Basic Concepts of the R Language

L. Torgo

ltorgo@dal.ca
Faculty of Computer Science / Institute for Big Data Analytics
Dalhousie University

Jan, 2020

Basic interaction with the R console

- The most common form of interaction with R is through the command line at the console
  - User types a command
  - Presses the ENTER key
  - R “returns” the answer
- It is also possible to store a sequence of commands in a file (typically with the `.R` extension) and then ask R to execute all commands in the file

Basic interaction with the R console (2)

- We may also use the console as a simple calculator

```
1 + 3/5 * 6^2

## [1] 22.6
```


Basic interaction with the R console (3)

- We may also take advantage of the many functions available in R

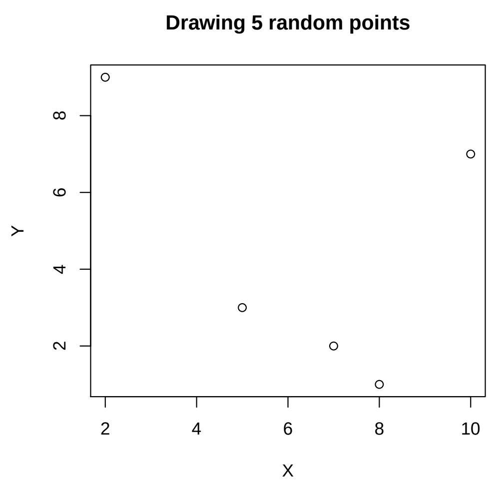
```
rnorm(5, mean = 30, sd = 10)

## [1] 28.100  4.092 29.904 10.611 23.599
```

```
# function composition example
mean(sample(1:1000, 30))

## [1] 530.3
```

Basic interaction with the R console (4)

- We may produce plots

```
plot(sample(1:10, 5), sample(1:10, 5),
    main = "Drawing 5 random points",
    xlab = "X", ylab = "Y")
```



The notion of Variable

- In R, data are stored in variables.
- A variable is a "place" with a **name** used to store information
  - Different types of **objects** (e.g. numbers, text, data tables, graphs, etc.).
- The assignment is the operation that allows us to store an object on a variable
- Later we may use the content stored in a variable using its name.

Basic data types

R objects may store a diverse type of information.

R basic data types

- *Numbers*: e.g. `5`, `6.3`, `10.344`, `-2.3`, `-7`
- *Strings* : e.g. `"hello"`, `"it is sunny"`, `"my name is Ana"`
  Note: one the of the most frequent errors - confusing *names* of variables with *text values* (i.e. strings)! `hello` is the name of a variable, whilst `"hello"` is a string.
- *Logical values*: `TRUE`, `FALSE`
  Note: R is case-sensitive!
  `TRUE` is a logical value; `true` is the name of a variable.

The assignment - 1

- The assignment operator "`<-`" allows to store some content on a variable

```
vat <- 0.2
```

- The above stores the number 0.2 on a variable named `vat`
- Afterwards we may use the value stored on the variable using its name

```
priceVAT <- 240 * (1 + vat)
```

- This new example stores the value 288 ($= 240 \times (1 + 0.2)$) on the variable `priceVAT`
- We may thus put expressions on the right-side of an assignment

The assignement - 2

What goes on in an assignment?

1. **Calculate** the result of the expression on the right-side of the assignment (e.g. a numerical expression, a function call, etc.)
2. **Store** the result of the calculation in the variable indicated on the left side

- In this context, what do you think it is the value of `x` after the following operations?

```
k <- 10
g <- k/2
x <- g * 2
```

AFIRM

Still the variables...

- We may check the value stored in a variable at any time by typing its name followed by hitting the ENTER key

```
x <- 23^3
x

## [1] 12167
```

- The `^` signal is the exponentiation operator
- The odd `[1]` will be explained soon...
- And now a common mistake!

```
x <- true

## Error: object 'true' not found
```

AFIRM

A last note on the assignment operation...

