[DS-02] Introduction to R

Miguel-Angel Canela
Associate Professor, IESE Business School

General introduction

R is a language which comes with an environment for computing and graphics, available for Windows, Linux and Macintosh. It includes an extensive variety of techniques for statistical testing, predictive modeling and data visualization. R can be extended by hundreds of additional packages available at the **Comprehensive R Archive Network** (CRAN), which cover virtually every aspect of data management and analysis. As a rule, this course does not use one of these packages unless it makes a real difference.

R has been gaining acceptance in the business intelligence industry in the last years: nowadays we can use R resources within data management systems like Oracle, or resource planning systems like SAP. They are also available in cloud environments like Amazon Web Services (AWS), Microsoft Azure or IBM Data Scientist Workbench (so far completely free). Although facing fierce competition from Python, R is still the default analytics tool in many well known companies (see Bion *et al*, 2017, for an example).

It will be very easy for you to download R from `www.r-project.com` and install it in your computer. This provides you with a graphical user interface (GUI), called the **console**, in which you can type or paste your code. The console works a bit different in Macintosh and Windows. In the Windows console, what we type is in red and the R response is in blue. In Macintosh, we type in blue and the response comes in black. When R is ready for our code, the console shows a **prompt** which is the "greater than" symbol (`>`). With the `Return` key, we finish a line of code, which is usually interpreted as a request for execution. But R can detect that your input is not finished, and then it waits for more input, showing a different prompt, the plus symbol (`+`).

In the console, this is usually seen as follows

```
> 2 + 2
 [1] 4
```

If we type `2 + 2`, we get the result of this calculation. But, when we want to store this result, we give it a name, as follows.

```
> a = 2 + 2
```

Note that the value of `2 + 2` is not printed now. If we want it to be printed to the screen, we have to ask for that explicitly.

```
> a
 [1] 4
```

¶ In some programming languages, you should type `print(a)` or similar to get `a` printed on the screen.

Typically, R programmers replace the equal sign (`=`) by an arrow (`<-`), and nothing changes. It is recommended, to the beginner, to use the arrow system, to avoid mistakes. Nevertheless, I use the equal sign, to make my R code more comparable to Python code.

Interacting with R

The R console is not user-friendly, so you will probably prefer to work in an **interactive developer environment** (IDE). Among IDE's, **RStudio** is the leading choice and, nowadays, most R coders prefer RStudio to the console. In RStudio, you have the console plus other windows that may help you to organize your task.

Besides working directly in the console or in an IDE like RStudio, a third approach is based on the **notebook** concept. In a notebook, you enter your code as the **input**, getting the results as the **output**. You can include in the notebook graphics that, in the console or in RStudio, would appear in a separate graphics window. In the notebook field, **Jupyter** is the leading choice, followed by **Apache Zeppelin**. These two are multilingual, that is, can be used for other languages, besides R. Jupyter has powerful supporters and very smart people behind it, so we probably see plenty of Jupyter notebooks in the immediate future.

Learning about R

There are many books for learning about R, but some of them are too statistically-oriented, so they may not be for you. Kabacoff (2010) is a popular choice. If you want a thick one, Crawley (2012) will do. Also, Wickham & Grolemund (2016) is excellent, but strongly based on a set of packages (the tidyverse) developed by the authors (both at RStudio). These packages are used by many data scientists, but this course skips them to make it shorter.

There is also plenty of learning materials in Internet, including MOOC's. For instance, IBM provides some (free) Data Science courses in the **Data Science and Cognitive Computing Courses** (`cognitiveclass.ai`), including an *R 101 course*. Those willing to carry out deeper study can find in **Coursera** a pack of courses called *Data Science Specialization* (`www.coursera.org/specializations/jhu-data-science`), based on R.

Finally, **DataCamp** offers, under subscription or academic license, an impressive collection of Data Science courses, using either R or Python. In addition to follow DataCamp, you can benefit from the **Datacamp Community Tutorials**, which are free and cover a wide range of topics.

Vectors

R is **object-oriented**, with many classes of objects (but less than other languages like Python). I briefly comment in this lecture three examples. First, we have **vectors**. A vector is an ordered collection of elements which are all of the same type. Vectors can be **numeric**, **character** (string), **logical** (`TRUE`/`FALSE`) or of other types not discussed in this course. The following three examples are numeric (`x`), character (`y`) and logical (`z`), respectively.

```
> x = 1:10
> x
 [1] 1 2 3 4 5 6 7 8 9 10
> y = c('Messi', 'Neymar', 'Cristiano')
> y
 [1] "Messi" "Neymar" "Cristiano"
> z = x > 5
> z
 [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
```

Some comments on these examples:

- The expression `c(a,b)` joins the vectors `a` and `b`, returning a single vector whose length is the sum of the lengths of `a` and `b`. If they are not of the same type, R coerces them to a common type.
- The quote marks indicate character type. You can use single or double quotes, but take care of using the same on both sides of the string.
- An expression like `x > 5` returns a logical vector with one term for each term of `x`.

The first term of the vector `x` can be extracted as `x[1]`, the second term as `x[2]`, etc. A **factor** is a numeric vector in which the values have labels, called **levels**. Factors look very natural to statisticians (stat packages, like SPSS or Stata use similar systems), but weird to computer scientists. We can transform any character vector into a factor with the function `factor`.

```
> y_factor = factor(y)

> y_factor
 [1] Messi Neymar Cristiano
Levels:  Cristiano Messi Neymar
```

A saying of programmers with experience in other languages that explore R for the first is "in R everything is a vector". In other languages (e.g. in Python), vectors have to explicitly specified, but in R most definitions produce vectors without the coder having to specify it.

