Introduction to Programming in R

Martin Studer

June 29, 2017

Course Objectives

The goal of this course is to provide a **systematic introduction** to the open-source statistical programming language R. At the end of this course you should have an understanding of the **core language concepts**, allowing you to perform **basic data manipulation and analysis** tasks as well as to dig further into more advanced topics.

Introduction

What is R

- R is a functional and object-oriented programming language and environment for statistical computing and graphics
- R is an open-source implementation of the S programming language which was originally developed at Bell Laboratories (formerly AT&T, now Lucent technologies)
- R is free software and can be downloaded from https://www.r-project.org/
- R is highly extensible through so-called *packages* which can be freely downloaded from CRAN

Using R Interactively

- Default prompt `>`: shown when R is expecting input commands

```
> 1 + 1
```

- Continuation prompt `+`: shown when R is expecting further input for an incomplete command

```
> 5 *
+ 3
```

- Pressing `<Enter>` sends a command to the R engine for evaluation
- Use the up ↑ and down ↓ arrow keys to navigate the history of commands

Data Objects

Objectives

The goal of this first chapter is to provide a **basic understanding of R's data structures and object system** and as such to provide a good foundation to understand programming and performing statistical analyses in R.

Creating & Saving Objects (1)

Objects are created by function calls. A vector, for example, can be created using the c (combine) function:

```
> c(2, 3, 5.6)
```

```
[1] 2.0 3.0 5.6
```

To save the object in a *variable*, **assign** it to a name:

```
> numbers <- c(2.3, 31, 2.54, 4, 23.1)
> numbers = c(2.3, 31, 2.54, 4, 23.1)
```

Use either <- or = for assignments. Best practice is to use **<-**

Creating & Saving Objects (2)

In R, elementary commands consist of either **expressions** or **assignments**. If an expression is given as a command, it is evaluated, printed and the value is lost. An assignment also evaluates an expression but assigns the result to a variable. The result, however, is not automatically printed. By wrapping assignments in `(...)` you can turn them into expressions and as such also print the result:

```
> c(2.3, 31, 2.54, 4, 23.1) # expression
```

```
[1]  2.30 31.00  2.54  4.00 23.10
```

```
> numbers <- c(2.3,31,2.54,4,23.1) # assignment
> (numbers <- c(2.3,31,2.54,4,23.1)) # assignment to expression
```

```
[1]  2.30 31.00  2.54  4.00 23.10
```

Vector Definition

- Vectors are **ordered sequences of elements**
- They are the **basic building blocks** for all data structures
- Vectors come in five flavors, or so-called **modes**:
  - Numeric, e.g., `5`, `1.25`, `3.14159`, `2.0`
  - Complex, e.g., `3+4i`, `3.5i`
  - Character, e.g., `"small"`, `"medium"`, `"large"`
  - Logical, e.g., `TRUE`, `FALSE`
  - `NULL`, the null object
    - Sometimes results from a computation
    - Sometimes used as default value of function arguments
- Vectors are **atomic** data structures since **all elements must be of the same mode**

Creating Sequences

```
> (vec2 <- 5:10)
```

```
[1]  5  6  7  8  9 10
```

```
> (vec3 <- seq(from = 10, to = 1, by = -2))
```

```
[1] 10  8  6  4  2
```

```
> seq(along = vec1) # instead of 1:length(vec1)
```

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

```
> seq(from = 2, by = 3, length = 4)
```

```
[1]  2  5  8 11
```

Arithmetic in R is Vectorized (1)

R is **vectorized**. Vectors can be used in arithmetic expressions, in which case the operations are performed element by element.

```
> 1:4
```

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

```
> 1:4 - 2
```

```
[1] -1  0  1  2
```

```
> 1:4 * 2
```

```
[1] 2 4 6 8
```

Missing Values (1)

In some cases the components of a vector may not be completely known. When an element or value is *not available* or a **missing value** in the statistical sense, a place within a vector is indicated by the special value `NA`.