- It is important to be aware that the assignment is destructive
- If we assign some content to a variable and this variable was storing another content, this latter value is “lost”,

```
x <- 23
x

## [1] 23

x <- 4
x

## [1] 4
```


Vectors

- Vectors are a type of R objects that can store sets of values of the same base type
  - e.g. the prices of an article sold in several stores
- Everytime some set of data has something in common and are of the same type, it may make sense to store them as a vector
- A vector is another example of a content that we may store in a R variable

Vectors (2)

- Let us create a vector with the set of prices of a product across 5 different stores

```
prices <- c(32.4, 35.4, 30.2, 35, 31.99)
prices

## [1] 32.40 35.40 30.20 35.00 31.99
```

- Note that on the right side of the assignment we have a call to the function `c()` using as arguments a set of 5 prices
- The function `c()` creates a vector containing the values received as arguments

Vectors (3)

- The function `c()` allows us to associate names to the set members. In the above example we could associate the name of the store with each price,

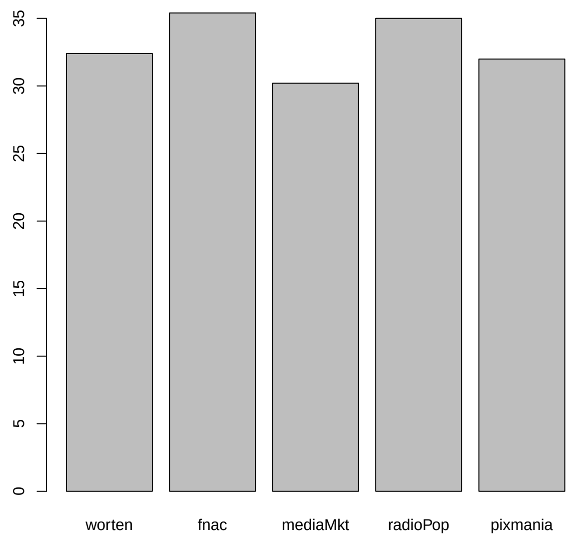
```
prices <- c(worten = 32.4, fnac = 35.4, mediaMkt = 30.2,
    radioPop = 35, pixmania = 31.99)
prices

##   worten     fnac mediaMkt radioPop pixmania
##    32.40    35.40    30.20    35.00    31.99
```

- This makes the vector meaning more clear and will also facilitate the access to the data as we will see.

Vectors (4)

- Besides being more clear, the use of names is also recommended as R will take advantage of these names in several situations.
- An example is in the creation of graphs with the data:

```
barplot(prices)
```



Basic Indexing

- When we have objects containing several values (e.g. vectors) we may want to access some of the values individually.
- That is the main **purpose of indexing**: access a subset of the values stored in a variable
- In mathematics we use indices. For instance, x_3 usually represents the 3rd element in a set of values x.
- In R the idea is similar:

```
prices <- c(worten=32.4,fnac=35.4,
            mediaMkt=30.2,radioPop=35,pixmania=31.99)
prices[3]

## mediaMkt
##     30.2
```

Basic Indexing (2)

- We may also use the vector position names to facilitate indexing

```
prices <- c(worten=32.4,fnac=35.4,
            mediaMkt=30.2,radioPop=35,pixmania=31.99)
prices["worten"]

## worten
##   32.4
```

- Please note that `worten` appears between quotation marks. This is essencial otherwise we would have an error! Why?
- Because without quotation marks R interprets `worten` as a variable name and tries to use its value. As it does not exists it complains,

```
prices[worten]

## Error:  object 'worten' not found
```

- Read and interpret error messages is one of the key competences we should practice.

Vectors of indices

- Using vectors as indices we may access more than one vector position at the same time

```
prices <- c(worten=32.4,fnac=35.4,
            mediaMkt=30.2,radioPop=35,pixmania=31.99)
prices[c(2,4)]

##     fnac radioPop
##     35.4     35.0
```

- We are thus accessing positions 2 and 4 of vector `prices`
- The same applies for vectors of names

```
prices[c("worten", "pixmania")]

##   worten pixmania
##    32.40    31.99
```

Vectors of indices (2)

- We may also use logical conditions to “query” the data!

```
prices[prices > 35]

## fnac
## 35.4
```

- The idea is that the result of the query are the values in the vector `prices` for which the logical condition is true
- Logical conditions can be as complex as we want using several logical operators available in R.
  What do you think the following instruction produces as result?

```
prices[prices > mean(prices)]

##      fnac radioPop
##      35.4     35.0
```

- Please note that this another example of function composition!

