2. Starting with R

Ali Ataullah and Hang Le

01/04/2020

Contents

1 Introduction

There are several excellent (many free) resources available to learn R. For beginners, we recommend Peng (2016) and Wickham and Grolemund (2016). The webpage for bookdown, an R package for writing books, has several free textbooks for learning R (including the two mentioned earlier). We also recommend on-line sources, such as excellent courses by academics from Johns Hopkins University and Datacamp.

2 Intended Learning Outcomes

By the end of this session, students should be able to

1. understand different types of objects in R;

2. use R to perform simple computations; and

3. use functions in R.

3 R and RStudio Set-up

To follow these sessions, you will need to install R from CRAN and RStudio. A useful guide for installation is available at RStudio. RStudio is an excellent *Integrated Development Environment* that will enable you to efficiently manage your research projects (and do several other things such creating webpages). We will only focus on basic financial data analysis in these sessions.

4 R Packages as Bags of Tools

Like many other statistical software, R has several *libraries* or *packages*. These packages contain functions that enable users to perform a variety of tasks. For example, as we will see shortly, the dplyr package, the most widely used package according to Revolution Analytics, contains several very useful functions that enable users to clean and transform data. We will learn some of these functions in the next session.

For our sessions, we need the tidyverse package, which contains several packages, including dplyr. Once installed, we load the package tidyverse as follows.

```r
library(tidyverse)
```

5 Basic Arithmetic in R

5.1 Examples

1. Addition and division

```r
2+2
```

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

```
6/3
```

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

2. Remainder and exponentiation

```
5 %% 2
```

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

```
2^3
```

```
## [1] 8
```

3. We can create objects based on our computations using `<-` as follows.

```
var1 <- 5**2
```

`var1` is available in our `R` environment.

```
var1
```

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

```
print(var1)
```

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

5.2 Exercise

1. Subtract 4099 from 3098.

```
# Your code
```

2. Multiply 59 with 76.

```
# Your code
```

3. What is 9 raised to power 4?

```
# Your code
```

4. An investor bought 100 shares of Dorian PLC. on January 1, 2019 for price $P_{t-1} = \$30$ per share. The

investor sold all her shares on January 1, 2020 for $P_t = \$33$ per share. Create a variable called `return` and assign to it the annual net return for the investor. The net return R_t is computed as follows:

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1 \tag{1}$$

```
# Your code
```

Later on, we will use gross return, which is simply

$$1 + R_t = \frac{P_t}{P_{t-1}} \tag{2}$$

It is important to note that the return on asset is given per unit of time. In the above example, the net return is 10% per annum.

We can clear our `R` environment either by removing names objects (e.g. `rm(var1)`) or all objects like this:

```
rm(list = ls())
```

6 Functions in R

There are thousands of functions available in `R` to perform a variety of tasks. We can also write our own functions. We begin with simple examples.

6.1 Examples

1. Taking log to the base 10.

```
log(1000, base = 10)
```

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

2. Rounding decimals.

```
round(1.7617, digits = 2)
```

```
## [1] 1.76
```

3. Counting the number of characters in a string.

```
nchar("University of Bologna")
```

```
## [1] 21
```

4. As noted earlier, we can create objects and assign them names using `<-` symbol (less than sign followed by minus sign). For example, let us create an object `gross_rt` and assign it the value of gross return that we computed above.

```
gross_rt <-  (33/30)
```

5. We can now use the object `gross_rt` in our computations. For example, if the investor invested $\$300,000$ in Dorian PLC last year. Then, her pay-off after one year given the gross return $(1 + R_t) = 1.1$ will be

```
300000 * gross_rt
```

```
## [1] 330000
```

6. Note that the object `gross_rt` is stored in your `R` environment. We can remove it by using the `rm()` function as follows.

```
rm(gross_rt)
```

7. Compute the square roots of 4,16, and 32. Note how we concatenate the three numbers 4,16, and 32 using `c()`.

```
sqrt(c(4,16,37))
```

```
## [1] 2.000000 4.000000 6.082763
```

6.2 Exercises

1. Take log (base 2) of 64.

```r
# Your code
```

2. Take log (base e) of 64. This is called the natural log. To take natural log, you do not need to specify base.

```r
# Your code
```

3. In finance, we frequently work with **log return**, which is $log(1 + R_t)$. Compute the log return for the investor who bought 100 shares of Dorian PLC. Assign it to `log_return`. Compare log return to net return computed earlier.

```r
# Your code
```

7 Objects in R

There are 5 types of objects in R. These are: (1) numeric (or double), (2) integer, (3) complex, (4) character and (5) logical. There is another type called 'raw' that will not use here.

