row1","Row2"),

c("Col1","Col2","Col3","Col4","Col5")))

mat\_B

mat\_A %\*% mat\_B # observe row and col names

# Compare matrix C and E

mat\_E <- matrix(11:20,5, byrow = TRUE)

mat\_E

**#** To check the dimensions of a matrix

dim(mat\_E)

nrow(mat\_B)

ncol(mat\_D)

length(mat\_E)

dim(mat\_E) <- c(2, 5) # to reshape a matrix

mat\_E

**#** Extract elements of a matrix

mat\_E[1, 2]

mat\_E[1, ]

mat\_E[ ,3:5]

mat\_E[3]

**Data Frames**

Data frame is R’s name for tabular data. Data frames are two-dimensional data structures like matrices, but, unlike matrices, they can contain multiple different data types. We can think of a data frame wherein each column is a vector, each of which has the same length. This implies that within a column each element must be of the same type. Data frames are commonly used to represent tabular data. A data frame can be used to represent an entire data set.

We generally want each row in a data frame to represent a unit of observation, and each column to contain a different type of information about the units of observation (variable). Tabular data in this form can be called tidy data.

There are numerous ways to construct a data frame, the simplest being to use the data.frame function. The function data.frame() creates a data frame object from a set of vectors. Data frames are displayed in a tabular layout, with column names above and row numbers to the left.

**#** Data Frames

# below are vectors already defined

participant\_name

participant\_qualification

participant\_income

participant\_sport

**#** creating the DF

df1.participant <- data.frame(participant\_name, participant\_qualification,

participant\_income, participant\_sport)

df1.participant

# simplify column names

names(df1.participant) <- c("name", "qualification", "income", "sport")

colnames(df1.participant)

**#** To check various attributes of a DF

class(df1.participant)

nrow(df1.participant)

ncol(df1.participant) # alternatively length()

dim(df1.participant)

names(df1.participant)

colnames(df1.participant)[2]

rownames(df1.participant)

head(df1.participant)

head(df1.participant, 2)

tail(df1.participant, 2)

df1.participant$name

df1.participant$sport

df1.participant[1, 3]

df1.participant[ , 3]

df1.participant["income"]

df1.participant[1, ]

df1.participant[1, c(1,3)]

df1.participant[ , c(1,3)]

df1.participant[[1]][2:3]

is.na(df1.participant)

**#** Manipulation of a DF

# Adding columns and rows to a DF

register\_no

total\_marks

df2.participant <- data.frame(register\_no, total\_marks)

df3.participant <- cbind(df1.participant, df2.participant)

df3.participant

df4.participant <- cbind(df1.participant$name, df2.participant,

df1.participant[ ,-1])

df4.participant

new\_admission <- data.frame(name = "Newguy",

qualification = "Dip",

sport ="Football",

income =150000,

register\_no = 1006,

total\_marks = 35)

rbind(df3.participant,new\_admission)

# rbind is smart enough to reorder the columns (sport/income) to match

**#** To merge two DF horizontally

fee <- c(100000, 75000, 80000, 100000, 0)

df5.partcipant <- data.frame(register\_no, fee)

df5.partcipant

merge(df3.participant, df5.partcipant)

merge(df3.participant, df5.partcipant, by = "register\_no")

**#** Sorting

df3.participant[order(df3.participant$total\_marks),]

df3.participant[order(-df3.participant$income),]

**#** Subsetting a DF

df6.participant <- df3.participant[c(-2:-4)]

df6.participant

subset(df3.participant, select = -c(2:4)) # alternate way

**#** Ediitng data frame

# Task is to add initials to all names

df6.new <- edit(data.frame(df6.participant))

df6.new

**Lists**

A list in R is like a container, which can hold arbitrary objects of either the same type or different. A list is a very flexible data structure. It can have any number of components, each of which can be any data structure of any length or size. Lists are created with the list() function where each argument to the function becomes an element of the list. The results of many high-level analyses in R are packaged as lists.

A list is similar to a folder on our computer system. A folder contains multiple files of different types and sizes. The folder can contain other folders also.

The most straightforward way to refer to a single component of a list is using the double square brackets [[ ]] notation.