Vectorization of operations

- The great majority of R functions and operations can be applied to sets of values (e.g vectors)
- Suppose we want to know the prices after VAT in our vector `prices`

```
vat <- 0.23
(1+vat)*prices

##   worten     fnac mediaMkt radioPop pixmania
##  39.8520  43.5420  37.1460  43.0500  39.3477
```

- Notice that we have multiplied a number (1.2) by a set of numbers!
- The result is another set of numbers that are the result of the multiplication of each number by 1.2

Vectorization of operations (2)

- Although it does not make a lot of sense, notice this other example of vectorization,

```
sqrt(prices)

##   worten     fnac mediaMkt radioPop pixmania
## 5.692100 5.949790 5.495453 5.916080 5.655970
```

- By applying the function `sqrt()` to a vector instead of a single number we get as result a vector with the same size, resulting from applying the function to each individual member of the given vector.

Vectorization of operations (3)

- We can do similar things with two sets of numbers
- Suppose you have the prices of the product on the same stores in another city,

```
prices2 <- c(worten=32.5,fnac=34.6,mediaMkt=32,
             radioPop=34.4,pixmania=32.1)
prices2

##   worten     fnac mediaMkt radioPop pixmania
##     32.5     34.6     32.0     34.4     32.1
```

- What are the average prices on each store over the two cities?

```
(prices+prices2)/2

##   worten     fnac mediaMkt radioPop pixmania
##   32.450   35.000   31.100   34.700   32.045
```

- Notice how we have summed two vectors!

Logical conditions involving vectors

- Logical conditions involving vectors are another example of vectorization

```
prices > 35

##    worten      fnac mediaMkt radioPop pixmania
##     FALSE      TRUE    FALSE    FALSE    FALSE
```

- `prices` is a set of 5 numbers. We are comparing these 5 numbers with one number (35). As before the result is a vector with the results of each comparison. Sometimes the condition is true, others it is false.
- Now we can fully understand what is going on on a statement like `prices[prices > 35]`. The result of this indexing expression is to return the positions where the condition is true, i.e. this is a vector of Boolean values as you may confirm above.

Matrices

- As vectors, matrices can be used to store **sets** of values of the same base type that are somehow related
- Contrary to vectors, matrices “spread” the values over two dimensions: rows and collumns
- Let us go back to the prices at the stores in two cities. It would make more sense to store them in a matrix, instead of two vectors
- Columns could correspond to stores and rows to cities

Matrices (2)

- Let us see how to create this matrix

```
prc <- matrix(c(32.40,35.40,30.20, 35.00, 31.99,
                32.50, 34.60, 32.00, 34.40, 32.01),
              nrow=2,ncol=5,byrow=TRUE)
prc

##       [,1] [,2] [,3] [,4]  [,5]
## [1,] 32.4 35.4 30.2 35.0 31.99
## [2,] 32.5 34.6 32.0 34.4 32.01
```

- The function `matrix()` can be used to create matrices
- We have at least to provide the values and the number of columns and rows

Matrices (3)

```
prc <- matrix(c(32.40,35.40,30.20, 35.00, 31.99,
                32.50, 34.60, 32.00, 34.40, 32.01),
              nrow=2,ncol=5,byrow=TRUE)
prc

##       [,1] [,2] [,3] [,4]  [,5]
## [1,] 32.4 35.4 30.2 35.0 31.99
## [2,] 32.5 34.6 32.0 34.4 32.01
```

- The parameter `nrow` indicates which is the number of rows while the parameter `ncol` provides the number of columns
- The parameter setting `byrow=TRUE` indicates that the values should be "spread" by row, instead of the default which is by column

Indexing matrices

- As with vectors but this time with two dimensions

```
prc

##       [,1] [,2] [,3] [,4]  [,5]
## [1,] 32.4 35.4 30.2 35.0 31.99
## [2,] 32.5 34.6 32.0 34.4 32.01

prc[2, 4]

## [1] 34.4
```

- We may also access a single column or row,

```
prc[1, ]

## [1] 32.40 35.40 30.20 35.00 31.99

prc[, 2]

## [1] 35.4 34.6
```


Giving names to Rows and Columns

- We may also give names to the two dimensions of matrices

```
colnames(prc) <- c("worten","fnac","mediaMkt","radioPop","pixmania")
rownames(prc) <- c("porto","lisboa")
prc

##        worten fnac mediaMkt radioPop pixmania
## porto    32.4 35.4     30.2     35.0    31.99
## lisboa   32.5 34.6     32.0     34.4    32.01
```

