Week 1 Session 3

Hussain Alsalman

April 29, 2019

```
## here() starts at /home/hussain/Documents/ArabianAnalyst/summerCourse/Course
```

Foundations of R Programming

What is R?

R is a free language and environment for statistical computing and graphics. You can perform a variety of tasks using R language. Some are as follows

- Exploring and Manipulating Data
- Building and validating predictive models
- Applying machine learning and text mining algorithms
- Creating visually appealing graphs
- Building online dynamic reports or dashboards

Command Line Interface for R.

- R is an engine that can be run through CLI.
- R comes with simple GUI but it is very premitive.
- You can run multiple instances of R with no problem.

Let's learn how to

- Set our working directory
- Save our workspace
- Load our workspace
- Display existing objects
- Remove objects
- Save our history

R Basics

Let's learn how to

- Basic Math

```r
1 + 1
```

```
## [1] 2
```

```r
3 * 7 * 2
```

```
## [1] 42
```

```r
(4 * 6) + 5
```

```
## [1] 29
```

- Variables

There are number of ways to assign variables. R most convientional way might look odd for you but trust me you will get used to it very quickly.

```
# if you are using RStudio you can use alt + "-" to automatically craete "<-" assignment operator
a <- 100
a
```

```
## [1] 100
```

We still can use the "=" operator. But it is really not very common the R community. It was even suggested in the Google R Style Guide. But again they suggested to use dots instead of underscores which I really don't like so it is a choice preference.

```
a = 300
a
```

```
## [1] 300
```

We can combine assignments of variables in one line. I really don't recommend it but you can do it.

```
a <- b <- 199
sprintf("a : %i b : %i", a,b)
```

```
## [1] "a : 199 b : 199"
```

Another way to assign variables is using the assign function. The labourios way of assigning variables.

```
assign("a", 348)
a
```

```
## [1] 348
```

- Data Types

The four main types of data most likely to be used are numeric , character (string), Date / POSIXct (time-based) and logical (TRUE / FALSE).

Numeric type (most flexible)

```
x <- 10
class(x)
```

```
## [1] "numeric"
```

```
is.numeric(x)
```

```
## [1] TRUE
```

As you all know that numbers can be `integers` or `doubles`. Sometimes we want to ensure that R knows that the value is integer. We can do that with adding L at the end of the number

```
i <- 18L
class(i)
```

```
## [1] "integer"
```

```
is.numeric(i)
```

```
## [1] TRUE
```

As in other modern programming languages doing operations on numbers will esclate them to the more flexible type.

```r
class(58L * 10)
```

```
## [1] "numeric"
```

```r
class (5L / 2L)
```

```
## [1] "numeric"
```

Warning

if you don't specify the number with L it will be non-integer and might cause problems when you do comparisions.

```r
 class(4L- 1)
```

```
## [1] "numeric"
```

```r
identical(4L, 4)
```

```
## [1] FALSE
```

Character type (most flexible)

R has two ways of dealing with Characters; as `character` or `factor`. factors play important role when we do modeling.

```r
x <- "data"
class(x)
```

```
## [1] "character"
```

```r
x
```

```
## [1] "data"
```

```r
y <- factor("data")
class(y)
```

```
## [1] "factor"
```

```r
y
```

```
## [1] data
## Levels: data
```

we can find the length of the character by using `nchar`. It is important to know that this will not work in `factor` types.

```r
x <- "hello"
nchar(x)
```

```
## [1] 5
```

```r
nchar(194)
```

```
## [1] 3
```

```r
y <- factor("factor")
nchar(y)
#Error in nchar(y) : 'nchar()' requires a character vector
```

Dates

Dealing with dates in R can be very challenging because R has too many flavors of `date` types. The most useful and common are `Date` and `POSIXct.Date`. `Date` stores only dates but `POSIXct.Date` stores dates and time. Both objects are actually represented as the number of days (`Date`) or seconds (`POSIXct`) since January 1, 1970

```
date1 <- as.Date ("2019-06-1")
class(date1)
```

```
## [1] "Date"
```

```
as.integer(date1)
```

```
## [1] 18048
```

```
date2 <- as.POSIXct ("2019-06-1 17:42")
class(date2)
```

```
## [1] "POSIXct" "POSIXt"
```

```
as.integer(date2)
```

```
## [1] 1559400120
```

luckily we have `lubridate` package that we will discuss later in the course that makes it easy to deal with dates.