The two functions `is.na` and `which` are used to locate and manage missing values:

- The function `is.na()` returns `TRUE` and `FALSE` for missing and non-missing values, respectively
- The function `which()` returns integer indices to the `TRUE` values in its input: `which(is.na(x))`

Subscript by Logical Vectors (1)

```
> vec <- c(5, 1, NA, 11.1, 3, NA)
> names(vec) <- LETTERS[1:6]
> vec > 4
```

```
    A     B     C     D     E     F 
 TRUE FALSE    NA  TRUE FALSE    NA 
```

```
> vec[vec > 4]
```

```
   A <NA>    D <NA> 
 5.0   NA 11.1   NA 
```

Subscripts and Missing Values (2)

```
> vec <- c(5, 1, NA, 11.1, 3, NA)
> vec[!is.na(vec)]
```

```
[1]  5.0  1.0 11.1  3.0
```

```
> vec # vec is still the full vector!
```

```
[1]  5.0  1.0   NA 11.1  3.0   NA
```

To have `vec` only keep the non-missing values, you have to **assign the result** of the expression back to `vec`:

```
> (vec <- vec[!is.na(vec)])
```

```
[1]  5.0  1.0 11.1  3.0
```

Creating Matrices (1)

Several functions are useful for creating matrices:

Function	Description
`matrix`	Creates a matrix from a vector by specifying number of rows and columns
`dim`	Creates a matrix from a vector by specifying the row and column dimensions
`rbind`	Creates a matrix by binding vectors together row by row
`cbind`	Creates a matrix by binding vectors together column by column
`diag`	Creates a diagonal matrix from a vector

Arrays

An **array** is a generalized version of a matrix for `n` **dimensions**. A classical example is a cube:

```
> arr <- array(1:20, dim = c(2, 5, 2))
> arr # the 3rd dimension is printed sequentially
```

```
, , 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
```

Array Subscripting

Extract elements or subsets using the bracket `[ , , ... ]` operator using as many subscripts as there are dimensions in the array. The syntax for subscripting is

`arrayName[dim1Subscript, dim2Subscript, ...]`

Example:

```
> arr[, 2:3, 2]
```

```
     [,1] [,2]
[1,]   13   15
[2,]   14   16
```

Creating Lists

Create a list with the `list` function:

```
> mat <- matrix(letters[1:10], nrow = 2, byrow = TRUE)
> (lst <- list(1:5, mat, mean))
```

```
[[1]]
[1] 1 2 3 4 5

[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,] "a"  "b"  "c"  "d"  "e"
[2,] "f"  "g"  "h"  "i"  "j"

[[3]]
function (x, ...)
UseMethod("mean")
<bytecode: 0x4db51b0>
<environment: namespace:base>
```

List Subscripting

There are three different operators for subscripting lists:

- `[ ]`: **selects** (possibly many) components of a list
  - the result is a **list** again (consider a list as a vector of components!)
  - the usual five vector subscripting types are allowed
- `[[ ]]`: **extracts** the **content** of a component in the list
  - only **one** component can be extracted from
  - accepts both the index or the name of a component
- `$`: **extracts** the **content** of a single **named** component

Chaining Subscripts

When using recursive lists it is quite often necessary to chain multiple subscripts together to extract data deeper in the structure.

```
> lst[["aaa"]][4:5]
```

```
[1] 4 5
```

```
> lst$bbb[, "c5"]
```

```
 r1  r2
"e" "j"
```

```
> # Brain exercise :-)
> lst["bbb"][[1]][1, c(2, 4), drop = FALSE][, 2]
```

```
[1] "d"
```

Data Frame Definition

The **data frame** is the most widely used data structure in R. Data frames provide a convenient way to represent observations on a set of variables.

- Like a **matrix**, a data frame is a rectangular structure
  - rows represent observations
  - columns represent the different variables
- Like a **list**, a data frame is recursive and its columns can have different modes (types); the components must be vectors of equal length

A data frame is best remembered as a **list of equal-length vectors**.

Creating Data Frames (1)

Use the **data.frame** function to combine several equal-length vectors into a data frame:

```
> (df <- data.frame(
+   aaa = 1:4,
+   bbb = letters[26:23],
+   ccc = c(FALSE, TRUE, TRUE, FALSE)
+ ))
```

```
  aaa bbb   ccc
1   1   z FALSE
2   2   y  TRUE
3   3   x  TRUE
4   4   w FALSE
```

Subscripting Data Frames (1)

Data frames are lists, so you can use **list subscripting syntax**:

```
> df$index # extract column 'index'
```

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

```
> df[[1]] # extract 1st column
```

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

```
> df[1:2] # select the first two columns
```

```
     index letter
row1     1      z
row2     2      y
row3     3      x
row4     4      w
```

Note: The result is a `data.frame`!