7.1 Examples

1. We now use the function `typeoff()` to check types of several objects.

```r
typeof(1)
```

```
## [1] "double"
```

```r
typeof(as.integer(1))
```

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

```r
typeof(1+1i)
```

```
## [1] "complex"
```

7.2 Exercise

1. Determine the types of "Bologna".

```
# Your code
```

2. Determine the type of `TRUE`.

```
# Your code.
```

8 Relations Between Objects

8.1 Examples

1. We can test whether a particular relation between two objects hold. The outcome would be a logical, which can be either `TRUE` or `FALSE`. For example,

```
2 < 3 # 2 is less than 3
```

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

```
2 <= 3 # 2 is less than or equal to 3
```

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

```
2 >= 3 # 2 is greater than or equal to 3
```

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

```
2 != 1 # 2 is not equal to 1
```

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

2. At times, we need to combine statements. For example, we may need to check whether two statements A and B are both true or whether one of them is true. For such tasks, we need `and (&)` and `or (|)` operators. Some examples are given below.

```
(4 < 3 & 3 == 3) # Check two statments are both true
```

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

```
(2 < 3 | 3 == 3) # Check at least one statment is true
```

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

```
(2 < 3 | 3 == 3 | 7 < 6) # Check at least one statment is true
```

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

8.2 Exercise

1. Guess what the output of the following code will be:

```
((4 < 3 & 3 == 3) | (2 < 3 | 3 == 3))
```

2. Run the above code to check your answer.

```
# Your code
```

9 Conditional Statements

9.1 Examples

1. Suppose we want R to perform a task given that certain condition holds. We can do this by using an `if` statement (or a conditional statement). We will work with conditional statement in detail in the following chapters. But it is good to get familiarised with them at an early stage. A simple example is given below in which we ask R to print `The Statement is True` if a given statement is true.

```
if (2 < 3){
  print("The Statement is True")
}
```

```
## [1] "The Statement is True"
```

2. It is important to remember the structure of the above conditional statement. The first part of the code (i.e. `if(2 < 3)`) tests the truth or falsity of the given statement (in the example above the statement `2 < 3` is true). The second part of the statement is enclosed in curly brackets. This part of the code runs if the preceding statement is true (which is the case in the above example). Let us see an example in which the statement is false.

```r
if (4 < 3){
  print("The Statement is True")
}
```

The above code does not print the statement because the statement `4 < 3` is false.

3. In our examples above, our code provides an output if the given statement is true; no output is produced when the statement is false. But suppose we wish to produce two distinct statements depending on the truth and falsity of our statement. For example, we may wish to print `The Statement is True` when the statement is true and `The Statement is False` when the statement is false. We need to expand our `if` statement by including an `else` as follows:

```r
if (4 < 3){
  print("The Statement is True")
} else {
  print("The Statement is False")
}
```

```
## [1] "The Statement is False"
```

4. We can construct statements with multiple conditions using `if`, `else if` and `else` structure. In the code below, `R` checks each statement recursively. An output associated with the first true statement is produced. Let us work with an example with four conditions.

```r
if (3 > 4) {
  print("First statement is TRUE")
} else if (7 < 4) {
 print("Second statement is TRUE")
} else if (3 < 2) {
  print("Third statement is TRUE")
} else {
  print("All Statements are False")
}
```

```
## [1] "All Statements are False"
```

In the code above, all statements are checked and they are all false. So, the output after `else` is produced. In contrast, in the code below, the first condition is true (i.e. `3 is greater than or equal to 3`) and, therefore, the output associated with this condition is produced. The remaining code is ignored by R.