Logical

Logical values in R are either `FALSE` or `TRUE`. Notice that they are both capital letters. In reality R represents `FALSE` as `0` and `TRUE` as `1`. We can verify that by the following commands

```
5 * TRUE
```

```
## [1] 5
```

```
8 * FALSE
```

```
## [1] 0
```

```
TRUE + FALSE + TRUE + TRUE
```

```
## [1] 3
```

```
class(TRUE)
```

```
## [1] "logical"
```

Any logical operations will be evaluted as logical value

```
44 == 43
```

```
## [1] FALSE
```

```
44 > 43
```

```
## [1] TRUE
```

```
43 != 49
```

```
## [1] TRUE
```

- Vectors Vectors are very special type in R. Think of them as arrays but only one dimension. Here is an example of how we create vectors.

```r
x <- c (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
class(x)
```

```
## [1] "numeric"
```

we can do operations so easily with vecotrs because R is a vectorized language

```r
x * 10
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

There is another way to create vectors especially if we want to create a squence of numbers.

```r
1:7
```

```
## [1] 1 2 3 4 5 6 7
```

we can start from any number. we can even reverse the order

```r
-5:0
```

```
## [1] -5 -4 -3 -2 -1  0
```

```r
10:0
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1  0
```

we can do vector to vector operations so easily. Notice that vectors need to be the same size, otherwise R will recycle the shorter vector

```r
a <- 1:10
b <- 10:1
a-b
```

```
##  [1] -9 -7 -5 -3 -1  1  3  5  7  9
```

```r
d <- 1:10
f <- c(0,5)
d + f
```

```
##  [1]  1  7  3  9  5 11  7 13  9 15
```

logical operations are also possible on vectors

```r
d > 5
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

we can name elements in the vector. Notice that we did not use `""` for the names.

```r
a <- c(Hussain = 3, Mona = 5, Haya = 7)
a
```

```
## Hussain    Mona    Haya
##       3       5       7
```

- Calling Functions

function calling is similiar than most languages. you provide the function name followed by paranthesis

```r
mean(a)
```

```
## [1] 5
```

- Function Documentation There are many ways to get help to know more about any fucntion. for now we will learn only about `?` and `??` and later I will show you how to get more help.

```
?mean()
```

you use the `aprops` function if you are not sure about the function name exactly

```
apropos("mea")
```

```
##  [1] "colMeans"           ".colMeans"          "influence.measures"
##  [4] "kmeans"             "mean"               "mean.Date"
##  [7] "mean.default"       "mean.difftime"      "mean.POSIXct"
## [10] "mean.POSIXlt"       "rowMeans"           ".rowMeans"
## [13] "weighted.mean"
```

- Missing Data

R represents missing data as `NA` and we can check a vector contains a missing data. This is very helpful as we will see later on in the course

```
missing_data <- c(1, 59, NA, 34, NA, 12)
is.na(missing_data)
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

- Pipes

Pipes are very awesome feature in R. Although it does not come with R naturally but it is only a small package you can use called `magrittr`. Notice, things may start to look weird if you want to apply mathimatical operations. Also notice that some packages like `ggplot2` use different type of pipes

```
library("magrittr")
x <- 1:100

x %>% sum()
```

```
## [1] 5050
```

```
x %>% sum() %>% `/`(.,2)
```

```
## [1] 2525
```

We will use pipes a lot later on the course.

R beyond basics

R beyond basics

- Other data Types (Data.Frames, Lists, Matrices, Arrays)

There other types of data that R is supporting like `Data.Frames` , `Lists` , `Matrecies` and `Arrays`. Each one has its own use and we will go through them all quickly to give you a flavor of what they do.

Data Frames

Data Frames are just like excel sheets. It as rows and columns and each column has its own type

```
x <- c("apple","orange", "grapes", "Kiwi", "lemon")

y <- c( 10L, 40L, 23L, 12L, 50L)

# if we change one of the values to character the whole vector will be changed to character as expected
```

```r
#y <- c( 10L, "40L", 23L, 12L, 50L)

df <- data.frame(x,y, stringsAsFactors = FALSE)
df
```

```
##        x  y
## 1  apple 10
## 2 orange 40
## 3 grapes 23
## 4   Kiwi 12
## 5  lemon 50
```

we can change the name of the columns if we like using the `names` function. which by default will change the column names. we can add row names if we like.

