# Advanced syntax and computationS in R

Antonio Gasparrini

01 November 2023

## Contents



---

This session introduces more advanced computational methods available in R, including the use of more complex syntactic terms to perform specific tasks. The content highlights the flexibility of the R software and language for obtaining complex computations using plain and short coding. This provides a more flexible framework for data analysis, although not easily approachable by beginners, as the various topics introduced here make use of sophisticated features of the R language. The first section starts with the illustration of *vectorization* and *recycling rules*, a key computational aspect of R. The second section describes the use of *replacement expressions*, while the third one introduces the *pipe*, two methods that facilitate complex computations. The last section offers a summary of the *apply family* of functions as flexible computational tools.

---

## Vectorization and recycling rule

A very important feature of R is *vectorization*, a property of functions which act element-wise to vectors. Vectorization brings a substantial increase in the computational speed, and allows a compact and clear-cut syntax for otherwise intricate coding. Several functions in R are vectorized, for instance, most of the arithmetic, logical or relational operators used in basic computations. As an example, we can see how this process applies to computations using simple vectors. First, we create them:

```
numvec1 <- c(4,2,-7,16,-2)
numvec2 <- c(3,0,0,-2,14)
```

And then we apply simple arithmetic and logical operators:

```
numvec1 * 2
```

```
## [1]   8   4 -14  32  -4
```

```
numvec2 > 0
```

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

In vectorized computations, the function is internally applied to all the elements of the vector. Some functions are not vectorized by definition, as they are based on a computation involving all the elements of the vector. Examples are the functions `max()`, `min()`, `mean()`, or `sum()`, among many others, which return a single number. Sometimes vectorized extensions of such functions are available. For instance, parallel maxima of two or more vectors are computed by `pmax()`. Let's compare the results with the non-vectorized counterpart:

```
max(numvec1, numvec2)
```

```
## [1] 16
```

```
pmax(numvec1, numvec2)
```

```
## [1]  4  2  0 16 14
```

While `max()` returns the single highest number among all the elements of all the vectors, the result of the vectorized `pmax()` is a vector of the same length of the arguments where each element is the maximum among the elements with the same index of the other vectors. The function `pmin()` provides the same for the minimum. Interestingly, all these functions work with matrices and character objects as well.

So far, we have performed computations between vectors of the same length or between vectors and scalars. Actually, this is not a requirement, and in most cases, the computations in R can be carried out with vectors of different lengths. Let's see an example:

```
rep(3,8) * 1:2
```

```
## [1] 3 6 3 6 3 6 3 6
```

What does happen here? The two expressions in the two sides of the multiplication operator create vectors with lengths equal to 8 and 2, respectively. In order to make them consistent with the element-by-element computation, the second vector is repeated again until it matches the length of the first one, and then the operation is carried out. Such action is determined by the *recycling rule*.

In fact, a computation between a vector and a scalar is another example of recycling. For instance, in the expression `numvec1 * 2` at the beginning of this section, the scalar `2` is treated as a vector of length 1, and recycled accordingly, *i.e.* repeated five times in that specific case.

The idea of vectorization, *i.e.* applying the computation of a function sequentially to multiple elements, as well as the recycling rule, extends to more complex data structures than vectors, such as matrices, lists, or data frames. For example, if you want to create a matrix with all the three rows equal to a specific vector, you can input:

```
matrix(numvec1, nrow=3, ncol=length(numvec1), byrow=TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    4    2   -7   16   -2
## [2,]    4    2   -7   16   -2
## [3,]    4    2   -7   16   -2
```

Here, the dimensions of the matrix are fixed, and the vector `numvec1` is recycled until the matrix is complete. Another example is with data frames:

```
data.frame(id=1:4, level=c(2,1,6,4), country="UK")
```

```
##   id level country
## 1  1     2      UK
## 2  2     1      UK
## 3  3     6      UK
## 4  4     4      UK
```

Here we assume that all the records are collected in the UK, whose label inputted as a character vector of length 1 is recycled as necessary.

The recycling rule is a powerful tool for producing short and clear code. See this example on how to select the odd numbers between 11 and 20:

```
(11:20)[c(T,F)]
```

```
## [1] 11 13 15 17 19
```

Here the index vector produced by `c(T,F)` is recycled five times, with the resulting `TRUE` corresponding to indices 1, 3, 5, 7 and 9. Also, note how the brackets are required to give priority to the colon operator '`:`' before the indexing operator'`[`'.

