Chapter 7 Advanced R

> Talk is cheap. Show me the code.
> –Linus Torvalds

7.1 Vectorization

While we already intesively talked about loops we have expercienced and discussed some or their drawbacks. Especially in R loops are incomprehensible slow. Additionally they require to hancle index or iterator values, which may be confusing some times. Luckily R provides an alternative that eradicates these problems and allows to eventually deprecated the usage of loops in R in favor of vectorized functions. Ff you understand the nuts and bolts of vectorization in R, the following functions allo you to write shorter, simpler, safer, and faster code.

In the previous sections we already introduced some vectorized functions without explicitly metioning their rationale and that they are vectorized. The following functions all belong to the so called apply-Family. Vectorization in R requires a thorough understanding of the available data strucutres, as the following functions iterate (automatically) over different slices of data strucutes and perform (loop-wise) repetitions on the data slices from vectors, matrices, arrays, lists and dataframes. More specifically, the family consists of apply(), lapply() , sapply(), vapply(), mapply(), rapply(), and tapply().

```
base::apply     Apply Functions Over Array Margins
base::eapply        Apply a Function Over Values in an Environment
base::lapply        Apply a Function over a List or Vector
base::mapply        Apply a Function to Multiple List or Vector Arguments
base::rapply        Recursively Apply a Function to a List
base::tapply        Apply a Function Over a Ragged Array
```

7.1.1 apply

We already introduced `apply` and used it to apply a function to the rows or columns of a matrix, in the same fashion as functions like `rowMeans` or `colMeans` calculate a specific values for either a row or a column of a matrix. Generally speaking apply operates on (two dimensional) arrays, a.k.a. matrices. To get started we create a sample data set consisting of a matrix with 20 cells partitioned into five rows and four columns.

```
mat <- matrix(c(1:10, 21:30), nrow = 5, ncol = 4)
mat
```

```
#R>      [,1] [,2] [,3] [,4]
#R> [1,]    1    6   21   26
#R> [2,]    2    7   22   27
#R> [3,]    3    8   23   28
#R> [4,]    4    9   24   29
#R> [5,]    5   10   25   30
```

To mimic the functionality of `rowSums`, we now can use the apply function to find the sum over all elements of each row as follows.

```
apply(X=mat, MARGIN=1, FUN=sum)
```

```
#R> [1] 54 58 62 66 70
```

Notice that the funciton call to `apply` takes three arguments, where `X` is the data, `MARGIN` corresponds eiter to the rows as they are the first dimension of the data or to the columns, which correspond to the second dimension. `FUN` is the funciton that should be applied on the specified margin of the data. Note that the function in the snippet below is passed without parantheses (`sum` instead of `sum()`).

Remember that in R everything is a vector. Therefore, a matrix can be seen as a collection of line vectors when you cross the matrix from top to bottom (along `MARGIN=1`), or as a list of column vectors, spanning the matrix left to right (along `MARGIN=2`). The code in the above R chunk therefore translates directo to the instruction to: “apply the function sum to the matrix `mat` along the rows”. Ursuprisingly this leads to a vector containing the sums of the values of each row. Mathematically speaking we would expect a column vector here, while R outputs a line vector. As R does not differentiate here while outputting these on the console this makes no difference for this case. The following picture illustrates the process.

7.1.2 lapply

While matrices are an important and often used data structure they are not the only one. Quite often data comes as `list` and it may be a reasonable purpose to apply a function to every (sub-) element of given list. As lists have no dimensions (see `dim`), the application of `apply` fails. Thefeore if you want to apply a specific function to every element of a list you can use a list compatible version of apply, the `lapply`-function. The syntax is quite comparable to our usual `apply`, which can be seen when executing `?lapply`.

Due to the flexibility and ubiquitousness of lists, `lapply` can be widely used and e.g. also works on dataframes in addition to lists. Additionally it is compatible with vectors, where the second most important part about `lapply` comes into place. Regardless if the data input `X` is a list, a dataframe or a vector, the returned data is always a `list`.

Our toy example, depicted in figure 2 can be coded as:

```
lst <- list(1:5, 6:10, 21:25, 26:30)
lst
```

```
#R> [[1]]
#R> [1] 1 2 3 4 5
#R>
#R> [[2]]
#R> [1]  6  7  8  9 10
#R>
#R> [[3]]
#R> [1] 21 22 23 24 25
#R>
#R> [[4]]
#R> [1] 26 27 28 29 30
```

```
lapply(X=lst, FUN=sum)
```

```
#R> [[1]]
#R> [1] 15
#R>
#R> [[2]]
#R> [1] 40
#R>
#R> [[3]]
#R> [1] 115
#R>
#R> [[4]]
#R> [1] 140
```

