Basic Course on R: Objects and Functions

Karl Brand* and Elizabeth Ribble†

28 Oct - 1 Nov 2019

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Things you can have: object classes and modes

Any *thing* in `R` is of a certain type, defined as a `class` and `mode`. As an analogy, think of numbers as *fruits* and text as *vegetables*. They're not the same, although they can be analysed, or 'cooked' like one another.

1.1 Numbers

Fruits.

```
class(1)
```

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

```
mode(1)
```

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

```
class(2.3123)
```

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

```
an_integer <- as.integer(2.3123)
an_integer
```

```
## [1] 2
```

```
class(an_integer)
```

```
## [1] "integer"
```

```
mode(an_integer)
```

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

Change the class to numeric:

```
numeric_again <- as.numeric(an_integer)
class(numeric_again)
```

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

1.2 Text

The *veggies*.

```
class("a")
```

```
## [1] "character"
```

```
mode("a")
```

```
## [1] "character"
```

```
class("1")
```

```
## [1] "character"
```

```
mode("1")
```

```
## [1] "character"
```

```
a_factor <- factor("a")
a_factor
```

```
## [1] a
## Levels: a
```

```
class(a_factor)
```

```
## [1] "factor"
```

```
mode(a_factor)
```

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

1.3 Logical

It can only be a *yes* or a *no*. More specifically, a `TRUE` or a `FALSE`.

```
class(TRUE)
```

```
## [1] "logical"
```

```
class(FALSE)
```

```
## [1] "logical"
```

```
mode(TRUE)
```

```
## [1] "logical"
```

1.4 Functions

An object that does something to the fruits and veggies, a *frying pan!*

2 Containers: more object classes and modes

Besides the basic types of things, there are different *containers* available to hold these things which are also defined by `class` and `mode`. Containers are thus things, or `objects`, that you can put other things, or `objects`, into to conveniently hold them while we work with them.

2.1 Vectors are an ordered series of elements

Think of them like a *string* or a *chain*.

```
a_vector <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
class(a_vector)
```

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

```
mode(a_vector)
```

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

```
length(a_vector)
```

```
## [1] 12
```

```
ano_vector <- c(1)
length(x = ano_vector)
```

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

Combine vectors to make another vector:

```
combined_vec <- c(a_vector, ano_vector)
combined_vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12  1
```

2.2 Matrices are vectors with two dimensions

Matrices are like a spread sheet in that they have rows and columns. In addition the contents have a specific, column on column order or sequence exactly equivalent to a

vector. Think of them like a *string on a tray*. Matrices consist of only one type or *class* of data.

```
a_matrix <- matrix(data = a_vector)
a_matrix
```

```
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
```

```
a_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
dim(a_matrix)
```

```
## [1] 12  1
```

```
ano_matrix <- matrix(data = a_vector, nrow = 3)
ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
dim(ano_matrix)
```

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

```
letters_mat <- matrix(data = letters[1:12], nrow = 3)

class(a_matrix)
```

```
## [1] "matrix"
```

```
mode(a_matrix)
```

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

```
class(letters_mat)
```

```
## [1] "matrix"
```

```
mode(letters_mat)
```

```
## [1] "character"
```

2.3 Arrays are matrices with multiple dimensions

```
an_array <- array(letters[1:12], c(2, 2, 3))
an_array
```

```
## , , 1
##
##      [,1] [,2]
## [1,] "a"  "c"
## [2,] "b"  "d"
##
## , , 2
##
##      [,1] [,2]
## [1,] "e"  "g"
## [2,] "f"  "h"
##
## , , 3
##
##      [,1] [,2]
## [1,] "i"  "k"
## [2,] "j"  "l"
```

```
class(an_array)
```

```
## [1] "array"
```

```
mode(an_array)
```

```
## [1] "character"
```