However, you should pay attention as the recycling rule can also cause mistakes if the computation involves vectors of different lengths which are not supposed to be recycled. In particular, R returns a warning when the length of the longer object is not a multiple of that of the shorter one. In other cases, for the application of specific functions, inconsistent dimensions are not allowed, and an error is instead returned. However, when the dimensions are consistent, you are not informed about the recycling being applied even when the computation is in fact wrong.

## Replacement expressions

The functional style of R implies that all computations are performed by the application of functions to existing objects, and that these actions do not modify the state and environment. Objects can be created by assignments through the operator `->`, and existing objects can be re-assigned a new value by a new assignment to the same name or erased by using the function `rm()`. However, these tools are limited, as for instance they cannot be applied to modify existing objects, which is a common programming task.

In order to partly change existing objects, we can rely on *replacement expressions.* These are special assignment operations in which the left-hand side of the expression involving the operator `->` , instead of a name, is a call to a function identifying a part or characteristic of the object.

An example will clarify the concept. Let's take the vector `numvec1` created in the previous section, and assume we want to replace the negative values to 0. Rather than re-creating the object from scratch, we can replace only these specific values using:

```
numvec1
```

```
## [1]  4  2 -7 16 -2
```

```
numvec1[numvec1 < 0] <- 0
numvec1
```

```
## [1]  4  2  0 16  0
```

We can see that the negative values in the third and fifth positions have been replaced by `0`. In the expression above, the assignment only applies to the part of the object defined by the indexing operation.

Replacement expressions are a flexible method for manipulating objects, and can be used also with more complex data structures. For example, let's use it with a newly created list:

```
mylist <- list(vec1=numvec1, vec2=numvec2)
```

We can now add a new component to the list by using a replacement expression:

```
mymat <- matrix(c(3,1,7,4,2,2,5,1), nrow=2)
mylist$mat <- mymat
```

In this case, the replacement expression involves the operator '`$`' to identify/extract a specific component, which is in this case is assigned to the object. You can now check that a new component has been added:

```r
names(mylist)
```

```
## [1] "vec1" "vec2" "mat"
```

Similarly, existing components can be replaced, or otherwise removed by assigning them to `NULL`. For instance:

```r
mylist$vec1 <- NULL
names(mylist)
```

```
## [1] "vec2" "mat"
```

You can appreciate that replacement expression offers a flexible and powerful computational method, using a simple and direct syntax to perform potentially complex computational tasks. The left-hand side can be any call involving any function or operator that selects part of an object. The replacement expression is actually a call to internal *replacement functions* specific to this purpose.

## Pipes

The functional style of R offers a flexible computing environment where a complex set of operations can be performed by nesting multiple function calls. Let's see an example:

```r
mean(log(runif(8, 0, 10)))
```

```
## [1] 1.216181
```

Here, we first sample a draw of eight observations from a uniform distribution between 0 and 10 (see `help(runif)`), then we take their log, and finally we compute their average. This approach of using calls to functions as arguments for other functions offers a way to produce complex computations in a compact code.

However, there are drawbacks. First, the syntax becomes more intricate by increasing the levels of nesting operations, with the use of multiple brackets. The code is then harder to read and prone to errors. More importantly, the order of the operation in the syntax, from outer to inner, proceeds in the opposite sense, starting with a call to the last operation (`mean()`) to the first one (`runif()`) while going from outer to inner. An alternative way is to separate the three computations by assigning intermediate objects, which of course it is not ideal.

A more natural syntax can be implemented by using the *pipe* operator, defined by the symbol '`|>`' (an alternative operator '`%>%`' is available in other packages). Pipes offer a way to nest operations using their *natural* order. For instance, the set of operations above can be performed by:

```r
runif(8, 0, 10) |> log() |> mean()
```

```
## [1] 1.579072
```

While the results are different due to the random sampling, the two procedures are identical. Here, the result of the call on the left-hand side of the pipe operator '`|>`' is used as the (first) argument of the call on the right-hand side. This provides a simple way to nest an indefinite number of operations while using a clean and easily readable syntax.