```r
names(df) <- c("fruits", "Quantity")
df
```

```
##   fruits Quantity
## 1  apple       10
## 2 orange       40
## 3 grapes       23
## 4   Kiwi       12
## 5  lemon       50
```

```r
rownames(df) <- letters[1:5]
df
```

```
##   fruits Quantity
## a  apple       10
## b orange       40
## c grapes       23
## d   Kiwi       12
## e  lemon       50
```

we can get some poperties of about the data frame such as the number of columns and rows. Also the dimension of the DF

```r
ncol(df)
```

```
## [1] 2
```

```r
nrow(df)
```

```
## [1] 5
```

```r
dim(df)
```

```
## [1] 5 2
```

We can see the top of the data frame using the `head` and `tail` functions. This is very useful when we have big data and we want to see only a small sample of it.

```r
head(df, n = 4) # we can use the top of the df
```

```
##   fruits Quantity
## a  apple       10
## b orange       40
## c grapes       23
## d   Kiwi       12
```

```r
tail(df, n=3) # we can use the bottom of the df
```

```
##   fruits Quantity
## c grapes       23
## d   Kiwi       12
## e  lemon       50
```

We can access any column using `$` sign.

```r
df$fruits
```

```
## [1] "apple"  "orange" "grapes" "Kiwi"   "lemon"
```

or using the subscripts square brakets `[]` .

```r
df[1:2,]
```

```
##   fruits Quantity
## a  apple       10
## b orange       40
```

```r
df[,2]
```

```
## [1] 10 40 23 12 50
```

lists

Often a container is needed to hold arbitrary objects of either the same type or varying types. R accomplishes this through lists. They store any number of items of any type. A list can contain all numerics or character s or a mix of the two or data.frames or, recursively, other lists.

```r
list(1,2,3)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

```r
list(c(1,2,3))
```

```
## [[1]]
## [1] 1 2 3
```

we can combine practically and data type into a list

```r
lsts <- list(df, 1:10)
```

```r
lsts[[1]]
```

```
##   fruits Quantity
## a  apple       10
## b orange       40
## c grapes       23
## d   Kiwi       12
## e  lemon       50
```

```r
# what is the difference between [[]] and [] for lists
# [] keeps out put as a list while [[]] simplify it to what is inside the list
```

again we can use `names` function to keep life easier for us

```r
names(lsts) <- c("dataframe","vector")
lsts$vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Matrices

Matricies are similar to data frames except that the whole rows and columns need to be of the sames type.

```r
as.matrix(df)
```

```
##   fruits   Quantity
## a "apple"  "10"
## b "orange" "40"
## c "grapes" "23"
## d "Kiwi"   "12"
## e "lemon"  "50"
```

Arrays

are multidimensional vectors. We rarely use them but they are useful to do complex mathimatics. Simply to narray in `NumPy` module in Python

```r
a <- array(1:20, dim = c(2,5,2))
a
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
##
## , , 2
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   13   15   17   19
## [2,]   12   14   16   18   20
```

```r
a[1,1,2]
```

```
## [1] 11
```

- Writing Functions

Functions in R

Functions are really straight forward in R. All you need to know is the syntax

```r
greetings <- function(name) {

  return(paste("Hello ", name))
```

```
}

greetings(name = "Hussain")
```

```
## [1] "Hello  Hussain"
```

```
# or we can provide arguments positionally
greetings("Hussain")
```

```
## [1] "Hello  Hussain"
```

We can specify default values to function to we can ensure that the function does not fail if no arguments specified.

