Introduction to the R-Programming Language

Oliwier Przewlocki
Technologisches Gewerbemuseum
oprzewlocki@student.tgm.ac.at

Contents

1. Introduction to R

This chapter discusses the purpose of R as a language and lays out the different data types and operations within R.

1.1. What is R

R is both a language and an enviornment developed mainly for statistical computing and visualization. It provides a wide range of tools designed for statistical techniques, like time-series analyses, clustering, classical statistical tests, etc. One of its strengths is the ease at which well-designed, publication-quality plots can be produced. It's environment is well suited for effective data handling, managing operations for array calculations, especially matrices, graphically facillitate data analyses & display either on-screen or on hardcopy.

1.2. Arithmetic Operations

The syntax of arithmetic operations is mostly uniform with every other language (`^` is raising to a power, `/` is dividing, `*` is multiplying, etc.), although an exception exists for the modulo operation, which is the reason this section even came to be.

```
3 %% 2 # 1
```

The modulo in R is made out of two percentage signs.

1.3. Variables & Data Types

R uses an arrow symbol to define a variable. The arrow points out of the variable value and into its identifier i.e. name.

```
a.variable <- 100
```

This assigns the value `100` to the variable `a.variable`. The most common way of naming variables is with a dot as a separator. Camel-case (`aVariable`) is also sometimes used, but conventions like snake-case aren't used. You can see your current variables in the "Environment" tab. Let us now go over to the different data types R has to offer.

R has an integer type (**numeric class**), a floating-point type (**numeric class**) and a string type (you can use either single or double quotes) (**character class**) The data type that are syntactically different from C-like languages is the boolean (**logical class**). You can either assign it with spelling either `TRUE` or `FALSE` in all-caps, or using the shorthand notation, `T` or `F`. Make sure they are in all-caps, otherwise R will think it's a function. There also exists a function to return the data type: `class(a.variable)`.

1.4. Vectors

We use the combine function to create a vector

```
vec <- c(1,2,3,4,5)
```

A vector is essentially a one-dimensional array. You can assign other data types to a vector, although they can't be mixed. For example, if you try to insert a character and a numeric, it will convert the numerics to characters.

1.4.1. Naming Vectors

You can assign a label to each element of a vector by combining it with another label vector using the `names()` function:

```
temps <- c(32, 28, 30, 31, 29, 25, 34)
days <- c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
names(temps) <- days
```

This creates a labeled vector `temps` out of the original vector and the days label vector.

1.4.2. Vector Operations

You can add, subtract, multiply (element-by-element) and divide (element-by-element) vectors like standard numeric values.

Additionally, there are special functions to perform operations different from the standard arithmetics. For example `sum()` will sum the vector and `mean()` will calculate the vector's mean, `sd()` will calculate its standard deviation, `max()` finds the maximum element, `min()` finds the minimum element and `prod()` returns the product of the elements, e.g. `[5,6,7]` would return `5*6*7=210`.

```
v1 <- c(1,2,3)
sum(v1) # 6
mean(v1) # 2
sd(v1) # 1
max(v1) # 3
min(v1) # 1
prod(v1) # 6
```

Another feature of R is applying **comparison operations** to vectors, e.g.

```
v1 <- c(1,2,3,4,5)
v1 == 3 # FALSE FALSE TRUE FALSE FALSE
v1 > 1 # FALSE TRUE TRUE TRUE TRUE
```

This allows for selecting specific values out of vectors. Element-by-element comparison of two vectors is also available (`v1 < v2`).

1.4.3. Vector Indexing

Before proceeding, it's very important to underline that **indeces start at 1** in R.

Other than that, accessing each element is straight-forward:

```
v1 <- c(23,44,69,2,53)
v1[1] # 23
v1[3] # 69
# ...
```

Grabbing individual values is possible with the use of a vector inside of the index brackets

```
v1 <- c(23,44,69,2,53)
v1[c(2,4)] # 44 2 - grabs the 2nd and the 4th element
v1[c(1,4,2)] # 23 2 44 - grabs the 1st, 2nd and 4th element
```

If you have assigned labels to your vector, you can also use the labels as accessing parameters just like in a dictionary

```
v1 <- c(23,44,69,2,53)
names(v1) <- c('a', 'b', 'c', 'd', 'e')
v1['b'] # 44
v1[c('c','b','e')] # 69 44 53
```

Slicing in R works in the same way as Python:

```
v1 <- c(23,44,69,2,53)
v1[1:4] # 23 44 69 2 - grabs the 1st, 2nd, 3rd and 4th element
```

As shown above, slicing indeces are inclusive. A useful trick is that you can use slicing to *create* vectors, example:

```r
v1 <- 1:10 # created a vector with numbers from 1 to 10
```

There is a way of combining vectors and comparisons in a way that allows for direct access to the elements, not just boolean values

```r
v1 <- c(23,44,69,2,53)
v1[v1>30] # 44 69 53
```

The `v1>30` returns a logical result (true/false values) that is then passed to the vector itself as a way of telling it which elements to grab (true grabs them, false skips them). Additionally, you can even name these expressions to create a kind of filter.

