The Beauty of R Programming

Exercises

RLadies, Berlin

18/04/2018

Verena Pflieger
INWT Statistics GmbH
Hauptstr. 8
Meisenbach Höfe, Aufgang 3a
10827 Berlin
+49 30 120823133
verena.pflieger@inwt-statistics.de
www.inwt-statistics.de

1 Exercises Functions

In the following section, we present some exercises for the topics you just learned. You find the solutions at the end of the section .

Functions

1. For each of the following functions, predict the return value and think about meaningful function names instead of `f1, f2, ...`. If you do not know what the functions `sum, length` and `order` do, have a look at the help files. Do the computation to check if your predictions are correct.

a)

```
f1 <- function(r) pi * r^2
f1(r = 1)
```

b)

```
f2 <- function(r) 2 * pi * r
f2(r = 1)
```

c)

```
f3 <- function(x) sum(x) / length(x)
f3(x = c(1, 1, 3, 3))
```

d)

```
f4 <- function(x) sum((x - f3(x))^2)
f4(x = c(1, 1, 3, 3))
```

e)

```
f5 <- function(x) x[order(x)]
f5(x = c("c", "b", "a"))
```

f)

```
f6 <- function(x, ...) x[order(x, ...)]
f6(x = c("a", "b", "c"),
   decreasing = TRUE)
```

2. Write a function `meanVarSdSe` that takes a numeric vector `x` as argument. The function should return a named vector that contains the mean, the variance, the standard deviation `sd` and the standard error `se` of `x`. The standard error is defined as

$$se(x) = \frac{sd(x)}{\sqrt{\#x}}, \tag{1}$$

where `#x` denotes the cardinality, i.e., the number of elements contained in `x`.

a) The code should have the following structure

```
meanVarSdSe <- function(x) {
  ### your code
}
```

and return a named vector according to

```
x <- 1:100
meanVarSdSe(x)
```

```
##       mean        var         sd         se
##  50.500000 841.666667  29.011492   2.901149
```

You can use the functions `mean`, `var`, `sd` and `length`. Check the help files for these functions for further arguments that can be used optionally.

b) Look at the following code sequence. What result do you expect?

```
x <- c(NA, 1:100)
meanVarSdSe(x)
```

Now run the code. Explain the result. Extend the function definition of `meanVarSdSe` with the argument `...`, as is illustrated as follows:

```
meanVarSdSe <- function(x, ...) {
  ## your code
}
```

so that the `na.rm = TRUE` argument can be passed optionally to the functions `mean`, `var` and `sd`.

What is the correct value for $\#x$ in the case of missing values? Use `sum(!is.na(x))` as denominator in equation (1). Read the help page for the function `is.na()`. The optimized function should return:

```
meanVarSdSe( c(x, NA), na.rm = TRUE)
```

```
##       mean        var         sd         se
##  50.500000 841.666667  29.011492   2.901149
```

3. See what happens if after arguments marked with `...`, you want to set another argument, but forget to name it correctly:

```
cat(letters[1:10], fill = TRUE, labels = paste0("{", 1:10, "}:"))
```

What do you think happened?

4. Remember what happened when we called on the following function:

```
f <- function(a, b) {
  print(a)
  print(b)
}
```

Now rebuild the function such that it does not throw an error when only one argument is passed.

5. Sometimes it might be useful to specify a function's arguments outside the function, e.g., when they are used in several calls and you don't want to copy and paste them over and over again. Look at the help page of the function `do.call()` and rewrite the following function call specifying its arguments beforehand.

```
mydata <- 1:20
sd(mydata, na.rm = FALSE)
```

Environments

12. Create a new environment and check which environment was set as its parent.

13. Now do the same and set the parent explicitly. How can you check whether it worked or not?

14. Imagine we have a dataset `x` with two observations and one variable `V1`. So let's use the function `nrow` to examine it further. What does it do? And how does the function code look like? Now let's define our own functions `dim` and `newNrow` – they probably act somewhat differently.

```
x <- data.frame(V1 = 1:2)
str(x)
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ V1: int  1 2
```

```
nrow # looking at function code
```

```
## function (x)
## dim(x)[1L]
## <bytecode: 0x0000000016c0dde8>
## <environment: namespace:base>
```

```
# let's define our own functions
dim <- function(x) 1
newNrow <- function(x) dim(x)[1L]
```

What would you expect calling `nrow(x)` and `newNrow(x)`? What does actually happen? Why do you think is that the case?