9.2 Exercise

1. Modify the code below so that the output is "Second statement is TRUE".

```
if (3 > 4) {
  print("First statement is TRUE")
} else if (7 < 4) {
 print("Second statement is TRUE")
} else if (3 < 2) {
  print("Third statement is TRUE")
} else {
  print("All Statements are False")
}
```

```
## [1] "All Statements are False"
```

10 Vectors

Our brief discussion below is based on the material covered in Wickham (2019). This text is available online at Advanced R. Before we begin, note that when we create objects in `R`, we must name them appropriately. Briefly put, names of objects you create must begin with a letter.

10.1 Examples

1. In R, the most basic type of data is an **atomic vector** of dimension 1. Consider the following examples in which we assign the number 5 to `x` and $(1, 2, 3)$ to `y`. Both `x` and `y` are vectors. Note how we use `<-` to assign values to create objects that are then stored in R's environment.

```r
x <- 5 # x in now equal to 5 but nothing is printed.


y <- c(1,2,3,4)


x
```

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

```r
y
```

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

```r
is.vector(x) # To check if x is a vector
```

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

2. We can extract elements from a vector that we are interested in.

```r
y[1] # First element of y
```

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

```r
y[1:2] # Elements 1 to 2 of y
```

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

```r
y[c(1,3)] # First and third element of y
```

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

```r
y[y!=3] # Elements of y that are not 3
```

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

3. Atomic vectors contain elements of same type. In the example below, we try to create a vector containing number 1 and string “portfolio”. `R` automatically converts or coerces the number 1 to be a string object.

```r
z <- c(1, "portfolio")
z
```

```
## [1] "1"         "portfolio"
```

4. We can check whether a vector has missing values, represented by `NA`, using `is.na()` function.

```
x1 <- c(1:3,NA,c(1,2,3))


# Check x_1


x1
```

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

```
is.na(x1)
```

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

5. We can change types of vectors. For example, a vector containing numbers can be converted in a vectors containing characters (strings).

```
x2 <- c(1,2,3)


typeof(x2)
```

```
## [1] "double"
```

```
x2 <- as.character(x2)


typeof(x2)
```

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

```
x2
```

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

10.2 Exercise

1. Suppose on April 3, 2020, the stock price of Apple was $241 per share, of Amazon $1906 per share, of Alphabet $1097 per share, and of Facebook $154 per share. Create a vector `prices` that contains this data.

```
# Your code
```

2. An investor buys 50 shares of Apple, 70 shares of Amazon, 80 shares of Alphabet, and 100 shares of Facebook. Create a vector `shares` that contains these numbers of shares and use it to determine the amount that the investor allocated on each of the above companies.

```
# Your code
# Hint: c(1,2) * c(2,3) is equal to 2,6
```

3. Use the function `sum()` to compute the total amount that the investor allocated on the four companies.

```
# Your code
# Hint: sum(1,2,3) is equal to 6
```

We can now clear our environment.

```
rm(list = ls())
```

11 Matrices

Matrices in `R` can be created using the function `matrix()`. All elements of a matrix must have the same type.

11.1 Examples

1. We construct a 2 x 2 matrix `X` with elements `1,2,3,4`.

```
X <- matrix(data = 1:4, nrow = 2, ncol = 2, byrow = TRUE)

# byrow argument in the above function reads the data into our matrix by rows.

X
```

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

2. Portfolio theory in finance relies heavily on variance-covariance matrices of returns on assets. At a later stage we will learn how to **estimate** variances and covariances of returns. Suppose the variance of return on asset A is 0.5, while the variance of return on asset B is 0.7. The covariance between the two is 0.3. This data can be presented as the following variance-covariance matrix Σ:

$$\Sigma = \begin{bmatrix} 0.5 & 0.3 \\ 0.3 & 0.7 \end{bmatrix}$$

We can store this matrix in `R` as follows.

```
vcovMatrix1 <- matrix(c(0.5,0.3,0.3,0.7), nrow = 2, byrow = TRUE)
vcovMatrix1
```

```
##      [,1] [,2]
## [1,]  0.5  0.3
## [2,]  0.3  0.7
```

3. The determinant of a matrix can be computed using `det()` function as follows.

```
det(vcovMatrix1)
```

```
## [1] 0.26
```

4. The inverse of a matrix, if it exists, can be obtained using `solve()` as follows.

```
solve(vcovMatrix1)
```

```
##           [,1]      [,2]
## [1,]  2.692308 -1.153846
## [2,] -1.153846  1.923077
```