However, consider that the pipe operator does not involve complex computations, but it simply implements a syntax transformation. In fact, the expression is parsed internally to the usual outer-to-inner syntax. Let's see that the two expressions above are eventually identical, by using the function `quote()` (see `help(quote)`):

```r
quote(runif(8, 0, 10) |> log() |> mean())
```

```
## mean(log(runif(8, 0, 10)))
```

By default, the left-hand side expression in a pipe is always passed to the first argument of the function call on the right-hand side. However, additional arguments can be specified. For example, let's use pipes to inspect the first three rows of the dataset MTCARS available in the DATASET package:

```
mtcars |> head(3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

Here, `3` is assigned automatically to the second (unnamed) argument of `head()`. Named arguments can be used to be more specific about the assignment. When the left-hand side expression must be assigned to a different argument, the *placeholder* symbol '_' can be used. For instance:

```
3 |> head(mtcars, n=_)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

Alternatively, arguments (including the first one) can be named, and the left-hand side expression is assigned to the first available argument:

The placeholder can be used also as the first argument in an extraction call using the operators '$' or '[]'. For example, let's use pipes to extract the first three rows of `mtcars`, select the variable `mpg` (miles per gallon), and then compute its average:

```
mtcars |> head(3) |> _$mpg |> mean()
```

```
## [1] 21.6
```

The examples above are a bit silly, but they can illustrate the potential of pipes in defining a complex set of operations using clear-cut syntax.

## The apply family

A set of functions grouped in the *apply family* offers advanced computational tools. The common feature of this family is to apply existing functions repeatedly in groups of observations, defined in different ways, to perform specific computations.

The first function presented here is `apply()`, which is used to obtain marginal results over the dimensions of a matrix or array. In the simpler case of a matrix, the computation is performed over rows or columns. This function generalises the use of `rowMeans()` and `rowSums()` (and `col-` counterparts) by allowing the application of statistics beyond mean and total.

As an example, we use it to derive the standard deviation of each row and column of the matrix `mymat` created in a previous section:

```
mymat
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3    7    2    5
## [2,]    1    4    2    1
```

```
apply(mymat, MARGIN=1, FUN=sd)
```

```
## [1] 2.217356 1.414214
```

```
apply(mymat, MARGIN=2, FUN=sd)
```

```
## [1] 1.414214 2.121320 0.000000 2.828427
```

The first argument of `apply()` is a matrix or array object, or an object that can be coerced as such, for example, a data frame. The arguments `MARGIN` and `FUN` determine the dimension over which marginal computations must be produced (`1` for rows, `2` for columns) and the function to be applied, respectively.

Among the arguments of this function (see the help page using `help(apply)`), there is a special one called *ellipsis,* defined by the three-dot symbol '…'. The ellipsis, featuring in many other functions in R, can include an indefinite number of arguments to be passed internally to other functions.

Let's see an example with `apply()`:

```
apply(mymat, 1, quantile, probs=0.25)
```

```
## [1] 2.75 1.00
```

The expression above applies the function `quantile()` to compute the 25$^{th}$ percentile of each row of `mymat`, defining the specific value by passing internally the argument `probs` (see `?quantile`). This offers a simple syntax to apply repeatedly complex functions to rows or columns.

Other functions of the apply family are used with different types of objects. For instance, `lapply()` and `sapply()` work with lists, repeating computations for all their components. Let's try these two functions for computing the length of the objects stored as components of the list `mylist`, created in a previous section:

```
lapply(mylist, length)
```

```
## $vec2
## [1] 5
##
## $mat
## [1] 8
```

```
sapply(mylist, length)
```

```
## vec2  mat
##    5    8
```

While the `lapply()` function returns another list, `sapply()` also tries to *simplify* the results, in this case to a vector (therefore, its name). Names of the components, if any, are reused in the results. The use of lists in computing with R is very powerful, as they can store objects with different lengths and also different modes. Interestingly, the functions above can also be applied to data frames, which are simply treated as lists with variables as components.

One important function in the apply family is `tapply()`, which repeats the computation provided by another function over groups defined by one or more factors. Let's use it with the dataset `esoph`, from the package `dataset`, computing the total number of controls by age groups (see `help(esoph)` for details):

```
tapply(esoph$ncontrols, INDEX=esoph$agegp, FUN=sum)
```

```
## 25-34 35-44 45-54 55-64 65-74   75+
##   115   190   167   166   106    31
```

The arguments of `tapply()` define a vector used for computation, a cross-classifying factor, and the function to be applied, respectively. More cross-classifying factors can be included in a list in `INDEX`.