Subscripting Data Frames (2)

You can also use **matrix subscripting syntax**, because data frames are *rectangular*:

```
> df[, 2] # extract 2nd column
```

```
[1] z y x w
Levels: w x y z
```

```
> df[, "letter"] # extract column 'letter'
```

```
[1] z y x w
Levels: w x y z
```

```
> df[1:2, c(TRUE, TRUE, FALSE)]
```

```
     index letter
row1     1      z
row2     2      y
```

Functions

Argument Types

There are three types of arguments for functions:

- **Named required arguments** have a specific name but no default value and as such require user input.
- **Named optional arguments** have a specific name and a default value and do not require user input.
- **Unspecified arguments** are represented by the **ellipses argument** (...; sometimes referred to as dot-dot-dot) in the argument list. This allows specification of arbitrary named and unnamed parameters.

Function Definitions

The basic syntax for defining a function is

```
functionName <- function(arguments) {
  body
}
```

where

- `functionName` is the name of a new function
- `function`, followed by parentheses that enclose the arguments, is a *keyword*
- `arguments` is a comma-separated list of arguments
- `body` is the set of commands executed by the function

Function Return Values (1)

A function typically returns the **last expression** of the body, unless it causes an error or a **return is explicitly defined** by using the **`return()`** function.

```
> sphere <- function(r) {
+   # The list(..) statement is the last statement
+   # in the function and therefore it is the return
+   # value
+   list(
+     circumference = 2 * pi * r,
+     surface = 4 * pi * r^2,
+     volume = 4 / 3 * pi * r^3
+   )
+ }
```

Conditional Execution: if-else Examples (1)

Euclid's algorithm for the greatest common divisor:

```
> # Assumption: a > 0, b > 0
> gcd <- function(a, b) {
+   if(a == b) {
+     a
+   } else if(a > b) {
+     gcd(a - b, b) # recursion!
+   } else {
+     gcd(a, b - a) # recursion!
+   }
+ }
```

for Loop

The syntax of the **for** loop is

```
for (variable in vector) {
  ... code ...
}
```

Example:

```
> for (month in month.name) { # month.name is built-in
+   print(sprintf("Month %s has %s days.", month,
+                 monthToDays(month, year = 2017)))
+ }
```

Implicit Looping with the `*apply` Functions

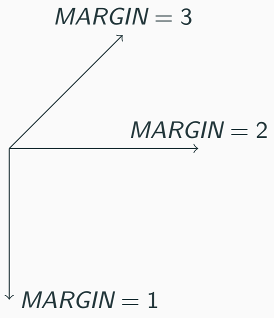
The **`*apply`** functions (`apply`, `lapply`, `sapply`, `tapply`, `by`, …) implement looping using internal compiled code. This generally makes them faster than `for`, `while` and `repeat`.

The `*apply` functions have the following core arguments:

- The object over which to loop (`X`, `data`)
- The function to be applied (`FUN`)
- Additional arguments to be passed to the function (`FUN`) through the ellipses argument (`...`)

apply

Apply a function to the **margins of an array/matrix** (sometimes also a data frame).



```
apply(data, MARGIN = 1, mean)
apply(data, MARGIN = 2, summary)
apply(data, MARGIN = 3, sum)
```

tapply (1)

tapply is used to apply a function to a **vector**, subject to one or more **grouping variables** (= list of vectors/factors).

Example with one grouping variable:

```
> tapply(mtcars$mpg, mtcars$am, mean)
```

```
       0        1
17.14737 24.39231
```

Importing / Exporting Data

Importing / Exporting Data

R provides **many built-in** functions to read/write data from various sources and there are many more available through **external packages** from e.g. CRAN.

In this chapter we will only scratch the surface and will be covering some of the most typical file formats such as delimited text, CSV and Excel files.

