Hands-On Exercise 1

Elena Dubova

2/22/2020

Let's go!

Dear R enthusiast, warm welcome! This is the first Hands-On Exercise file in our course **Big Data, Machine Learning and their Real World Applications**. Upon completion of the exercise you will be able to:

- run R code in RStudio
- assign values to variables and do operations using these variables
- generate a random dataset
- slice and subset a dataset.

For now, I just want to welcome you once again from the console. Find the green triangular 'Run' button in the grey chunk of code below and see my greeting in the console.

```
my_string <- 'Welcome to the world of R!'
my_string
```

```
## [1] "Welcome to the world of R!"
```

Creating R objects and manipulating them.

Objects in R are created with *assignment operator* that consists of an angled bracket and a minus sign: **<-**.

Good to know!: Sometimes you will see people use = sign instead of **<-**. In many cases you can use either and your code will work just fine. However, there is a subtle difference between them. To avoid ambiguity and make your code easy to read, **Google's R Style Guide** recommends using **<-** to assign a value to an object.

Well, let's create some objects. Note that when we create an object, a certain amount of memory is allocated to store the object. If you assign a different value to the existing object, the previous object is erased.

```
x <- 25
y <- 12
x; y
```

```
## [1] 25
```

```
## [1] 12
```

```
y <- 11
x; y
```

```
## [1] 25
```

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

We just assigned a number to objects called **x** and **y**. We can also assign a result of some mathematical operation or output of a function to an object. Also, we can use existing objects in memory to create new objects.

```
a <- 25 / 5 - 1
a
```

```
## [1] 4
```

```
b.1 <- sum(25, 11)
b.1
```

```
## [1] 36
```

```
b.2 <- sum(x, y)
b.2
```

```
## [1] 36
```

Can you see why b.1 object stores the same value as b.2 object?

Let's now see what operators we can use in R. There are (1) **arithmetic**, (2) **comparison**, and (3) **logical** operators.

(1) **arithmetic**:

- + addition
- - subtraction
- * multiplication
- / division
- ^ power
- %% modulo
- %/% integer division

```
25*25
```

```
## [1] 625
```

```
x*x
```

```
## [1] 625
```

```
x^2
```

```
## [1] 625
```

Can you see why three expressions produce similar result?

```
x%%y
```

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

```
x%/%y
```

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

Can you explain what is going on here? Can you think of a practical situation, when modulo or integer division come handy?

- do something every nth time
- split seconds into hours, minutes and seconds
- determine if the number is even

My example: every 5th time customer makes a purchase, give him a discount coupon

```
customer_purchases <- 1:10

for (purchase in customer_purchases) {
  if (purchase %% 5 == 0) {
      print('Have a nice day! Here is your coupon!')
  }
  else {
    print('Have a nice day!')
  }
}
```

```
## [1] "Have a nice day!"
## [1] "Have a nice day!"
## [1] "Have a nice day!"
## [1] "Have a nice day!"
## [1] "Have a nice day! Here is your coupon!"
## [1] "Have a nice day!"
## [1] "Have a nice day!"
## [1] "Have a nice day!"
## [1] "Have a nice day!"
## [1] "Have a nice day! Here is your coupon!"
```

(2) **comparison**: compare values and return **TRUE** or **FALSE**

- < lesser than
- > greater than
- <=* lesser than or equal to
- >= lesser than or equal to
- == equal
- != not equal

```
x>y
```

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

```
x<y
```

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

```
x!=y
```

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

```
x==y
```

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

```
25*25 == x*x
```

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

```
x*x == x^2
```

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

How is this useful? Can it apply to a real business scenario? First, you can check your code. As you code more, it comes very handy.

```
seconds_total <- 5000
hours <- seconds_total %/% 3600
munutes <- seconds_total %% 3600 %/% 60
```

```
seconds <- seconds_total %% 3600 %% 60

seconds_total == hours*3600 + munutes*60 + seconds
```

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

Second, you can use conditions to implement a task that is important for, say, marketing campaign. Or compliance with law regulations.

```
library(lubridate)
```

```
##
## Attaching package: 'lubridate'
```

```
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
```

```
customer_birthday <- c(day(as.Date('03/30/2000','%m/%d/%Y')),
                       month(as.Date('03/30/2000','%m/%d/%Y')))

  if (day(today()) == customer_birthday[1] && month(today()) == customer_birthday[2]) {
      print('Happy birthday! Here is your 10% discount in our restaurant!')
  } else {
    print("Have a nice day!")
  }
```

```
## [1] "Have a nice day!"
```

```
age <- 18

  if (age >= 18) {
      print('You can get a driver license!')
  } else {
    print('Sorry, to drive a car you should be at least 18 years old.')
  }
```

```
## [1] "You can get a driver license!"
```

What is your example, when comparison operators are of practical value?