15. Look at the following function. What is the return value of `f(10)`?

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x ^2
    }
    f(x) + 1
  }
```

```
  f(x) * 2
}
f(10)
```

Functions and Environments

16. What is the result of a call to `someFunction`? What happened?

```
x <- 1
someFunction <- function() x
```

17. What is the result of a call to `someFunction`?

```
rm(list = ls())
x <- 1
someFunction <- function() x * y
```

18. The same rules apply if you define functions inside other functions. So what do you expect as result when the following function is called?

```
x <- 1

someFunction <- function() {
  y <- 2
  otherFunction <- function() {
    z <- 3
    c(x, y, z)
  }
  otherFunction()
}
```

2 Solutions Functions

Functions

1. In the following, we present some meaningful function names. Note that we like the naming convention `lowerCamelCase`.You can check the return value yourself by executing the code.

a) areaCircle

b) circumferenceCircle

c) averageVector

d) sumOfSquares

e) sortVectorAscend

f) sortVector

2. Function `meanVarSdSe` for calculating mean, variance, standard deviation and standard error and returning them in form of a a named vector.

a) Function code:

```
meanVarSdSe <- function(x) {
  n <- length(x)
  c(mean = mean(x),
    var = var(x),
    sd = sd(x),
    se = sd(x) / sqrt(n))
}
```

b) Without further specifying an option for handling missing values, the functions used "return missing values"" in the presence of `NA`s. Hence, we have to extend the function such that the argument `na.rm = TRUE` can be passed through.

```
meanVarSdSe <- function(x, ...) {
  n <- length(na.omit(x))
  c(mean = mean(x, ...),
    var = var(x, ...),
    sd = sd(x, ...),
    se = sd(x, ...) / sqrt(sum(!is.na(x))))
}
```

```
x <- 1:100
meanVarSdSe(c(x, NA), na.rm = TRUE)
```

```
##       mean        var         sd         se
##  50.500000 841.666667  29.011492   2.901149
```

3. After `...`, partial matching does not work and thus the wrongly spelled argument `fil` was not identified as valid argument. Hence, its value `TRUE` was taken as additional element to be concatenated and printed. The succeeding argument `labels` was ignored. Setting things right, you can check what kind of result we wanted to achieve:

```
cat(letters[1:10], fill = TRUE, labels = paste0("{", 1:10, "}:"))
```

```
## {1}: a b c d e f g h i j
```

4. The function `f`, as it is defined above, requires two parameters to be printed: `a` and `b`. If only one argument is given, it throws an error when evaluated (lazy evaluation). This can be avoided by providing default values in the function definition:

```
f <- function(a = NULL, b = NULL) {
  print(a)
  print(b)
}
```

5. The function `do.call()` constructs a function call from a function name and a list of arguments. In this case, its first argument is `sd`, the function's name, and its second argument is `args`, a list of the arguments to be passed to the function `sd()`.

```
args <- list(mydata, na.rm=FALSE)
do.call(sd, args)
```

Environments

12. If we simply construct a new environment, it is built per default in the global environment:

```
f <- new.env()
f$x <- 1
parent.env(f)
```

```
## <environment: R_GlobalEnv>
```

13. We can alternatively set the parent explicitly. In order to check whether it worked, we can use the function `parent.env()`:

```
e <- new.env(parent = f)
e$y <- 2
parent.env(e)
```

```
## <environment: 0x000000001f2ff828>
```

```
f
```

```
## <environment: 0x000000001f2ff828>
```

14. This exercise is about functions and environments. First we examine the function `nrow()`:

```
# nrow environment
x <- data.frame(V1 = 1:2)
str(x)
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ V1: int  1 2
```

```
nrow # looking at function code
```

```
## function (x)
## dim(x)[1L]
## <bytecode: 0x0000000016c0dde8>
## <environment: namespace:base>
```

We see that the function is located in the *base* package and, internally, it uses the function `dim()`. Our own functions are both defined in the global environment.

```
dim <- function(x) 1
newNrow <- function(x) dim(x)[1L]
```

If we now execute both `nrow` functions, we can check whether or not they use different `dim` functions. And in fact, they return different results. Obviously, the old `nrow()` function uses the `dim()` function from the *base* package and the `newNrow()` function uses the previously defined `dim` function. Using `search()` returns the order in which the packages and objects are searched. Since our `newNrow()` function lives in the global environment, it uses the `dim()` function living in the global environment as well. On the contrary, the `nrow()` function living in the *base* package uses the `dim()` function living in the same enviroment.

