# BootCamp 2: Basic R object-verse

***Lucien Baumgartner***

***10/2/2018***

In the following I will just quickly run through the most basic object-classes you will encounter in R and how they can be manipulated.

So here we are. What is R all about actually? Well, data analysis, right? Yeah. Kinda. But until you get to crunch actual numbers to compute your fancy analyses, you might want to take a step back. R is based on the language S (no, this is not a joke), which is *semantically open*. This mean that you can express/denote/say the same in different ways. Some ways may be less code-intensive, some may be more efficient, - but in the end this means you can develop your own style of codewriting (YAY). Now, what are we dealing with primarily, if not with numbers? Objects, functions, assignments, and evaluations. That's literally all there is. Everything falls into one or more of those categories.

Let's get down to business then.

## 1 Assignments

Assignments define/create objects. The object to be created is left of the assignment operator (= the object as-a-name), the content of the object (= the object as-it-is) is right of it.

```
x <- 3          # assign a number to an object we call 'x'
x               # evaluate the expression and print result
```

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

Now we evaluate and print an object transformation:

```
x+5
```

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

You can also *store* the transformation by creating a new object…

```
y <- x+5
y
```

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

… or overwrite the current object:

```
x # before the transformation
```

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

```
x <- x+5
x # after the transformation
```

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

You can also let objects interact:

```
x <- 3
y <- x*6
x/y
```

```
## [1] 0.1666667
```

All your objects are stored in your *environment* ( `glovbalenv` ), which is also an object-class. For people with advanced knowledge in R, there is an interesting article on the use of custom environments here (http://adv-r.had.co.nz/Environments.html).

```
ls(globalenv())
```

```
## [1] "x" "y"
```

All objects also appear in your environment pane in RStudio. We will have a lookt at how we can remove objects later.

Now let's have a look at different object classes.

# 2 The main object classes

The main object classes of first order are:

- numeric (numbers like `1.2, 3, -0.12` )
- integer (whole numbers, locked with an L-suffix, like this: `3L, 405L` )

- character (= ‘words’, but really anything between parantheses, be it words, or numbers, or anything else, like `'ice cream', 'a', '1.2', '.', 'TRUE'` . They are evaluated as text.)
- factor (special form of character)

Second-order object classes are specifically structured collections of first order objects:

- vectors (strings of objects, written like `c(1, 3, 1.3)` )
- matrices (combinations of vectors)
- dataframes (matrices with more meta-properties)
- lists (special data storage form)
- arrays (multidimensional-matrices)
- environments (meta-object)

## 2.1 Vectors

The most common objects you will encounter are probably vectors, matrices and dataframes. Second-order objects always inherit the first-order classes from the objects they encompass. For example, a vector containing only character objects, is a charcter vector, one only containing whole numbers will be of both the classes numeric, and vector. To find out the dominant class of an object, we use `class()`

```
class(c(1,3,4))
```

```
## [1] "numeric"
```

We can also check for all the other classes individually:

```
num_vec <- c(1,3,4)
is.vector(num_vec)
```

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

We can also change the classes of object by coering them into another class:

```
char_vec <- as.character(num_vec)
is.vector(char_vec)
```

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

```
is.character(char_vec)
```

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

```
identical(char_vec, num_vec)
```

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

```
char_vec
```

```
## [1] "1" "3" "4"
```

```
num_vec
```

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

Note the parantheses added to the numeric version of the vector, as it is coerced to character.

### 2.1.1 How to create vectors

Other ways to create vectors include:

```
v <- 1:7          # same as c(1,2,3,4,5,6,7)
v <- rep(0, 77)  # repeat zero 77 times: v is a vector of 77 zeroes
v <- rep(1:3, times=2) # Repeat 1,2,3 twice
v <- rep(1:10, each=2) # Repeat each element twice
v <- seq(10,20,2) # sequence: numbers between 10 and 20, in jumps of 2
v1 <- 1:5         # 1,2,3,4,5
v2 <- rep(1,5)    # 1,1,1,1,1
```

### 2.1.2 Different operations with vectors

*Classes define the properties* of objects. For example, numeric transformation can only be performed on numeric or integer objects, but not on characters.

```
# multiply each object in the numeric vector by 2
2*num_vec
```

```
## [1] 2 6 8
```

```
# try the same with a character vector
2*char_vec
```

```
## Error in 2 * char_vec: non-numeric argument to binary operator
```

**Check the length of a vector:**

```
length(v1)
```

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

```
length(v2)
```

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

### Element-wise operations:

```
v1+v2      # Element-wise addition
```

```
## [1] 2 3 4 5 6
```

```
v1+1       # Add 1 to each element
```

```
## [1] 2 3 4 5 6
```

```
v1*2       # Multiply each element by 2
```