2. Matrices

This chapter covers the essentials of matrices in R.

2.1. Creating Matrices

A matrix is created by passing a vector to it:

```r
v1 <- 1:10

matrix(v1, nrow = 2)
```

This creates a 2x5 matrix. Notice how we manually set the number of rows and how R automatically adjusted the number of columns. If we wouldn't do that, it would create a matrix with one column and ten rows. By default, the order in which the elements are placed are by column, meaning it first populates the entire first column, then goes on to the next. This causes the first row to be `[1, 3, 5, 7, 9]`. If this is undesired, there's a `byrow` boolean parameter that can be passed which is set to `FALSE` by default.

2.1.1. Matrix Labels

To pass in labels for the row and column labels, you first create two vectors and then use the apropriate functions to spread them out

```r
days <- c('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun')
stocks <- c('Google', 'Meta')

colnames(stock.matrix) <- days
rownames(stock.matrix) <- stocks
```

2.2. Matrix Multiplication

To perform true matrix multiplication, you use an asterix surrounded by percentage signs instead of just the asterix which performs an element-by-element multiplication:

```r
m * m # standard element-by-element
m %*% m # matrix multiplication
```

2.3. Matrix Operations

You can sum all columns or rows using their respective functions

```r
colSum(stock.matrix)
rowSum(stock.matrix)
```

There exists a mean function for matrices aswell

```r
rowMeans(stock.matrix)
colMeans(stock.matrix)
```

You can also add columns & rows to an already existsing matrix

```r
fb <- c(111,112,322,344,123,532,232)
tech.stocks <- rbind(stock.matrix,fb)
```

For more operations consult a cheat sheet that is available under this link

2.4. Selection and Indexing

Selecting a matrix is generally done with `mat[rows, cols]` and leaving a field empty will select all its members.

```r
mat[,1] # selects all rows from the 1st column
mat[2,] # selects all columns from the 2nd row
mat[1:3,] # selects all columns out of the 1st, 2nd and 3rd rows
```

2.5. Factor & Categorical Matrices

The factor function is used for categorizing vectors and can be used as follows

```r
temps <- c("cold", "hot", "med", "hot", "hot", "cold" "hot")

fact.temp <- factor(temps, ordered = T, levels = c("cold", "med", hot))

# then you can call the summary function to get the frequencies of each level
summary(fact.temp)
```

3. Data Frames

There's alot of Data Frames already pre-built in R for experimenting. You can access them using `data()`.

To create one, use `data.frame()` and inside it put a list of vectors.

```r
df <- data.frame(temp, price, rain) # column labels are the vector names

df <- data.frame(temperature = temp, price.for.service = price, is.raining =
rain) # manually name columns
```

The `head()` and `tail()` functions return the beginning and end rows of a data frame respectively.

```r
head(state.x77) # first six rows
tail(state.x77) # last six rows
```

You can use the `str()` function to return the structure of a data frame and use the `summary()` function for a nicely formatted summary of each column including the mix/maxes, quartile ranges, means and medians. So essentially a numeric box/whisker plot.

```r
summary(state.x77)
```

3.1. Selecting & Indexing

It works like selecting & indexing matrices, but with the addition of label selection.

```r
df[,"rain"] # selects all the rows in the "rain" column
df[2,] # selects all the columns in the 2nd row
```

There is a distinction between single and double brackets. Single brackets return a data frame and double brackets (`[[]]`) return a vector.

You can also access the columns with `df$rain` as a shorthand. This is the most common way of grabbing columns from data frames.