```
nrow(x)
```

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

```
newNrow(x)
```

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

```
ls(all.names = TRUE)
```

```
## [1] "dim"         "e"           "f"           "meanVarSdSe" "newNrow"
## [6] "x"
```

15. The functions are evaluated from the inside to the outside. Hence, the result is 202.

Functions and environments

16. Since the function `someFunction()` does not take any parameter, it always returns `x` as defined in the workspace.

17. Since `y` is not defined, the call to `someFunction()` returns an error.

```
rm(list = ls())
x <- 1
someFunction <- function() x * y
someFunction()
```

```
## Error in someFunction(): Objekt 'y' nicht gefunden
```

18. Since `R` uses the concept of lazy evaluation, the parameters `x`, `y`, and `z` are not needed until the function `someFunction()` is called. Since `x` is defined in the working directory and `y` and `z` are defined inside of the function, a call to `someFunction()` returns `1,2,3`.

3 Exercises Control Structures

Control structures

1. For each of the following code sequences, predict the value of `answer` after running the loop. Then do the computation.

a)

```
answer <- 0
for (j in 3:5) {answer <- j + answer}
```

b)

```
answer <- 10
for (j in 3:5) {answer <- j + answer}
```

c)

```
answer <- 0
for (j in 3:5) {answer <- j * answer}
```

d)

```
answer <- 10
for (j in 3:5) {answer <- j * answer}
```

e)

```
answer <- NULL
for (j in 3:5) {answer <- c(answer, j)}
```

f)

```
answer <- c(2, 7, 1, 5, 12, 3, 4)
for (j in 2:length(answer)) {
  answer[j] <- max(answer[j], answer[j - 1])
}
```

2. The $\sqrt{N}$ law: The precision of the sample average improves with the square root of the sample size N. Simulate 10^6 Poisson-distributed random numbers with expectation value $\lambda = 100$ via

```
set.seed(1)  # see ?set.seed for details
x <- rpois(n = 1e6, lambda = 100)
```

a) Write a loop (`repeat` or `while`) which calculates the mean, variance, standard deviation, and standard error of the first N elements of `x` until `se` ≤ 0.05. Start with $N = 2$ and multiply N after each iteration by a factor of 2. Store N and the calculated values for each iteration as rows in a matrix named `result`. Use the function `meanVarSdSe()` developed in Section 1. Make sure that the column names of your result matrix match the following output. Your matrix should look like

```
tail(result)  #  see ?tail for details
```

```
##           N      mean       var       sd         se
## [11,]  2048  99.90137 104.63414 10.22908 0.22603293
## [12,]  4096  99.97803 101.65813 10.08257 0.15754008
## [13,]  8192 100.02466  99.13747  9.95678 0.11000792
## [14,] 16384 100.00885 100.58669 10.02929 0.07835384
## [15,] 32768 100.04257 101.13827 10.05675 0.05555623
## [16,] 65536  99.97209 100.32065 10.01602 0.03912508
```

Are these values plausible?

b) Explain why a `for` loop is not optimal for this specific task.

c) Lower the break condition to `se` ≤ 0.01. Run the changed code. Make sure that you reinitialize `N` and `result` before running the loop. What happens? Compare `N` and `length(x)`. Which expression causes the error?

3. Design a statement that checks several conditions of `x`: whether it is not numeric, whether it is greater than zero, smaller than zero or whether it equals zero. Depending

on what is true, return a character string stating which condition applies. You can use the `print` function here.

4. Imagine you have the following two vectors: `a` – containing names of students, and `b` – containing their grades.

```
a <- c("Peter", "Paul", "Mary", "Max", "Moritz")
b <- c("Grade", 3, 2, 1, 1, 5)
```

Use a `for` loop in order to print the names in combination with their grades. Peter has grade 3, Paul has grade 2 etc. Prepare your function such that the number of students could change without you having to modify the function.

Note that this exercise could be solved using a `mapply()` function, as well. So if you already know how, you are welcome to do just that.

5. Write a function called `cormatrix` that takes a dataframe as argument, selects the numeric variables from the dataframe, and prints a correlation matrix for these numeric variables. The variable names should be preserved in the correlation matrix. Note that it is about chosing the numeric columns here. The correlations themselves can be computed via the `cor()` function.

6. Let's look at a drunken sailor staggering forward:

```
pos <- 0
step <- -1
while (pos >= -3 && pos <= 5) {
  cat("Step =", step <- step + 1, "\n")
  cat("Position =", pos, "\n")
  coin <- rbinom(n = 1, size = 1, prob = 0.5)
  if (coin == 1) { ## random walk
    pos <- pos + 1
  } else {
    pos <- pos - 1
  }
}
```

Now imagine the drunken sailor needs a break every five steps. Use the `next` structure to modify the loop accordingly. Note that the modulo operation %% could be useful.

*apply() functions

7. Let's look at the `apply()` function, used to apply a function over the rows or columns of a matrix or array. First, generate the matrix `matA` via

```
set.seed(1)
matA <- matrix(sample(24), ncol = 6)
matA
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5   11   23    6   17
## [2,]    9   18    1   21   24   15
```

```
## [3,]   13   19    3    8   22    2
## [4,]   20   12   14   16    4   10
```

a) Sort the columns of the matrix in ascending order. Your result should look like

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5    1    8    4    2
## [2,]    9   12    3   16    6   10
## [3,]   13   18   11   21   22   15
## [4,]   20   19   14   23   24   17
```

b) Sort the columns of the matrix in descending order. Your result should look like

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   20   19   14   23   24   17
## [2,]   13   18   11   21   22   15
## [3,]    9   12    3   16    6   10
## [4,]    7    5    1    8    4    2
```

c) Sort the rows of the matrix in descending order. Your result should look like

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   23   17   11    7    6    5
## [2,]   24   21   18   15    9    1
## [3,]   22   19   13    8    3    2
## [4,]   20   16   14   12   10    4
```

9. Next, we generate a 2 by 2 by 10 array of the following form:

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
```

Using `rowMeans()`, it is possible to build means over the third dimension:

```
rowMeans(a, dims = 2)
```

```
##             [,1]       [,2]
## [1,] -0.05490216  0.4406218
## [2,] -0.03197821 -0.1952573
```

Now go ahead and use the `apply()` function in order to obtain the same result.

10. Create a list of matrices via

```
set.seed(1)
n <- 5     # number of list elements that will be created
matNames <- paste("m", 1:n, sep = "_")
matList <- list()

for (i in matNames) {
  cols  <- sample(2:5, 1)
  rows  <- sample(2:5, 1)
  matList[[i]] <- matrix(runif(rows * cols),
                         ncol = cols, nrow = rows)
```

```
}

str(matList)
```

```
## List of 5
##  $ m_1: num [1:3, 1:3] 0.573 0.908 0.202 0.898 0.945 ...
##  $ m_2: num [1:4, 1:2] 0.384 0.77 0.498 0.718 0.992 ...
##  $ m_3: num [1:4, 1:2] 0.1256 0.2672 0.3861 0.0134 0.3824 ...
##  $ m_4: num [1:3, 1:4] 0.186 0.827 0.668 0.794 0.108 ...
##  $ m_5: num [1:2, 1:5] 0.477 0.732 0.693 0.478 0.861 ...
```

a) Calculate the dimensions with `lapply()` and `sapply()` for each matrix contained in `matList`. The `sapply()` output should be equal to

```
##      m_1 m_2 m_3 m_4 m_5
## [1,]   3   4   4   3   2
## [2,]   3   2   2   4   5
```

```
## $m_1
## [1] 3 3
##
## $m_2
## [1] 4 2
##
## $m_3
## [1] 4 2
##
## $m_4
## [1] 3 4
##
## $m_5
## [1] 2 5
```

b) Check for each list element if it is a diagonal matrix with `sapply()/lapply()` and the following function:

```
is.diagonal <- function(x) nrow(x) == ncol(x)
```

12. Remember the `for` loop where we built the dynamic vector of results for our capital stock of the next five years.

```
capital <- 100
n <- 5
for (i in 1:n) {
  capital <- c(capital, capital[1] + i * 50)
}
```

Now rebuild the function using a function from the `*apply` family. Take both versions, rerun them making predictions for the next 10000 years, and look at the system time used to execute the commands. What do you notice?

14. Look at the following list of vectors:

```
lHard <- list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Why do you think we called it `lHard`? Well, of course there is an easier way to produce the same output. So go ahead and rewrite the expression using `mapply`.