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

```
v1+c(1,7)  # This doesn't work: (1,7) is a vector of different length
```

```
## Warning in v1 + c(1, 7): longer object length is not a multiple of short
## object length
```

```
## [1]  2  9  4 11  6
```

### Mathematical operations:

```
sum(v1)      # The sum of all elements
```

```
## [1] 15
```

```
mean(v1)     # The average of all elements
```

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

```
sd(v1)       # The standard deviation
```

```
## [1] 1.581139
```

```
cor(v1,v1*5) # Correlation between v1 and v1*5
```

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

**Logical operations:**

```
v1 > 2       # Each element is compared to 2, returns logical vector
```

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

```
v1==v2       # Are corresponding elements equivalent, returns logical vecto
```

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

```
v1!=v2       # Are corresponding elements *not* equivalent? Same as !(v1==v
```

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

```
(v1>2) | (v2>0)   # | is the boolean OR, returns a vector.
```

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

```
(v1>2) & (v2>0)   # & is the boolean AND, returns a vector.
```

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

```
(v1>2) || (v2>0)  # || is the boolean OR, returns a single value
```

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

```
(v1>2) && (v2>0)  # && is the boolean AND, ditto
```

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

## 2.2 Matrices

You can coerce vectors to two-dimensional objects like matrices and dataframes, containing rows and columns.

```
num_vec1 <- c(1,3,4)
matrix(num_vec1)
```

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

You can combine vectors to matrices by *binding* them together:

```
num_vec2 <- c(5,6,7)
num_mat <- cbind(num_vec1, num_vec2)
is.matrix(num_mat)
```

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

```
num_mat
```

```
##      num_vec1 num_vec2
## [1,]        1        5
## [2,]        3        6
## [3,]        4        7
```

Matrices, in contrast to dataframes, can only contain objects from one first-order class. As soon as a second class is added to a matrix, all columns in the matrix change to the dominant first-order class.

```
is.numeric(num_mat)
```

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

```
# by appedending a character vector to the numeric matrix, we change the wh
mat2 <- cbind(num_mat, char_vec)
is.character(mat2)
```

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

```
mat2
```

```
##      num_vec1 num_vec2 char_vec
## [1,] "1"      "5"      "1"
## [2,] "3"      "6"      "3"
## [3,] "4"      "7"      "4"
```

You might already have noticed that matrices in R are not identical to mathematical matrices who always have the same number of rows and columns.

## 2.3 Dataframes

Dataframes are a special breed of matrices. They, for example, allow for multi-class objects to be stored, preserving their class.

```
df <- data.frame(num_vec1, num_vec2, char_vec, stringsAsFactors = F)
df
```

```
##   num_vec1 num_vec2 char_vec
## 1        1        5        1
## 2        3        6        3
## 3        4        7        4
```

If we look at the structure of `df`, we see that the column `char_vec` is a character, while the others are numeric:

```
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ num_vec1: num  1 3 4
##  $ num_vec2: num  5 6 7
##  $ char_vec: chr  "1" "3" "4"
```

But why did we use the paramater `stringsAsFactors` to `FALSE`? Because dataframes coerce character vector to factors by default. Before we look at factors, we wanna perform index-based data transformations, depending on the specific class of each variable. You can access each and every value stored in the dataframe either by index-position, index-name, or by isolating a variable:

```
# lets multiply the variable/column num_vec1 by 25
df[1]*25 # by column index (number of the specific column)
```

```
##   num_vec1
## 1       25
## 2       75
## 3      100
```

```
df['num_vec1']*25 # by variable name
```

```
##   num_vec1
## 1       25
## 2       75
## 3      100
```

```
df$num_vec1*25 # by variable
```

```
## [1]  25  75 100
```

As you can see, in the first two cases, the resulting object is still a dataframe object (although only single-column); if you isolate a variable, on the other hand, you coerce it back to a vector.

Now if you want to 'save' the transformation in the same object as a replacement to the original variables, you have to overwrite the existing columns. This is done by assigning the transformed values to the same variable/column/vector:

```
df[1] <- df[1]*25 # by column index (number of the specific column)
df['num_vec1'] <- df['num_vec1']*25 # by variable name
df$num_vec1 <- df$num_vec1*25 # by variable

df
```

```
##   num_vec1 num_vec2 char_vec
## 1    15625        5        1
## 2    46875        6        3
## 3    62500        7        4
```

Now, you see that we actually multiplied `num_vec1*25*25*25`, since in each stage we performed a transformation on an already transformed object (except in the first step of course).