2.4 Lists hold just about anything and everything

Similar to vectors, lists have a specific order or sequence. Unlike vectors however, each element of a list can be of any `class` or `mode`. Think of them like a *shopping list* of: a string of fruit, a tray of veggies, another sublist of items

```r
a_list <- list(a_vector, a_matrix, letters_mat)
a_list
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## [[2]]
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
##
## [[3]]
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "d"  "g"  "j"
## [2,] "b"  "e"  "h"  "k"
## [3,] "c"  "f"  "i"  "l"
```

```r
class(a_list)
```

```
## [1] "list"
```

```r
mode(a_list)
```

```
## [1] "list"
```

```r
named_list <- list("vec" = a_vector, "mat" = a_matrix,
                   "lets" = letters_mat,
                   "log" = matrix(rep(c(TRUE, FALSE), 5), nrow = 5),
                   "lis" = list(1:5, letters[1:9]))
named_list
```

```
## $vec
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## $mat
```

```
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
##
## $lets
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "d"  "g"  "j"
## [2,] "b"  "e"  "h"  "k"
## [3,] "c"  "f"  "i"  "l"
##
## $log
##       [,1]  [,2]
## [1,]  TRUE FALSE
## [2,] FALSE  TRUE
## [3,]  TRUE FALSE
## [4,] FALSE  TRUE
## [5,]  TRUE FALSE
##
## $lis
## $lis[[1]]
## [1] 1 2 3 4 5
##
## $lis[[2]]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
class(named_list)
```

```
## [1] "list"
```

```
mode(named_list)
```

```
## [1] "list"
```

```
named_list[["log"]]
```

```
##       [,1]  [,2]
## [1,]  TRUE FALSE
## [2,] FALSE  TRUE
## [3,]  TRUE FALSE
## [4,] FALSE  TRUE
## [5,]  TRUE FALSE
```

```
class(named_list[["log"]])
```

```
## [1] "matrix"
```

```
mode(named_list[["log"]])
```

```
## [1] "logical"
```

2.5 Data frames are special lists

Data frames are even more similar to the familiar excel spreadsheets: a series, or `list` of equal length columns, or `vectors`. Notably, the columns (`vectors`) can be of different classes, unlike a matrix or array.

```r
aDF <- data.frame("vec" = a_vector, "lets" = letters[1:12])
aDF
```

```
##    vec lets
## 1    1    a
## 2    2    b
## 3    3    c
## 4    4    d
## 5    5    e
## 6    6    f
## 7    7    g
## 8    8    h
## 9    9    i
## 10  10    j
## 11  11    k
## 12  12    l
```

```r
dim(aDF)
```

```
## [1] 12  2
```

```r
class(aDF)
```

```
## [1] "data.frame"
```

```r
mode(aDF)
```

```
## [1] "list"
```

```r
str(aDF)
```

```
## 'data.frame': 12 obs. of  2 variables:
##  $ vec : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ lets: Factor w/ 12 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10 ...
```

```r
aDF[, 1]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```r
class(aDF[, 1])
```

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

```r
mode(aDF[, 1])
```

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

```r
aDF[, 2]
```

```
##  [1] a b c d e f g h i j k l
```

```
## Levels: a b c d e f g h i j k l
class(aDF[, 2])
## [1] "factor"
mode(aDF[, 2])
## [1] "numeric"
str(aDF[, 1])
##  num [1:12] 1 2 3 4 5 6 7 8 9 10 ...
```

It's important to understand that the things we work with in `R` have a 'mode' and a 'class'. The mode is a mutually exclusive classification of an object's basic structure and the class is a property used to determine how functions operate with object. Pay attention to your modes and classes.

3 Functions are things you can do

`R` comes with predefined functions which do many things from basic file management to complex statistics. To get started, below are some oft used functions. This default set of functions is easily extended by defining your own functions and adding those defined by others conveniently in CRAN and BioConductor packages:

`http://stat.ethz.ch/R-manual/R-patched/library/base/html/00Index.html`

`http://cran.r-project.org/web/packages/available_packages_by_name.html`

`http://www.bioconductor.org/packages/release/bioc/`

Functions are expressed as:

> `function.name()`, e.g., `t.test()`
>
> or,
>
> an operator, e.g., `+`

Note that operators can be expressed in `function.name()` format by enclosing in double quotes, e.g. `"+"()`. This is occasionally required for some expressions.

Easily obtain functions from other `R` users using `install.packages()`:

```
install.packages("packageName",
                 lib = "/directory/to/my custom R library",
                 repos = "http://cran.xl-mirror.nl")
```

The package name must be quoted when installing. Besides installing the package on your PC, you need to load it into your R session before you can use it:

```
library("packageName")      ## quotes are optional when loading a package
```

3.1 The Arithmetic Operators

That is, R as a *calculator*.

```
x <- 10
y <- 3
x + y
```

```
## [1] 13
```

```
x - y
```

```
## [1] 7
```

```
x * y
```

```
## [1] 30
```

```
x / y
```

```
## [1] 3.333333
```

```
x ^ y           # exponentiation
```

```
## [1] 1000
```

```
x %% y          # modular arithmetic, remainder after division
```

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

```
x %/% y         # integer part of a fraction
```

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

How these functions work on vectors:

```
a_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

3.1.1 By the way, R is Green

By design, R *recycles* data without telling you, at least when the recycling fits neatly.

