FDP – R for Data Science

Session 2: Data Management and Manipulation

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
average_marks
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")
participant_sport
sport_factor <- as.factor(participant_sport)
sport_factor # did you notice the levels
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(1:6, nrow = 3)
mat_A
mat_B <- matrix(1:6, ncol = 3)
mat_B
mat_C <- matrix(11:16, 3)
mat_C
mat_D <- matrix(11:16, , 3)
mat_D
add_mat <- mat_A + mat_C
add_mat
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 %*% mat_C
mat_A %*% t(mat_C) # t is used for transpose
colnames(mat_A) <- c("col1", "col2")
rownames(mat_A) <- c("row1", "row2", "row3")
print(mat_A)
mat_B <- matrix(1:6, ncol=3,
              dimnames = list(c("ROW1", "ROW2"),
              c("COL1", "COL2", "COL3")))
mat_B
mat_A %*% mat_B # observe row and col names
# Compare matrix C and E
mat_E <- matrix(11:16,3, byrow = TRUE)
mat_E
mat_C
# To check the dimensions of a matrix
dim(mat_E)
nrow(mat_E)
ncol(mat_E)
length(mat_E)
dim(mat_E) <- c(2, 3) # to reshape a matrix
mat_E
# Extract elements of a matrix
mat_E[1, 2]
mat_E[1, ]
mat_E[ ,2:3]
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]
anyNA(df1.participant)
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, total_marks)
df5.partcipant
merge(df3.participant, df5.partcipant)
merge(df3.participant, df5.partcipant, by = "register_no")
merge(df3.participant, df5.partcipant, by = c("register_no", "total_marks"))
```

```
# 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(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.

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 <- "C:/Users/Admin/Desktop/FDP_R/Placement_Data_FDP.csv" # copy applicable path
placement <- read.csv(location3, stringsAsFactors = T)
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, 1:3)
select(p_xtreme,-(ssc_p:degree_p)) # can be used to drop few columns
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 variables

# create new varaibles which are functions of existing variables
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(placement, mean(degree_p))
summarize(group_by(placement, degree_t), mean(degree_p)) # s or z will do

# select rows randomly
sample_n(placement, size = 10)
set.seed(100) # seed helps to get the same set
sample_n(placement, size = 10)
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(specialisation, workex) %>% summarise(mean(mba_p))
placement %>% group_by(status) %>% summarise(tally = n(), mean(mba_p)) #tally or count works
placement %>% filter(status == "Not Placed") %>%
  group_by(degree_t) %>% summarise(count = n())
```