Now we create a variable `new_var=num_vec1-num_vec1`:

```
df$new_var <- df$num_vec1-num_vec2
df
```

```
##   num_vec1 num_vec2 char_vec new_var
## 1    15625        5        1   15620
## 2    46875        6        3   46869
## 3    62500        7        4   62493
```

Let's work on the character variable now and paste 'a' as a suffix to the existing characters:

```
df$char_vec
```

```
## [1] "1" "3" "4"
```

```
df$char_vec <- paste0(df$char_vec, 'a')
df
```

```
##   num_vec1 num_vec2 char_vec new_var
## 1    15625        5       1a   15620
## 2    46875        6       3a   46869
## 3    62500        7       4a   62493
```

Now if you want to access a specific subset of you can additonally specify the rownumber in addition to the column-index:

```
# 2nd observation from the first column
df[2, 1] # by column number
```

```
## [1] 46875
```

```
df[2, 'num_vec1'] # by column name
```

```
## [1] 46875
```

```
df$num_vec1[2] # by variable
```

```
## [1] 46875
```

Of course, you can also access several columns for every observation:

```
# variable num_vec1 and char_vec only for 2nd observation
df[2, c(1, 3)] # by column number
```

```
##   num_vec1 char_vec
## 2    46875       3a
```

```
df[2, c('num_vec1', 'char_vec')] # by column name
```

```
##   num_vec1 char_vec
## 2    46875       3a
```

```
# by variable - makes no sense here

# all variables only for 2nd observation
df[2, ]
```

```
##   num_vec1 num_vec2 char_vec new_var
## 2    46875        6       3a   46869
```

## 2.4 Lists

Lists are collections of objects. A single list can contain all kinds of elements - character strings, numeric vectors, matrices, other lists, and so on. The elements of lists are often named for easier access.

```
l1 <- list(boo=v1,foo=v2,moo=c(1.2),zoo="Animals!")  # A list with four com
l1
```

```
## $boo
## [1] 1 2 3 4 5
##
## $foo
## [1] 1 1 1 1 1
##
## $moo
## [1] 1.2
##
## $zoo
## [1] "Animals!"
```

```
l2 <- list(v1,v2,"Animals!")  # A list with three components
l2
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] 1 1 1 1 1
##
## [[3]]
## [1] "Animals!"
```

Create an empty list:

```
l3 <- list()
l3
```

```
## list()
```

```
l4 <- NULL
l4
```

```
## NULL
```

Accessing list elements:

```
l1["boo"]   # Access boo with single brackets: this returns a list.
```

```
## $boo
## [1] 1 2 3 4 5
```

```
l1[["boo"]] # Access boo with double brackets: this returns the numeric vec
```

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

```
l1[[1]]     # Returns the first component of the list, equivalent to above.
```

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

```
l1$boo      # Named elements can be accessed with the $ operator, as with [
```

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

Adding more elements to a list:

```
l3[[1]] <- 11 # add an element to the empty list l3
l4[[3]] <- c(22, 23) # add a vector as element 3 in the empty list l4.
```

Since we added element 3 to the list l4above, elements 1 and 2 will be generated and empty (NULL).

```
l1[[5]] <- "More elements!" # The list l1 had 4 elements, we're adding a 5t
l1[[8]] <- 1:11
```

We added an 8th element, but not 6th and 7th to the listl1 above. Elements number 6 and 7 will be created empty (NULL).

```
l1$Something <- "A thing"  # Adds a ninth element - "A thing", named "Somet
```

# 3 Value comparisons and special constants

Special constants include:

- `NA` for missing or undefined data
- `NULL` for empty object (e.g. null/empty lists)
- `Inf` and `-Inf` for positive and negative infinity
- `NaN` for results that cannot be reasonably defined

```
# NA - missing or undefined data

5+NA      # When used in an expression, the result is generally NA
```

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

```
is.na(5+NA) # Check if missing
```

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

```
# NULL - an empty object, e.g. a null/empty list

10+NULL     # use returns an empty object (length zero)
```

```
## numeric(0)
```

```
is.null(NULL) # check if NULL
```

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

Inf and -Inf represent positive and negative infinity. They can be returned by mathematical operations like division of a number by zero:

```
5/0
```

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

```
is.finite(5/0) # Check if a number is finite (it is not).
```

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

NaN (Not a Number) - the result of an operation that cannot be reasonably defined, such as dividing zero by zero.

```
0/0
```

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

```
is.nan(0/0)
```

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

## 4 Special case: factors

So now on to factors. Factors are special cases of character vectors. Their little twist is that they additionally provide *levels*, this is all unique tokens in the character string. lots of blah, this is how it looks:

```
char_str <- c('CH', 'DE', 'CH', 'IT', 'IT', 'CH', 'FR', 'ES')
char_str
```

```
## [1] "CH" "DE" "CH" "IT" "IT" "CH" "FR" "ES"
```

```
# coerce to factor
fctr_str <- factor(char_str)
fctr_str
```

```
## [1] CH DE CH IT IT CH FR ES
## Levels: CH DE ES FR IT
```

CH is mentioned multiple times, but only generates one level, so that levels are identical to groups in your data. Factor variables can be useful for grouping/aggregating data, as well as plotting.