```
a_vector + x                    # x is recycled without warning
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20 21 22
```

Operators can also be used in the `function.name()` format. Note the quotes:

```
"+"(a_vector, x)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20 21 22
```

```
a_vector - y
```

```
##  [1] -2 -1  0  1  2  3  4  5  6  7  8  9
```

```
a_vector + a_vector
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24
```

```
vec_of_thr <- c(2, 4, 6)
a_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
a_vector + vec_of_thr           # recycled without warning
```

```
##  [1]  3  6  9  6  9 12  9 12 15 12 15 18
```

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

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

When it doesn't fit neatly, you'll get a warning:

```
a_vector + vec_of_fi            # warning
```

```
## Warning in a_vector + vec_of_fi:  longer object length is not a multiple
of shorter object length
```

```
##  [1]  2  4  6  8 10  7  9 11 13 15 12 14
```

```
## Warning message:
## Package xcolor Warning: Incompatible color definition on input line 119.
## Package xcolor Warning: Incompatible color definition on input line 132.
## Package xcolor Warning: Incompatible color definition on input line 145.
## Package xcolor Warning: Incompatible color definition on input line 158.
## Package xcolor Warning: Incompatible color definition on input line 172.
## Package xcolor Warning: Incompatible color definition on input line 187.
```

```
## Package xcolor Warning: Incompatible color definition on input line 201.
## Package xcolor Warning: Incompatible color definition on input line 224.
## Package xcolor Warning: Incompatible color definition on input line 254.
## Package xcolor Warning: Incompatible color definition on input line 269.
## Package xcolor Warning: Incompatible color definition on input line 284.
## Package xcolor Warning: Incompatible color definition on input line 297.
## Package xcolor Warning: Incompatible color definition on input line 342.
## Package xcolor Warning: Incompatible color definiti [... truncated]
```

How these functions work on matrices:

```
ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
ano_matrix + x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   14   17   20
## [2,]   12   15   18   21
## [3,]   13   16   19   22
```

```
ano_matrix - y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   -2    1    4    7
## [2,]   -1    2    5    8
## [3,]    0    3    6    9
```

```
ano_matrix + ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    8   14   20
## [2,]    4   10   16   22
## [3,]    6   12   18   24
```

```
ano_matrix * ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   16   49  100
## [2,]    4   25   64  121
## [3,]    9   36   81  144
```

3.2 Functions you'll use to interact with data

Relational Operators:

```
x < y
## [1] FALSE
x > y
## [1] TRUE
x <= y
## [1] FALSE
x >= y
## [1] TRUE
x == y
## [1] FALSE
x != y
## [1] TRUE
```

3.3 Functions you need to get work done with R

R has an extensive set of built-in ***functions***, a few of which are listed below:

Print structure of an object:

```
str()
```

Print structure of an object:

```
class()
```

First six elements/rows:

```
head()
```

Last six elements/rows:

```
tail()
```

List all objects and functions held in your global environment:

```
ls()
```

Generate a sequence:

```
seq(from = 1, to = 10, by = 2)
```

Or, use the `:` operator:

```
1:10
```

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

Run the entire contents of a script:

```
source("myScript.R")
```

Each function accepts one or more values passed to it as ***arguments***, performs computations or operations on those values, and returns a result.

To perform a ***function call***, type the name of the function with the values of its argument(s) in parentheses, then hit 'Enter'. For example:

```
sqrt(2)
```

```
## [1] 1.414214
```

Values passed as arguments can be in the form of variables, such as `x` below:

```
x <- 2
sqrt(x)
```

```
## [1] 1.414214
```

or they can be entire expressions, such as `x^2 + 5` below:

```
sqrt(x^2 + 5)
```

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

3.4 Viewing a Function's Formal Arguments

Most functions take multiple arguments, each of which has a name, and some of which are optional.

One way to see what arguments a function takes and which ones are optional is to use the function:

`args()`

Another way to view a function's arguments is to look at its help file (e.g. `?sqrt`).

For example, to see what arguments `round()` takes (using `args()`), we'd type:

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

We see that `round()` has two arguments, `x`, a numeric value to be rounded, and `digits`, an integer specifying the number of decimal places to round to. Thus to round 4.679 to 2 decimal places, we type:

```
round(4.679, 2)
```

```
## [1] 4.68
```

3.5 Optional Arguments and Default Values

The specification `digits = 0` in the output from `args(round)` tells us that `digits` has a ***default value*** of 0. This means that it's an ***optional argument*** and if no value is passed for that argument, rounding is done to 0 decimal places (i.e. to the nearest integer).

3.6 Positional Matching and Named Argument Matching

When we type