11.2 Exercise

1. Store the following variance-covariance matrix called in `R`. Call it `vcovMatrix2`.

$$\Sigma = \begin{bmatrix} 0.3 & 0.7 & 0.5 \\ 0.7 & 0.2 & 0.1 \\ 0.5 & 0.1 & 0.4 \end{bmatrix}$$

```
# Your code
```

2. Compute the determinant of `vcovMatrix2`.

```
# Your code
```

3. Obtain the determinant of `vcovMatrix2`.

```
# Your code
```

4. Obtain the inverse of `vcovMatrix2`.

```
# Your code
```

5. A matrix `Y` is created for you. Check if its inverse exists. Can you tell why not?

```
Y <- matrix(data = c(1,2,1,2), nrow = 2, byrow = TRUE)


Y
```

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

12 Dataframes

For our data analysis, we will mostly deal with tabular data stored as `dataframes`. These are objects that contains columns and rows of data, where each column may contain different class of data. You can think of a dataframe as a group of vectors. Each vector in a data frame can be of different types.

12.1 Examples

1. Let us create a data frame that has three variables, x, y and z. Each variable is a column vector of length 4. We can create data frame using the function `data.frame()` or `tibble()`. Tibbles are special kind of data frames that are extremely useful for data analysis. For details, see the following online resource on tibbles. Note the use of `set.seed(1234)`. This allows us to replicate our results.

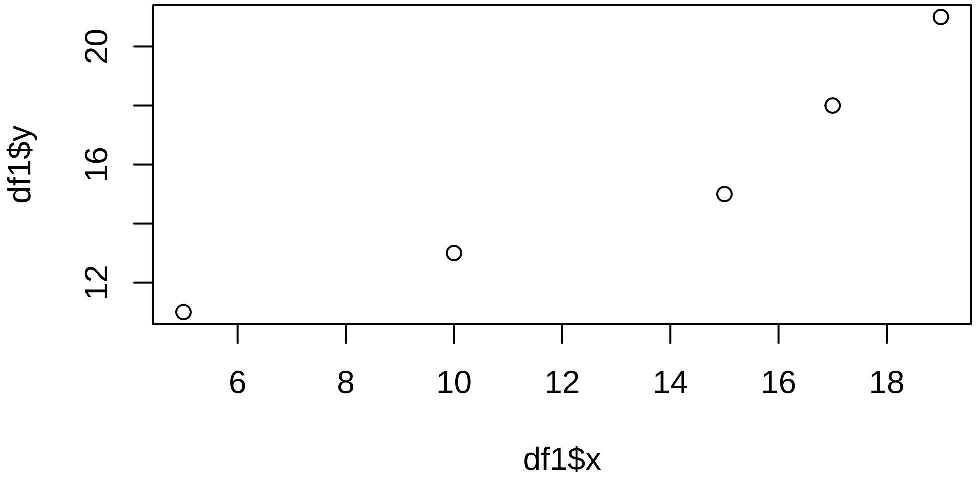
```
set.seed(1234)

df1 <- data.frame(x = c(5,10,15,19,17),
                  y = c(11,13,15,21, 18),
                  z = rnorm(n=5,mean=20,sd=10))

# OR using tibbles
set.seed(1234)
df2 <- tibble(x = c(5,10,15,19,17),
                  y = c(11,13,15,21, 18),
                  z = rnorm(n=5,mean=10,sd=5))

# rnorm() is a function that generate a sample of random variables
# from a normal probability distribution. More on random variables later.
```

2. Let us view the data frame as a table and then make a simple scatter plot. To extract a particular element of our data frame, we use the $ sign. For example, we can obtain the `y` column of our data frame `df` by writing `df\$y`.

```
head(df1)
```

x	y	z
5	11	7.929342
10	13	22.774292
15	15	30.844412
19	21	-3.456977
17	18	24.291247

```
plot(df1$x,df1$y)
```



```
# We will learn basic data visualisation using the excellent ggplot2 package
```

3. In the above example, we created a data frame in R. For your research, you are likely to use data stored in different formats. We now load a file called "first_df.csv" that contains our data in `csv` format. We use the `readr` package, which is included in `tidyverse`.

```
df3 <- read_csv(file="first_df.csv")
```

```
## Parsed with column specification:
## cols(
##   Education = col_double(),
##   Income = col_double()
## )
```

This data contains two variables:

- Education: This is the number of years of education that an individual has;
- Income: Annual salary (in thousands of pounds) for an individual.