7.1.3 sapply

```
sapply(X=lst, FUN=sum)
```

```
#R> [1]  15  40 115 140
```

7.1.4 vapply

https://ademos.people.uic.edu/Chapter4.html#303_example_3:_vapply

vapply is similar to sapply, but it requires you to specify what type of data you are expecting the arguments for vapply are vapply(X, FUN, FUN.VALUE). FUN.VALUE is where you specify the type of data you are expecting. I am expecting each item in the list to return a single numeric value, so FUN.VALUE = numeric(1).

```
#vapply(vec, sum, numeric(1))
#vapply(my.lst, sum, numeric(1))
```

If your function were to return more than one numeric value, FUN.VALUE = numeric(1) will cause the function to return an error. This could be useful if you are expecting only one result per subject.

7.1.5 mapply

The mapply() function stands for 'multivariate' apply. Its purpose is to be able to vectorize arguments to a function that is not usually accepting vectors as arguments.

In short, mapply() applies a Function to Multiple List or multiple Vector Arguments.

Let's look at an mapply() example where you create a 4 x 4 matrix with a call to the rep() function repeatedly:

Another example:

```
x <- 1:5
b <- 6:10
mapply(sum, x, b)


#R> [1]  7  9 11 13 15
```

7.1.6 tapply

tapply splits the array based on specified data, usually factor levels and then applies the function to it.

For example, in the mtcars dataset:

```
library(datasets)
tapply(mtcars$wt, mtcars$cyl, mean)


#R>        4        6        8
#R> 2.285727 3.117143 3.999214
```

The tapply function first groups the cars together based on the number of cylinders they have, and then calculates the mean weight for each group.

7.1.7 rapply

http://www.datasciencemadesimple.com/apply-function-r/
http://www.dataperspective.info/2016/03/apply-lapply-rapply-sapply-functions-r.html
https://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/

7.1.8 eapply

7.2 Other vectorized functions

rep map by

7.3 Environments + Scoping

7.4 Lazy Evaluation

7.5 Debugging Software

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. – Brian W. Kernighan

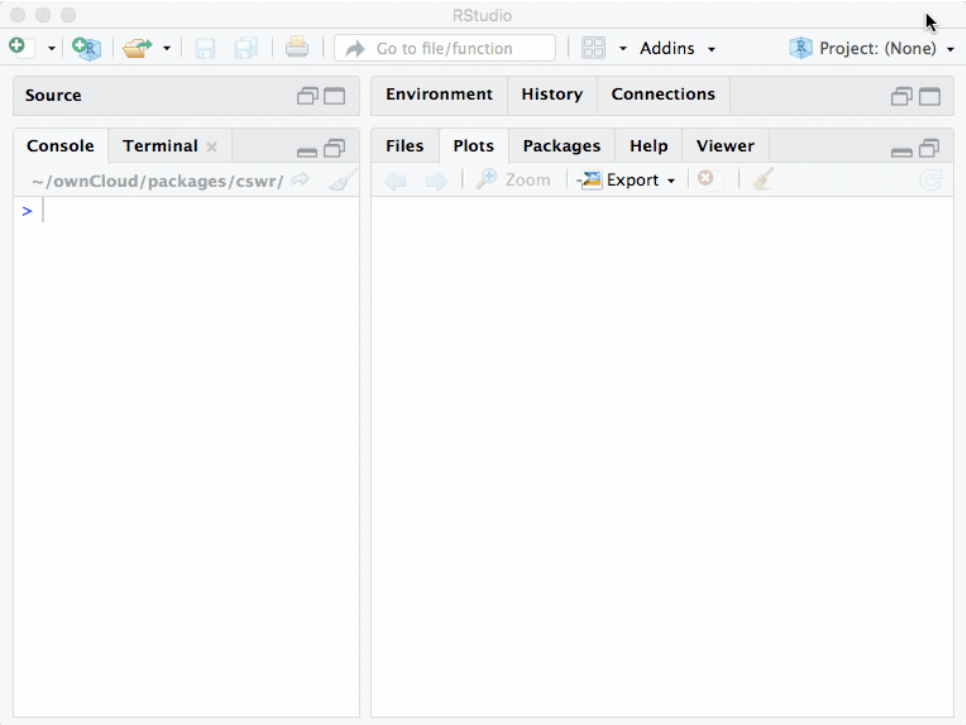
Debugging != Testing ## Functions are Objects

7.6 Object Orientation in R: S3 and S4 Classes

7.7 Recursion

> In order to understand recursion, one must first understand recursion. – Anonymous

Tower of Hanoi A more advanced example that can be solved using a recursive algorithm is the childrens game “Tower of Hanoi”.



The R Code for this puzzle game can be found in the repository at github.com/yihui.

7.8 Speed and Vectorized Code

apply tabelle ligges s.104