**#** Lists

list\_1 <- list(1:5, c("a","b","c","d","e"))

list\_1

list\_2 <- list(df3.participant, mat\_A, fee, myname = "DG")

list\_2[[2]]

list\_2[[5]] <- list\_1

list\_2

**Recap**

* A vector is a one-dimensional data structure and all its elements are of the same data type.
* A factor is one-dimensional and every element must be one of a fixed set of values, called the levels of the factor.
* A matrix is a two-dimensional data structure and all its elements are of the same type.
* A data frame is two-dimensional and different columns may contain different data types, though all values within a column must be of the same data type and all columns must have the same length.
* A list is a hierarchical data structure and each component of a list may be any type of data structure whatsoever.

**csv Files – An intro**

CSV stands for Comma Separated Values where the data lives in a simple text file that is split up by commas (delimiter). CSV files are used as a way to communicate data between different applications. CSV is a common data exchange format that is widely supported by consumer, business, and scientific applications. Among its most common uses is moving tabular data. Since the CSV file format is extremely simple and lightweight it is easier for different applications to support it.

DIY: Create a csv file from a Notepad (Text Document)

**Import and Export of data**

R can read data from a wide variety of sources and in a wide variety of formats.

**Reading CSVs:**

The best way to read data from a csv file is to use read.table() function. The read.table function reads delimited files and stores the results in a data frame.

The first argument of read.table should be the name of your file (if it is in your working directory), or the file path to your file (if it is not in your working directory). There are other arguments as well, important being *sep* and *header*. Use sep to tell read.table what character your file uses to separate data entries. Use header to tell read.table whether the first line of the file contains variable names instead of values.

The read.table is a general purpose read function and there are several convenient wrapper functions to it. The read.csv can be directly used to read a csv file, here the default separator is set to comma and it is assumed that the data has a header row.

Tip: You can also load plain text files with RStudio’s Import Dataset button. RStudio will also open a data viewer, so you can see your new data in a spreadsheet format.

**#** Reading CSVs

# Download Nifty data as csv to your working directory from -

# https://www1.nseindia.com/products/content/equities/indices/historical\_index\_data.htm

location1 <- "D:/CMS/R\_2020/nifty\_data\_2019.csv" # copy applicable path

nifty\_values <- read.table(file = location1, sep = ",", header = TRUE)

nifty\_csv <- read.csv(location1) # alternate way

class(nifty\_csv)

head(nifty\_csv)

View(nifty\_csv)

str(nifty\_csv)

# as.Date(nifty\_csv$Date, format = "%d-%b-%y") # to change class of Date

**Reading Excel data:**

It is unfortunately difficult to read Excel data into R. The simplest way out would be to save the Excel file as a CSV file and read it using read.csv function.

The xlsx package is one of the powerful R packages to read, write and format Excel files (java should have been installed). It is necessary to add a sheet name or a sheet index to the function read.xlsx(). If you have a bigger data set, you might get better performance when using the read.xlsx2 function. XLConnect is another package which is frequently used for manipulating Microsoft Excel files from within R.

DIY: Read an SPSS file using the foreign package

**#** Reading Excel files

install.packages("xlsx")

library(xlsx)

location2 <- "D:/CMS/R\_2020/nifty\_data\_2019.xlsx" # copy applicable path

nifty\_excel <- read.xlsx(location2, sheetIndex = 1)

**Web Data – Sites with API:**

R has a variety of ways to import data from web sources. Retrieving the data programmatically makes it possible to collect much more of it with much less effort.

Several packages exist that download data directly into R using a website’s application programming interface (API). Example - The quantmod package (Quantitative Financial Modelling Framework) provides access to stock tickers on Yahoo! finance.

**#** Web Data - Sites with API

library(quantmod)

nifty\_yahoo <- getSymbols("^NSEI")

tail(nifty\_yahoo)

chartSeries(nifty\_yahoo$NSEI.Close) # create a quick chart

**Exporting a dataset**

Writing files is generally simpler than reading files. We use the write.table and write.csv functions. Both functions take a data frame and a file path to write to if you want to save the file to somewhere other than the working directory. The functions write.xlsx and write.xlsx2 (from xlsx package) can be used to export data from R to an Excel workbook. Unless you really need to use Excel spreadsheets, you are better off saving your data in a text format for portability.