```
round(4.679, 2)
```

```
## [1] 4.68
```

R knows, by ***positional matching***, that the first value, 4.679, is the value to be rounded and the second one, 2, is the number of decimal places to round to.

We can also specify values for the arguments by name. For example:

```
round(x = 4.679, digits = 2)
```

```
## [1] 4.68
```

When ***named argument matching*** is used, as above, the order of the arguments is irrelevant. For example, we get the same result by typing:

```
round(digits = 2, x = 4.679)
```

```
## [1] 4.68
```

The two types of argument specification (positional and named argument matching) can be mixed in the same function call.

3.7 Formal Arguments and Actual Arguments

In the function call

```
round(x = 4.679, digits = 2)
```

the values `4.679` and `2` passed to `round()` are referred to as ***actual arguments***.

The ***formal arguments*** for the function `round()` are `x` and `digits`, to which we pass the actual values `4.679` and `2`.

Look at the arguments for the function `signif()`:

```
args(signif)

## function (x, digits = 6)
## NULL
```

`signif()` prints the value passed for `x` to the number of significant digits specified by `digits`.

3.8 Variable Number of Arguments Using "..."

- Functions can be written to take a variable number of arguments. The argument name "..." in the function definition will match any number of arguments.
- For example, here's a function that returns the mean of all the values in an arbitrary number of vectors:

  ```
  meanOfAll <- function(...) {
    return(mean(c(...)))
  }
  ```

  The command

  ```
  meanOfAll(usSales, europeSales, otherSales)
  ```

  would combine the three vectors and take the mean of all the data. The effect of `c(...)` is as if `c()` were called with the same three arguments passed to `meanOfAll()`.
- Many of R's built-in functions take a variable number of arguments. For example look at the help files for `list()` and `c()`.

4 Saving and restoring your session

Because the typical way of using `R` involves writing text into a file with the `.r` or `.R` extension it's natural to save your commands written for a specific purpose in this `my-r-file.r`. Hopefully you're doing this right now. However there are other bits and pieces of your `R` session described below you may want or need to save.

Save all objects to the working directory in a file called `.RData`:

```
save.image()
```

Or give the file a name:

```
save.image("mySession.RData")
```

Save all commands entered into the R console during your session to the working directory in a file called .Rhistory:

```
savehistory()
```

Such a file may be hidden by default in some file browsers, including linux.

Note that this typically occurs (console dependent) by default when you quit your R session. If an .Rhistory file already exists from a previous session, the current session is appended to the end of this file and saved. Thus, by design, a complete log of your work is saved together with your script file, if your console session contains less than 512 lines.

Save specific objects:

```
save(object_1, object_2, object_3, file = "rObjects123.RData")
```

Restore or load previous sessions or objects:

```
load("mySession.RData")         # load from the working directory
```

Load .Rhistory file to access the history from the console:

```
loadhistory()
```

5 Miscellaneous Tips

R is case sensitive:

```
c("Hello" == "hello", "goodbye" == "goodbye")
```

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

And is *very* sensitive in general:

```
# c("Hello" == "hello" "goodbye" == "goodbye") # not run
```

Missing data. NA 'Not Available' is how R defines a missing value i.e., an empty cell in excel

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

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

R is an environment. What's in yours?

```
ls()                                    # global environment
objects()                               # alias for ls()
```

Too many objects cluttering up your environment and computer memory? Remove the ones you don't need:

```
rm(object_name)
```

What is the current *working directory* where files are saved and loaded from by default?

```
getwd()
```

Need to set or change the current working directory?

```
setwd(dir = "c:\\Windows\\Users\\karl\\My Documents\\R stuff")
setwd(dir = "c:/Windows/Users/karl/My Documents/R stuff")
```

Note the double backslashes required by `R` running under Windows. Single forward slashes, the Unix convention works equivalently.

Also be aware that if you started your `R` session by double clicking a `file.name.r` file, by design the directory where this file resides is set as the working directory. This directory is also a good place, and the default place, to store the respective `.RData` and `.Rhistory` files if you have any. When these files are present in the same directory they will be loaded automatically, unless you explicitly tell `R` not to load them by using the `---no-restore-data` argument when starting `R`.

Quit your `R` session:

```
q()                                  # not run
```

Object names can not begin with numbers. For example:

```
## 3vector <- c(2, 4, 6))     # not run
```

But can end with numbers if necessary:

```
vector3 <- c(2, 4, 6)
```

Try to code with style, for example:

`https://google.github.io/styleguide/Rguide.xml`

`http://adv-r.had.co.nz/Style.html`

6 Document License

GNU General Public License v2.0 or higher (GPL>=v2)