```
greetings <- function(name= "awesome person") {

  return(paste("Hello ", name))
}

greetings()
```

```
## [1] "Hello  awesome person"
```

Extra arguments

There is a cool feature in R that allows you to provide extra arguments with out the need to repeat everything. This is a slightly complicated example but I hope you see the idea behind it.

```
spell_name <- function(name,...) {
  argument <-list(...)
  if(length(argument) > 0 & !is.null(argument$check) & is.logical(argument$check)) {
    check <- argument$check
  if(check == TRUE){
   spelling <- substr(name, nchar(name)-2, nchar(name))
  return(paste0(". Just to make sure, last ", min(nchar(name),3) , " letters of  your name ",ifelse(ncha
  }
  return(NULL)
}


greetings <- function(name= "awesome person",...) {

  return(paste0("Hello ", name, spell_name(name,...)))
}

greetings(name = "s", check = TRUE)
```

```
## [1] "Hello s. Just to make sure, last 1 letters of  your name is S"
```

- Control Statements & Loops

Control Statements

If you have programming background this should be really easy for you. We will go through the syntax really quickly because it is a straight forward proces.

```r
a <- TRUE
if (a == TRUE)
{
 print(a)
}
```

[1] TRUE

We can use `else` execute statemets in case the condition is not met.

```r
if(a != TRUE) {
  print(a)
} else {
  print(!a)
}
```

```
## [1] FALSE
```

We also can use `else if` to put another condition

```r
#a <- 34
if( a == 3) {
  print("a is 3")
} else if (a == 1){
  print("a is 1")
}else {
  print("not sure what a is")
}
```

```
## [1] "a is 1"
```

There is more concise way to do if statements if you have only one condition using the `ifelse`

```r
ifelse(1 == 3, TRUE, FALSE )
```

```
## [1] FALSE
```

Loops

Same thing with if statements. loops are very similar to other languages. There is nothing so special about loops, except for vectorized loops

```r
for (i in 1:4) print(i)
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

Vectorized operation vs loops

```r
fruits <- c("Apple", "Orange", "Grapes")
wrd_len <- c(NA,NA,NA)
names(wrd_len) <- fruits
for (i in 1:length(fruits)){
  wrd_len[i] <- nchar(fruits[i])
}
wrd_len
```

```
##  Apple Orange Grapes
##      5      6      6
```

This is vectorized Operation

```
wrd_len <- c(NA,NA,NA)
wrd_len <- nchar(fruits)
names(wrd_len) <- fruits
wrd_len
```

```
##  Apple Orange Grapes
##      5      6      6
```

- Reading Data into R
  - CSV, Excel ## Rading CSV

There are couple of functions to read data we will focus on the `read.csv` because it is really common. Notice, `read.csv` converts strings into factors by default. In most cases this might not be what you want. You can change this behavior by setting `stringsAsFactors` to `FALSE` . Notice, R always save the output of this function as a data frame. There are many options we will explor some of them here.

```
read.csv(file = "../datasets/Read_Data.csv", stringsAsFactors = FALSE, colClasses = c("character", "inte
```

```
##    Country Age Salary Purchased
## 1   France  44  72000        No
## 2    Spain  27  48000       Yes
## 3  Germany  30  54000        No
## 4    Spain  38  61000        No
## 5  Germany  40     NA       Yes
## 6   France  35  58000       Yes
## 7    Spain  NA  52000        No
## 8   France  48  79000       Yes
## 9  Germany  50  83000        No
## 10  France  37  67000       Yes
```

Reading Excel

In order to read Excel files we will need a package called `readxl`. It as a function called `read_excel` that is very useful. Notice, this package return a the data with class type `tibble`. You don't have to worry about it now as it acts just like data frame.

```
library("readxl")
file_path <- "../datasets/Read_Data.xlsx"

df <-read_excel(path = file_path)
```

```
class(df)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

- R Binary data

Reading from R binary files.

We can save our work in binary and load it back to our envirnoment using `save` and `load` functions

```
save(list=ls(),file = "../datasets/all_data.RData")
rm(list = ls())
```

```r
load("../datasets/all_data.RData")
```

- Graphing in R
  - Base Graphs
  - ggplot2