15. Load the `airquality` dataset via

```
data(airquality)    # the 'data' function loads the dataset
summary(airquality) # ?summary for details
```

```
##      Ozone           Solar.R          Wind             Temp
##  Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00
##  1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00
##  Median : 31.50   Median :205.0   Median : 9.700   Median :79.00
##  Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88
##  3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00
##  Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00
##  NA's   :37       NA's   :7
##      Month            Day
##  Min.   :5.000   Min.   : 1.0
##  1st Qu.:6.000   1st Qu.: 8.0
##  Median :7.000   Median :16.0
##  Mean   :6.993   Mean   :15.8
##  3rd Qu.:8.000   3rd Qu.:23.0
##  Max.   :9.000   Max.   :31.0
##
```

a) Calculate the mean temperature for each month separately with

```
tapply(airquality$Temp,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE)
```

```
##        5        6        7        8        9
## 65.54839 79.10000 83.90323 83.96774 76.90000
```

b) Now calculate the mean `Ozone` concentration for each month. Your result should be

```
##        5        6        7        8        9
## 23.61538 29.44444 59.11538 59.96154 31.44828
```

c) Which parameter do you have to change to obtain the following output?

```
## $`5`
## [1] 23.61538
##
## $`6`
## [1] 29.44444
##
## $`7`
```

```
## [1] 59.11538
##
## $`8`
## [1] 59.96154
##
## $`9`
## [1] 31.44828
```

4 Solutions Control Structures

Control structures

1. This question is best solved if we mentally go through each iteration of the loop and keep in mind the result of the iteration. This is hard for two reasons: first, remembering with which value of `answer` we started. And second, understanding the rather unclear notation of the loop. So a nice practice would be to explain in advance what the loop is programmed for.

2. The $\sqrt{N}$ law is crucial for doing statistics. A lot of concepts work the way they do because they assume the $\sqrt{N}$ law.

```
set.seed(1)  # see ?set.seed for details
x <- rpois(n = 1e6, lambda = 100)
```

a) Set the break condition and the factor by which `N` is multiplied after each iteration of the loop, first. Note that we should be sure that the break condition will be reached sometime, otherwise the main memory gets crowded.

```
tol <- 0.05
factor <- 2
```

```
meanVarSdSe <- function(x) {
  c(mean = mean(x),
    var = var(x),
    sd = sd(x),
    se = sd(x) / sqrt(length(x)))
}

result <- NULL
n <- 2

repeat {
  result <- rbind(result, c(N = n, meanVarSdSe(x[1:n])))
  if (result[nrow(result), "se"] <= tol) break
  n <- n * factor
}

tail(result)
```

```
##            N      mean       var       sd         se
## [11,]   2048  99.90137 104.63414 10.22908 0.22603293
## [12,]   4096  99.97803 101.65813 10.08257 0.15754008
## [13,]   8192 100.02466  99.13747  9.95678 0.11000792
## [14,] 16384 100.00885 100.58669 10.02929 0.07835384
## [15,] 32768 100.04257 101.13827 10.05675 0.05555623
## [16,] 65536  99.97209 100.32065 10.01602 0.03912508
```

The values returned are plausible indeed. The more values are involved in the computation, the more precise the estimation of the mean gets. Also, since the number of observations is used in the denominator of the standard error, the standard error becomes smaller as the sample size increases. Hence, the law of large numbers works.

b) When using a `for` loop, we have to prespecify the amount of iterations the loop is meant to run. In the example above, we are not sure how many iterations will be necessary to calculate the mean up to a certain precision. Hence, a `for` loop would not be adequate.

c) If the break condition is lowered, the number of iterations needed for convergence increases. Then the number of observations needed for the estimation exceeds the length of our prespecified vector. Hence, calculation of the mean, variance, standard deviation, and standard error return a missing value `NA`. When the missing value is compared to the break condition, the loop stops and an error is returned:

```
Error in if (result[nrow(result), "se"] <= tol) break :
  missing value where TRUE/FALSE needed
```

3. Note that in the following statement the order of queries is not chosen at random. It makes absolutely sense to check whether or not `x` is a number before it is checked whether or not `x` is positive or negative etc.

```
if (!is.numeric(x)) {
  print("x is not a number")
} else if (x > 0) {
  print("x is a positive number")
} else if (x < 0) {
  print("x is a negative number")
} else if (x == 0) {
  print("x equals zero")
}
```