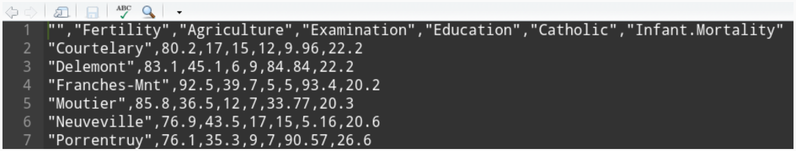
Reading Tabular Data (1)

`read.table()`, `read.csv()`, `read.csv2()`, `read.delim()` and `read.delim2()` are the most commonly used functions to **read tabular data from text files**. They all share similar arguments and all return a `data.frame` object. The most important arguments are:

- `file`: file path
- `sep`: the field separator; delimiter separating the column (e.g. `,` or `;`)
- `header`: a logical value indicating whether the file contains the names of the variables as its first line
- `row.names`: a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or a character string giving the name of the table column containing the row names
- `col.names`: a vector of optional names for the variables

See `?read.table` for a complete description.

Reading Tabular Data (2)



```
"","Fertility","Agriculture","Examination","Education","Catholic","Infant.Mortality"
"Courtelary",80.2,17,15,12,9.96,22.2
"Delemont",83.1,45.1,6,9,84.84,22.2
"Franches-Mnt",92.5,39.7,5,5,93.4,20.2
"Moutier",85.8,36.5,12,7,33.77,20.3
"Neuveville",76.9,43.5,17,15,5.16,20.6
"Porrentruy",76.1,35.3,9,7,90.57,26.6
```

```
> mySwiss <- read.table("examples/swiss.txt", sep = ",",
+                        header = TRUE, row.names = 1)
> mySwiss[1:3, 1:3]
```

```
             Fertility Agriculture Examination
Courtelary        80.2        17.0          15
Delemont          83.1        45.1           6
Franches-Mnt      92.5        39.7           5
```

Reading & Writing Excel Files

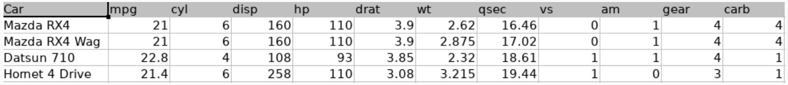
In order to read and write Excel files, you will need to use one of the many freely available packages. Here, we are going to use **XLConnect**, a package developed and maintained by Mirai Solutions.

XLConnect is a comprehensive and **cross-platform** R package for manipulating Microsoft Excel files from within R. It is completely cross-platform and as such runs under Windows, Unix/Linux and Mac (32- and 64-bit). Moreover, it **does not require any installation of Microsoft Excel** or any other special drivers to be able to read & write Excel files. The only requirement is a recent version of a **Java** Runtime Environment (JRE).

If you don't have the package installed yet, you will need to install it using

```
> install.packages("XLConnect", dependencies = TRUE)
```

Reading Excel Files (1)



Car	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
Homet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1

```
> require(XLConnect) # load package
> wb <- loadWorkbook("examples/data.xlsx") # load workbook
> data <- readWorksheet(wb, sheet = "mtcars", # read worksheet
+                       rownames = "Car")
> data[1:5, 1:9]
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0
```

Reading Excel Files (2)

Generally, first *load* a workbook using `loadWorkbook()`, then use any combination of

- `readWorksheet()` to read a worksheet

```
> getSheets(wb) # query worksheets
```

```
[1] "mtcars" "co2"
```

```
> data <- readWorksheet(wb, sheet = "mtcars")
> data <- readWorksheet(wb, sheet = 1)
```

- `readNamedRegion()` to read named regions

```
> getDefinedNames(wb) # query named regions
```

```
[1] "cotwo"  "mtcars"
```

```
> data <- readNamedRegion(wb, name = "mtcars")
```

Writing Workbooks

XLConnect also allows creating and writing worksheets and named regions. While XLConnect can be used to produce fairly complex reports, we just cover the basic usage of `writeWorksheet()` and `writeNamedRegion()` for data export.

The process of creating/writing a workbook generally involves loading a workbook, writing worksheets and named regions and finally **saving the workbook to disk using** `saveWorkbook()`:

```
> # Create a new workbook
> wb <- loadWorkbook("swiss.xlsx", create = TRUE)
> # Create a sheet named 'swiss'
> createSheet(wb, name = "swiss")
> # Write built-in data set swiss to sheet 'swiss'
> writeWorksheet(wb, data = swiss, sheet = "swiss", header = TRUE,
+                rownames = "Province")
> # Save workbook - only now the workbook is written to disk!
> saveWorkbook(wb)
```