(2) **logical**: combine several values or **TRUE/FALSE** expressions (**boolean** expressions) and return a single **TRUE** or **FALSE** output

- **!x** logical OR
- **x & y** logical AND
- **x && y** id
- **x | y** logical OR
- **x || y** id
- **xor(x,y)** exclusive OR

```
x <- FALSE
y <- 1
!x
```

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

```
x & y
```

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

```
x | y
```

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

```
xor(x,y)
```

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

Now let's see what types of values we can assign to an object. R supports the following types:

- **integer**
- **numeric**
- **character**
- **logical** Note: in R code you can see T/F instead to TRUE/FALSE - both ways work.
- **factor**

```
my.integer <- 5L

class(my.integer)
```

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

```
my.numeric <- 5.0

class(my.numeric)
```

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

```
my.string <- '5.0'

class(my.string)
```

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

```
my.logical <- TRUE

class(my.logical)
```

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

```
my.factor <- as.factor(my.string)

class(my.factor)
```

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

So far, our objects store only single values. Let us see more complex objects:

- **vector**: contains elements of one type
- **list**: contains elements of mixed types, Note, we can use **$** sign to access elements of the list
- **matrix**: 2-dimensional vector, it has columns and rows
- **data frame**: 2-dimensional data table that allows vectors of mixed types; list of vectors of same length.
- **array**: multidimensional data structures of elements of one type

```
my_vector <- c(1, 2, 3, 4, 5)
my_other_vector <- 1:5


my_list <- list(1, 'Mike', 'A', T)
my_other_list <- list(id = 1, name = 'Mike', grade = as.factor('A'), soccer_team = T)
my_other_list$name
```

```
## [1] "Mike"
```

```
my_matrix <- matrix(c(1,2,3,4), nrow = 2, ncol = 2)
my_other_matrix <- matrix(c(1,2,3,4), nrow = 2, ncol = 2, byrow = T)
my_matrix
```

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

```
my_other_matrix
```

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

Can you see the difference? What **byrow** argument is doing?

```
id <- c(1, 2, 3, 4)
name <- c("Mike", "John", "Judy", "Kate")
grade <- as.factor(c('A', 'B', 'A', 'B-'))
soccer_team <- c(TRUE, TRUE, TRUE, FALSE)
my_data_frame <- data.frame(id, name, grade, soccer_team)
my_data_frame
```

```
##   id name grade soccer_team
## 1  1 Mike     A        TRUE
## 2  2 John     B        TRUE
## 3  3 Judy     A        TRUE
## 4  4 Kate    B-       FALSE
```

We are going to work primarily with data frames in this course.

This is the second Hands-On Exercise file in our course **Big Data, Machine Learning and their Real World Applications**. Upon completion of the exercise you will be able to: - get the idea of the role of functions in R. - import datasets into R - slice and subset a dataset - apply functions to manipulate data

##Introduction to functions. We just learnt a lot about data types and basic operations on them. In Data Science we oftentimes need to quickly do more sophisticated operations involving a lot of basic operations. This is when functions do the tedious job for us. They automate an operation, consisting of a number of sequential operations. You can write your own function, but in a lot of cases you find that somebody already wrote a function and you can just reuse it! This is where **packages**, or **libraries** come into play. A package is a collection of functions, united by some common theme. R has especially rich assortment of packages and thanks to an open source community you can benefit froms packages people publish around the world.

So, let's see the function structure:

my.result <- my.function(argument.1 = value/object, argument.2 = value/object)

Some functions are in the **base** package and you can just use them right away.

```
my.result <- mean(x = c(1,2,3))
my.result
```

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

Others require you to download a package and call a library first.

```
now()
```

```
## [1] "2020-06-24 15:20:55 EDT"
```

```r
#install.packages('lubridate')
library(lubridate)
now()
```

```
## [1] "2020-06-24 15:20:55 EDT"
```

Some of the functions will stick to you naturally, but do not try to memorize anything! Just practice. It is ok not to have syntax on top of your mind. Use help, cheat sheets, search CRAN library, or just Gooooooogle it! Stack Overflow is also your best friend!

Congrats!

This is the end of our introduction to R.

References

1. Federico Hathoum. Practical uses for the modulo operator. https://hatoum.com/blog/2012/12/practical-uses-for-modulo-operator.html