A simple example may help to understand this. Take the definition:

```
> a = 3
```

What kind of object is `a`? In Python, it would be a numeric variable and, to make it a vector, we should create a new object of vector type (called `array` in Python). In R it is, directly, a vector of length 1:

```
> is.vector(a)
 [1] TRUE

> length(a)
 [1] 1
```

Lists

A **list** is like a vector, but it can contain objects of different type. Lists are practical to pack diverse stuff, but long lists are inefficient, because R has to check the type of every object in the list, and this makes processing slower.

The elements of a list are identified by the index or by the name (if they have it). Let me start with a list of unnamed objects:

```
> L = list(1:10, c('Messi', 'Neymar', 'Cristiano')), TRUE)

> L
[[1]]
 [1] 1 2 3 4 5 6 7 8 9 10

[[2]]
 [1] "Messi" "Neymar" "Cristiano"
```

```
[[3]]
 [1] TRUE
```

Now, I call the first object of the list (note the double brackets):

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

I redefine now `L` as a list of named objects:

```
> L = list(L1=1:10, L2=c('Messi', 'Neymar', 'Cristiano'), L3=TRUE)
```

Now, the first element in the list can be specified with its name. There are two ways of doing this: (a) with the name of the list followed by the name of the element, separated by the dollar sign (`L$L2`), and (b) with the name of the element within square brackets (`L['L2']`). I use the second method, for comparability with Python.

```
> L['L2']
$L2
 [1] "Messi" "Neymar" "Cristiano"
```

Subsetting

Extracting parts of R objects is called is called **subsetting**. Vectors, and also the data frames discussed later in this course, can be subsetted in various ways. Subsetting through indexes is straightfortward:

```
> x[1:3]
 [1] 1 2 3
```

The minus sign (`-`) within the brackets means "drop" (not "starting from the end", as in other languages):

```
> x[-c(1, 4)]
 [1] 2 3 5 6 7 8 9 10
```

Subsetting can also be performed with expressions, as in:

```
> x[x >= 5]
 [1] 5 6 7 8 9 10
```

Note that the expression within the brackets creates a logical vector,

```
> x >= 5
 [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
```

on which subsetting is really based. Only the `TRUE` terms are selected when we use the expression for subsetting purposes.

For loop

The **for loop** exists in all current programming languages, and all coders have used it many times to avoid repetition. I give first a supersimple example:

```
> x = 1

> for(i in 1:10) x = x + 1

> x
 [1] 11
```

Now, a bit more difficult. The following loop generates a sequence of Fibonacci numbers (quite popular since they appeared in the *The Da Vinci Code*, used to unlock a safe).

```
> x = c(1, 1)

> for(i in 3:10) x[i] = x[i-1] + x[i-2]

> x
 [1] 1 1 2 3 5 8 13 21 34 55
```

Functions

R is a fully functional language. Its real power comes from defining the operations that we wish to perform as **functions**, so they can be applied many times. A simple example of the definition of a function follows.

```
> f = function(x) 1/(1 - x**2)
```

¶ R admits the "hat" (^) for taking powers, but here, as in other places, I favor the comparability with Python.

When we define a function, R just takes note of the definition, accepting it when it is syntactically correct (parentheses, commas, etc). The function can be applied later to different arguments.

```
> f(0.5)
 [1] 1.333333
```

As you can see in the example, the syntax is `fname = function(x) expression`. The expression involves the argument `x` and objects already defined. If we apply the function to an argument for which it does not make sense, R will complain. The complaint can come in various ways. For instance, even if the argument supplied is not right, we get a response in the following example.

```
> f(1)
 [1] Inf
```

But, trying `f('Mary')`, we would get an error message.

```
> f('Mary')
 "Error in x^2 :  non-numeric argument to binary operator"
```

Functions can have more than one argument, as in

```
> f = function(x, y) x*y/(x**2 + y**2).
> f(1, 1)
 [1] 0.5
```

If the definition of a function involves several steps, for instance in functions defined by means of `for` loops, the expression is deployed in several lines and these lines are enclosed by curly braces ({}). In that case, it is better to include within the brackets a last line in which it is specified, with `return(value)`, what the function returns. In the example which follows, the function returns a Fibonacci sequence of the length requested.

```
> f = function(n) {
+   x = c(1, 1)
+   for(i in 3:n) x[i] = x[i-1] + x[i-2]
+   return(x)
+ }
> f(10)
 [1] 1 1 2 3 5 8 13 21 34 55
```

A **vectorized function** is one that, when applied to a vector, works on every term of the vector separately and returns a vector of the same length. Many native R functions, like `exp` or `sqrt`, and operators like `+` or `*` are vectorized:

```
> sqrt(5:1)
 [1] 2.236068 2.000000 1.732051 1.414214 1.000000
```

User-defined functions can be vectorized with the function `Vectorize`.

References

1. R Bion, R Chang & J Goodman (2017), *How R Helps Airbnb Make the Most of Its Data*, `peerj.com/preprints/3182`.
2. MJ Crawley (2012), *The R Book*, Wiley. Free access at `ftp.tuebingen.mpg.de/pub/kyb/bresciani`.
3. RI Kabacoff (2010), *R in Action*, Manning. Free access at `kek.ksu.ru/eos/DataMining`.
4. H Wickham & G Grolemund (2016), *R for Data Science*, O'Reilly. Free access at `r4ds.had.co.nz`.