4. We can subset a particular part of the dataframe in several ways. Some examples are provided below.

```
df3$Education #Education column of df1
```

```
## [1]  3 15  2  9 16  3 12
```

```
df3[,1] # All rows and firm column of df1
```

Education
3
15
2
9
16
3
12

```
df3[1,2] # First row and second column of df1
```

Income
30000

5. Suppose we want to add a new column containing the age of seven individuals in our data frame `df1`.

   One way to do this is as follows (we will learn how to use `dplyr` in the future):

```
df3$age <- c(35,27,49,51,34,41,29)


df3
```

Education	Income	age
3	30000	35
15	60000	27
2	40000	49
9	50000	51
16	70000	34
3	30000	41
12	90000	29

12.2 Exercise

1. Subset the `Income` column of `df3` in two different ways.

```r
# Your code
```

2. Draw a scatterplot showing `Education` and `Income` contained in `df3`.

```r
# Your code
```

12.3 Saving new data

We can save our dataframe in a variety of formats. Here, we save `df1`, which now contains three variables, in `csv` format using the `write_csv()` package from the `readr` package.

```r
write_csv(df1, file.path( "df3.csv"), col_names = TRUE)


# Data saved in the same directory; you can specify directory.
```

13 Time Series Objects

Many financial analysts use functions that require data to contain a time stamp. A popular package to handle time-series data is `xts`. An `xts` format has a matrix containing the data, along with a column containing the time index.

13.1 Examples

We install and load the `xts` package.

```r
library(xts)
```

Let us create data that along with a time index.

```r
dmatrix <- matrix(data = c(1:10), nrow = 5, ncol = 2, byrow = TRUE)


# Display the matrix created above


dmatrix
```

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

We now append the time index to the above matrix in two steps.

Step 1: Create a time index. The time index must be in an increasing order.

```
index_time <- as.Date(c("2019-01-01","2019-01-02","2019-01-03","2019-01-04","2019-01-05"))
```

Step 2: Append the `index_time` to `dmatrix`.

```
dmatrix <- xts(dmatrix, order.by = index_time)
```

View `dmatrix` and check its structure.

```
str(dmatrix)
```

```
## An 'xts' object on 2019-01-01/2019-01-05 containing:
##   Data: int [1:5, 1:2] 1 3 5 7 9 2 4 6 8 10
##   Indexed by objects of class: [Date] TZ: UTC
##   xts Attributes:
##  NULL
```

```
dmatrix
```

```
##            [,1] [,2]
## 2019-01-01    1    2
## 2019-01-02    3    4
## 2019-01-03    5    6
## 2019-01-04    7    8
## 2019-01-05    9   10
```

We can obtain the time index from an `xts` object by using the function `index()` as follows.

```
index(dmatrix)
```

```
## [1] "2019-01-01" "2019-01-02" "2019-01-03" "2019-01-04" "2019-01-05"
```

13.2 Exercise

1. Create a matrix `myMatrix` with 3 rows and 4 columns.

```
# Your code
```

2. Create a time index called `index_time2` from 2020-01-01 to 2020-01-03.

```
# Your code
```

3. Create an `xts` object using `myMatrix` and `index_time2`.

```
# Your code
```

14 Next Steps

This session focused on basics of `R`. We now have some understanding of how `R` can be used for computations. We also know different types of objects (e.g. vectors) in `R`. For financial analysis, we will be mostly working with data frames. Thus, the next session focuses on various functions in the `dplyr` package. These functions will enable us to clean, filter and transform our data in order to get it ready for our analysis.

References

Peng, R. D. (2016). *R Programming for Data Science.* Leanpub.

Wickham, H. (2019). *Advanced R.* CRC press.

Wickham, H. and Grolemund, G. (2016). *R for Data Dcience: Import, Tidy, Transform, Visualize, and Model Data.* O'Reilly Media, Inc.