- The functions `colnames()` and `rownames()` may be used to get or set the names of the respective dimensions of the matrix
- Names can also be used in indexing

```
prc["lisboa", ]

##   worten     fnac mediaMkt radioPop pixmania
##    32.50    34.60    32.00    34.40    32.01

prc["porto", "pixmania"]

## [1] 31.99
```

Lists

- Lists are ordered collections of other objects, known as the *components*
- List components do not have to be of the same type or size, which turn lists into a highly flexible data structure.
- List can be created as follows:

```
lst <- list(id=12323,name="John Smith",
            grades=c(13.2,12.4,5.6))
lst
```

```
## $id
## [1] 12323
##
## $name
## [1] "John Smith"
##
## $grades
## [1] 13.2 12.4  5.6
```


Indexing Lists

- To access the content of a component of a list we may use its name,

```
lst$grades
```

```
## [1] 13.2 12.4  5.6
```

- We may access several components at the same time, resulting in a sub-list

```
lst[c("name", "grades")]
```

```
## $name
## [1] "John Smith"
##
## $grades
## [1] 13.2 12.4  5.6
```

Data Frames

- Data frames are the R data structure used to store data tables
- As matrices they are bi-dimensional structures
- In a data frame each row represents a case (observation) of some phenomenon (e.g. a client, a product, a store, etc.)
- Each column represents some information that is provided about the entities (e.g. name, address, etc.)
- Contrary to matrices, data frames may store information of different data type

Create Data Frames

- Usually data sets are already stored in some infrastructure external to R (e.g. other software, a data base, a text file, the Web, etc.)
- Nevertheless, sometimes we may want to introduce the data ourselves
- We can do it in R as follows

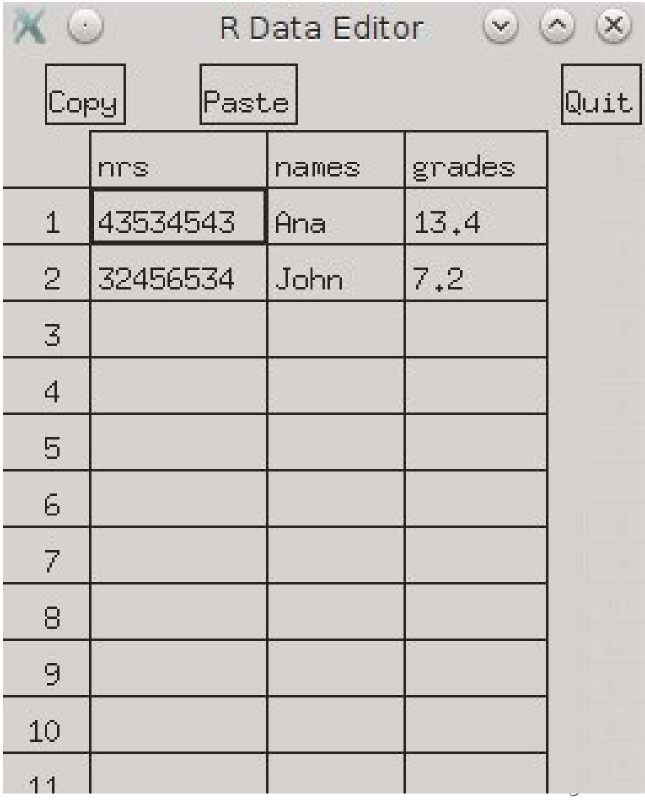
```
stud <- data.frame(nrs=c("43534543","32456534"),
                   names=c("Ana","John"),
                   grades=c(13.4,7.2))
stud
```

```
##        nrs names grades
## 1 43534543   Ana   13.4
## 2 32456534  John    7.2
```

Create Data Frames (2)

- If we have too many data to introduce it is more practical to add new information using a spreadsheet like editor,

```
stud <- edit(stud)
```



Querying the data

- Data frames are visualized as a data table

```
stud

##         nrs names grades
## 1 43534543   Ana   13.4
## 2 32456534  John    7.2
```

- Data can be accessed in a similar way as in matrices

```
stud[2,3]

## [1] 7.2

stud[1,"names"]

## [1] Ana
## Levels: Ana John
```

Querying the data (cont.)

- Function `subset()` can be used to easily query the data set

```
subset(stud,grades > 13,names)

##   names
## 1   Ana

subset(stud,grades <= 9.5,c(nrs,names))

##        nrs names
## 2 32456534  John
```