There exists the `subset()` function that allows for extracting chunks of a df based on a certain condition

```
subset(df, subset = rain == T) # only outputs rows where rain is true.
```

To order a dataframe, you use the `order()` function to create an order hierarchy for ordinal values

```
temp <- c(22, 32.4, 28.2, 32.9)

df <- data.frame(temp)

sorted.temp <- order(df['temp']) # 4 2 3 1

desc.temp <- order(-df['temp']) # 1 3 2 4

df[sorted.temp,] # returns the indexed elements in the ascending order.
```

3.2. Data Frame Operations

There's `csv` support for data frames, you can read from them and write to them:

```
df <- read.csv("some_file.csv")

write.csv(df, file = "export_file.csv")
```

To get information about certain aspects of a data frame, you can use the following methods:

```
ncols(df) # number of columns
nrows(df) # number of rows

colnames(df) # returns all the columnn Labels
rownames(df) # returns the row names (be careful as in large data frames this output will be massive)
```

Editing is also allowed using assignments

```
df[1,2] <- TRUE
```

To create new columns, you can use the dollar sign shorthand I briefly mentioned earlier

```
df$newcol <- 2*df$temp
```

Also, excluding rows & columns from a selection is possible with a negative sign

```
df[-2,] # selects every row except the 2nd
```

4. Lists

Lists allow combinations of data types. They are straight forward and don't require a deeper explaination

```
v <- c(12, 32, 44)
m <- matrix(1:100, ncols=5)
df <- mtcars

my.list <- list(the.vector = v, the.matrix = m, cars = df)
```

You can essentially query them as you would data frames using `my.list[1]`, `my.list['the.vector']` or `my.list$the.vector`. The bracket notation returns a list, and then dollar sign returns a vector. You would have to use double brackets (`my.list[[1]]`) to return a vector without using a dollar sign.

5. Data I/O

This chapter focuses on the different ways of inputting and outputting data such as **csv**, **excel** or **sql**.

5.1. CSV Files

CSV stands for "comma seperated values" and is one of the most common ways of receiving data for further analysys.

As mentioned in the "Data Frames Operations" section, you can read from a csv and write to a csv using the following syntax

```
  write.csv(mtcars, file='some_file.csv')

  ex <- read.csv('another_file.csv')
```

Whenever you read a csv, you will get a data frame so all the data frame operations apply for csv imports.

5.2. Excel Files

You can either use the **readxl** and **writexl** from tidyverse or use an older library, which isn't as easy to use. I'll use **readxl**.

First, you have to install the library by executing `install.packages("readxl")` inside of the console and typing `library(readxl)` to load it up.

Then you'll have to use the `excel_sheets("path")` function to output the available sheet names and then call `read_excel("path" sheet = "somesheet")` to save it to a Data Frame. For example

```

excel_sheets('sample-sales-data.xlsx') # "Sheet1"

df <- read_excel('sample-sales-data.xlsx', sheet = "Sheet1")

```

You can also download an entire workbook (multiple sheets) into a "list":

```
  entire.workbook <- lapply(excel_sheets('sample-sales-data.xlsx'), read_excel,
path='sample-sales-data.xlsx')
```

This applies the `read_excel` function on every sheet from the sample-sales-data file and puts it into a list, so `entire.workbook[[1]]` returns the first sheet.

To write to an excel file, you need to install an additional library: `install.packages('xlsx')`. Then, you'll need to load it up using `library(xlsx)` and finally write the following:

```
  write.xlsx(mtcars, 'output_example.xlsx')
```

5.3. SQL

There's alot of SQL-flavors, so this section is going to focus on the general approach of finding out how to connect R with an SQL-database.

The **RODBC** library is one way of connecting to databases. Regardless of what you use, it is highly recommended that you first google your database choice + R. Here's an example use of RODBC:

```r
install.packages("RODBC")
library(RODBC)

myconn <- odbcConnect("Database_name", uid="User_ID", pwd="password")
dat <- sqlFetch(myconn, "Table_name")
querydat <- sqlQuery(myconn, "SELECT * FROM table")
close(myconn)
```

Here are some general tips for most common SQL-flavors:

- **MySQL** - The RMySQL package
- **Oracle** - The ROracle package
- **JDBC** - The RJDBC package

Like already mentioned, consult Google or ChatGPT for your individual SQL-flavor.

5.4. Web Scraping

To fully understand web scraping in R, knowledge of basic HTML and CSSis required. Furthermore, you need to understand the pipe operator (%>%), which is the same as "|" in unix, so essentially chaining commands with their inputs and outputs.

The first step is to open up "inspect element" and figure out what information you want to scrape and how it's structured.

Then the 'rvest' library needs to be installed and we can scrape an imdb-page using it.

```r
install.packages('rvest')
library(rvest)

lego_movie <- read_html("http://www.imdb.com/title/tt1490017")

rating <- lego_movie %>%
  html_nodes("strong span") %>%
  html_text() %>%
  as.numeric()

cast <- lego_movie %>%
  html_nodes("#titleCast .itemprop span") %>%
  html_text

...
```

A powerful feature of 'rvest' is that you can view demo scrapes to see more use cases under `demo(package='rvest')`. Then, a list of topics will display, choose one and call `demo(package='rvest', topic='yourtopic')`.