**#** Writing a file

setwd("D:/CMS/R\_2020") # copy applicable path

df6.participant

write.csv(df6.participant, "df6\_csv")

write.csv(df6.participant, "df6\_csv", row.names = F) # to prevent the extra column

created earlier

write.xlsx(df6.participant, "df6\_excel.xlsx", sheetName = "Sheet1", row.names = F)

View(mtcars)

write.xlsx(mtcars, "mtcars.xlsx", sheetName = "carinfo", row.names = F)

**Data Transformation with dplyr**

The dplyr package helps us to answer questions that we may have about the data.  dplyr package provides a set of tools for efficiently manipulating datasets in R. It provides some great, easy-to-use functions that are very handy when performing exploratory data analysis and manipulation. It provides simple “verbs”, functions that correspond to the most common data manipulation tasks, to help one translate thoughts into code.

Dplyr aims to provide a function for each basic verb of data manipulation:

* filter() to select cases (observations) based on their values.
* arrange() to reorder the cases.
* select() and rename() to select variables based on their names.
* mutate() and transmute() to add new variables that are functions of existing variables.
* summarise() to condense multiple values to a single value.
* sample\_n() and sample\_frac() to take random samples.

dplyr provides the ability to chain operations together from left to right with the pipe (%>%) operator. It is very useful when you are performing several operations on data, and don’t want to save the output at each intermediate step.

library(dplyr)

# load the placement data file

location3 <- "D:/CMS/R\_2020/Placement\_Data\_Class.csv" # copy applicable path

placement <- read.csv(location3)

class(placement)

View(placement)

colnames(placement)

head(placement)

str(placement)

levels(placement$hsc\_s)

levels(placement$degree\_t)

levels(placement$degree\_t) <- c("CM", "Ot", "ST")

str(placement$degree\_t)

# pick observations by their values using filter()

filter(placement, degree\_t == "CM",

degree\_p >= 80)

filter(placement, degree\_t == "CM",

degree\_p >= 80,

gender != "M")

filter(placement, degree\_t == "CM",

degree\_p >= 80 | degree\_p <= 55)

filter(placement, degree\_t == "CM",

degree\_p >= 80 | degree\_p <= 55,

gender != "F")

# select a small subset of data for ease of understanding

p\_xtreme <- filter(placement, degree\_p >= 85 | degree\_p <= 52)

p\_xtreme

# reorder the rows using arrange()

arrange(p\_xtreme, etest\_p)

arrange(p\_xtreme, desc(ssc\_p))

arrange(p\_xtreme, gender, desc(ssc\_p))

# pick variables (columns) by their names using select()

select(p\_xtreme, ssc\_p, hsc\_p, degree\_p)

select(p\_xtreme, ssc\_p:degree\_p)

select(p\_xtreme,-(ssc\_p:degree\_p))

select(p\_xtreme, ends\_with("\_p"))

select(p\_xtreme, starts\_with("h"))

select(p\_xtreme, contains("\_")) # there are other helper functions also

rename(p\_xtreme, gen = gender) # to rename varaibles

select(p\_xtreme, 1:3)

# create new varaibles which are functions of existing varaibles using mutuate()

mutate(p\_xtreme, avg\_p = (ssc\_p + hsc\_p + degree\_p)/3)

transmute(p\_xtreme, avg\_P = (ssc\_p + hsc\_p + degree\_p)/3)

# Group summary using summarize()

summarise(p\_xtreme, mean(degree\_p))

summarise(group\_by(p\_xtreme, degree\_t), mean(degree\_p))

# select rows randomly

sample\_n(placement, size = 10)

set.seed(100) # seed helps to get the same set

sample\_n(placement, size = 10, seed = 100)

sample\_frac(placement, size = 0.10)

# combining multiple operations with the Pipe

placement %>% group\_by(degree\_t) %>% summarise(count = n()) # read %>% as 'then'. Use shortcut

placement %>% group\_by(degree\_t) %>%

summarise(mean(degree\_p))

placement %>% group\_by(degree\_t, gender) %>%

summarise(mean(degree\_p))

placement %>% filter(status == "Placed") %>%

group\_by(degree\_t, workex) %>% summarise(mean(degree\_p))

placement %>% group\_by(status) %>% summarise(count = n(), mean(degree\_p))

placement %>% filter(status == "Not Placed") %>%

group\_by(degree\_t) %>% summarise(count = n())