## 5 Removing objects

you want to remove objects, just use the `rm` -function:

```
rm(x)
```

Once this has been evaluated, you will not be able to call the removed object anymore:

```
x
```

```
## <simpleError in doTryCatch(return(expr), name, parentenv, handler): obje
```

If you want to remove several objects, you can just list the objects:

```
x <- 3 # create object x again
rm(x, y)
```

Whenever you want to start from scratch (usually when you start a new R script!), you may want to clean your whole environment. Most scripts should include this line in the header to avoid working with objects that are not actually generated by that specific script (and also to avoid any accidential objectification of variables, but we get to that later).

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

What does it do? - `ls()` is a function to list all objects in your environment. It denotes the same as `ls(globalenv())`, so it is semantically identical ( == ).

```
# create objects again (they have been removed by rm(list=ls()), remember?)
x <- 3
y <- x*6

# list objects
ls()
```

```
## [1] "x" "y"
```

```
ls()==ls(globalenv())
```

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

The way the objects are listed is in an of itself an object again, namely a character vector, this is a vector containing one or more character objects. Here, the character objects are the names of the objects `x` and `y`, in this case `'x'`, `'y'` (!parantheses!). How do we prove it is actually a character vector? We can use the functions `is.vector()` and `is.character()` to evaluate whether the objects in the function are vector, or characters, respectively. The output of the function itself is a *logical* (object).

```
is.vector(ls()) # proof that the function ls() equates to a vector
```

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

```
is.character(ls()) # proof that the function ls() equates to a character ob
```

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

Or simply:

```
is.vector(ls())&is.character(ls())
```

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

So why do we equate `ls()` to `list` in `rm(list=ls())` ? Because this is how we tell the function what to delete. Hm... But before that we could just call `rm(x, y)` to delete objects, *directly calling them* (without having to use their name). Yes, that is true, BUT you could also have written the exact same by using `rm(list=c('x', 'y'))` , where `c()` creates a vector. Why is this? The explanation can be found in the documentation of the `rm` -function which can be called prefixing the function with a question mark ( `?rm` [without brackets!]). Below you see a print of the documentation. The only thing that we are interested in right now is the **Usage** section. There you can see *what arguments can be passed to the function and how*. Ther we see that we `...` just means `the objects to be removed, as names (unquoted) or character strings (quoted)`. This is what we did in `rm(x, y)` which, hence, is a specification of `rm(...)` . Alternatively, you can also pass the objects via `list = character()` , where the parameter `list` is defined as `a character vector naming objects to be removed`. This is what we did in `rm(list=c('x', 'y'))` and what we do in `rm(list=ls())` .

```
?rm
```

```
remove {base}   R Documentation
Remove Objects from a Specified Environment

Description

remove and rm can be used to remove objects. These can be specified successi

If envir is NULL then the currently active environment is searched first.

If inherits is TRUE then parents of the supplied directory are searched unti

Usage

remove(..., list = character(), pos = -1,
       envir = as.environment(pos), inherits = FALSE)

rm    (..., list = character(), pos = -1,
       envir = as.environment(pos), inherits = FALSE)
Arguments

...
the objects to be removed, as names (unquoted) or character strings (quoted
list
a character vector naming objects to be removed.
pos
where to do the removal. By default, uses the current environment. See ‘deta
envir
the environment to use. See ‘details’.
inherits
should the enclosing frames of the environment be inspected?
Details

The pos argument can specify the environment from which to remove the objec

It is not allowed to remove variables from the base environment and base nam

Earlier versions of R incorrectly claimed that supplying a character vector

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language. W

See Also

ls, objects

Examples
```

```
tmp <- 1:4
## work with tmp  and cleanup
rm(tmp)

## Not run:
## remove (almost) everything in the working environment.
## You will get no warning, so don't do this unless you are really sure.
rm(list = ls())

## End(Not run)
[Package base version 3.5.1 Index]
```

[Some parts of this intro were copied from: http://kateto.net/networks-r-igraph (http://kateto.net/networks-r-igraph).]