6. Programming Basics

This chapter covers the basic programming syntax in R.

6.1. Logical Operators

The AND operator is found under the symbol "&", the OR operator is the symbol "|" and the NOT operator stays "!".

```r
(x < 20) & (x > 10)

(x < 10) | (x > 100)

!(x = 10) & (x != 100) # both ways are valid
```

6.2. If Statements

They are identical to Java if-statements:

```
if (x == 10) {
  print("X is equal to 10")
} else if (x == 12) {
  print("X is equal to 12")
} else {
  print("X is something else")
}
```

6.3. While Loops

They are identical to Java while-loops:

```
while (x<10) {
  x <- x+1

  if (x == 10) {
    # useless but I wanted to show how to break
    break
  }
}
```

6.4. For Loops

They are very similar to Java for-loops

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

for (variable in v) {
  print(variable)
}

for (i in 1:10) {
  print(i)
}
```

6.5. Functions

They are similar to JavaScript functions, although not exactly the same:

```
 hello <- function(name="Frank") {

   print(paste("Hello", name))

 }

 hello("Alex") # "Hello Alex"
 hello() # "Hello Frank"

 add_num <- function(num1, num2) {
   my.sum <- num1 + num2
   return(my.sum)
 }

 result <- add_num(4,5) # 9
```

You can also create anonymous functions (similar to lambdas) that you can use e.g. in apply functions (discussed later)

```
result <- sapply(v, function(num){num*2})
```

7. Intermediate R Programming

This chapter discusses the R-specific and generally more advanced programming features.

7.1. Built-in R Features

- `seq(0,100,by=2)` - generates a sequence, takes in the starting value, ending value end the step size.
- `sort(v, decreasing = T)` - sorts a vector (increasing order by default), works on characters aswell.
- `rev(v)` - reverses an object.
- `str(mtcars)` - shows you the structure of an object.
- `summary(mtcars)` - already mentioned statistical summary of an object.
- `append(v1,v2)` - merge objects together.
- `is.` - checks the data type, e.g. `is.vector(v)`
- `as.` - converts the data type, e.g. `as.list(v)`

7.2. Apply

Apply is essentially a for-each loop. For example, the `lappy` (list-apply) function takes in a vector and a function with a parameter input and "applies" the function to each vector element and returns the result as a list:

```
v <- c(1,2,3,4,5)

addrand <- function(x) {
  ran <- sample(1:100, 1) # takes one randon sample out of a vector
  return (x+ran)
}

result <- lapply(v, addrand)
print(result)
```

There are also other apply methods (that stem from `lapply`)

- `sapply()` - simple apply, preserves the dimensions, usually outputs a vector.
- `apply()` - used for data frames & matrices, additional parameter "MARGIN" (1 - rows, 2 - columns)
- `vapply()` - similar to sapply, but you can specify the type of the return value (FUN.VALUE = integer(1))
- etc.

You can also use anonymous functions as discussed in the "Functions" section.

7.3. Math Functions

- `abs()` - computes the absolute value
- `sum()` - returns the sum of all the elements present in the input.
- `mean()` computes the arithmetic mean (average)
- `round()` rounds values (additional arguments to nearest).

For further math functions reference the R reference card

7.4. Regular Expressions

There are two functions to use: `grep()` and `grepl()`. One returns an index (used for vectors, lists, etc.) and one returns a logical (boolean, used for actual strings)

```r
text <- "Hi there!"

grepl('there', text) # TRUE
grepl('dog', text) # FALSE

grep('b', c('a', 'b', 'c')) # 2
```

7.5. Dates and Timestamps

There exists a `Sys.Date()` function that returns the current date, and a `as.Date()` function that converts text into a date. You can pass in a format argument for converting the date.

```r
my.date <- as.Date("Nov-03-90", format = '%b-%d-%y')
```

The formatting options for dates are comprised of:

- `%d` - Day of the month (decimal)
- `%m` - Month (decimal)
- `%b` - Month (abbreviated)
- `%B` - Month (full name)
- `%y` - Year (2 digits)
- `%Y` - Year (4 digits)

If you want a date including the hour, minute, second and timezone, use `POSIXct`:

```r
as.POSIXct("11:02:03", format="%H:%M:%S") # "2024-08-26 11:02:03 GMT+2"
```