4. The hardest challenge here is that vector `a` starts with the names right away while the grades in vector `b` start at the second position. The rest of the `for` loop should be straight forward.

```
a <- c("Peter", "Paul", "Mary", "Max", "Moritz")
b <- c("Grade", 3, 2, 1, 1, 5)

for (i in seq_along(a)) {
  print(c(a[i], b[i+1]))
}
```

```
## [1] "Peter" "3"
## [1] "Paul" "2"
## [1] "Mary" "1"
## [1] "Max" "1"
## [1] "Moritz" "5"
```

If you used a `mapply()` structure, this is your solution:

```
mapply(paste, a, b[-1])
```

```
##      Peter       Paul       Mary        Max     Moritz
##  "Peter 3"   "Paul 2"   "Mary 1"    "Max 1" "Moritz 5"
```

As in most of the cases, the `apply()` structure takes less lines than a `for` loop.

5. The following function uses a `for` loop which loops over the columns of the given dataframe. It checks for each column if it is numeric. If so, the column is added to an extra dataframe and its name is written to a character vector. When all columns have been checked, the names are written to the extra dataframe and the correlations are calculated using the `cor()` function.

```
cormatrix <- function(x) {
  df <- NULL
  cnames <- NULL
  for (i in names(x)) {
    if (is.numeric(x[, i])) {
      df <- cbind(df, x[, i])
      cnames <- c(cnames, names(x[i]))
    }
  }
  colnames(df) <- cnames
  print(cor(df))
}
```

6. In order to give the drunken sailor a break every five steps, we can use the `next` structure. The challenge is to identify the time when the sailor has taken five steps. This can be achieved using the modulo operation. If the number of steps modulo five equals 0, we know that the sailor has taken five additional steps. We can then skip one step using `next`.

```
pos <- 0
step <- -1
while (pos >= -3 && pos <= 5) {
  cat("Step =", step <- step + 1, "\n")
  cat("Position =", pos, "\n")
  if (step %% 5 == 0 & step != 0) next
  coin <- rbinom(n = 1, size = 1, prob = 0.5)
  if (coin == 1) { ## random walk
    pos <- pos + 1
  } else {
    pos <- pos - 1
  }
```

```
}
```

*apply() functions

7. Let's turn to the solution for the `apply()` exercise now. To start with, we do just simple sorting on the rows or columns of a matrix.

a) You can sort the columns of the matrix using the following command. The `2` stands for the columns, the command `sort` uses ascending order by default.

```
set.seed(1)
matA <- matrix(sample(24), ncol = 6)
matA
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5   11   23    6   17
## [2,]    9   18    1   21   24   15
## [3,]   13   19    3    8   22    2
## [4,]   20   12   14   16    4   10
```

```
apply(matA, 2, sort)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5    1    8    4    2
## [2,]    9   12    3   16    6   10
## [3,]   13   18   11   21   22   15
## [4,]   20   19   14   23   24   17
```

b) In order to reverse the sorting direction, we can simply add the parameter `decreasing = TRUE` to our previous statement.

```
apply(matA, 2, sort, decreasing = TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   20   19   14   23   24   17
## [2,]   13   18   11   21   22   15
## [3,]    9   12    3   16    6   10
## [4,]    7    5    1    8    4    2
```

c) Now, sorting the rows seems to be a little bit tricky. If we simply exchange the argument `2` for a `1`, the right operation is done but additionally, the matrix is transposed. Hence, in order to get back the right format, we have to transpose the result.

```
t(apply(matA, 1, sort, decreasing = TRUE))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   23   17   11    7    6    5
## [2,]   24   21   18   15    9    1
## [3,]   22   19   13    8    3    2
## [4,]   20   16   14   12   10    4
```

9. Working with `apply()` in a 2-dimensional space is rather straightforward. It gets a little bit more difficult if the number of dimensions increases. In the following case, we have a total of three dimensions that computations could be done over. Reading the help page of `apply()` in detail helps to chose the right indices.

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(a, c(1, 2), mean)
```

```
##             [,1]       [,2]
## [1,] -0.05490216  0.4406218
## [2,] -0.03197821 -0.1952573
```

10. After we have followed the instructions, we gained a rather complex construct of matrices called `matList`. It is a list containing differently sized matrices. In order to work with them, the functions `lapply()` and `sapply()` can be used.

a) For calculating the dimensions of the matrices we use the `dim()` function. As expected, the output of the `sapply()` is somewhat less complex than `lapply()`'s output.

```
str(matList)
```

```
## List of 5
##  $ m_1: num [1:3, 1:3] 0.573 0.908 0.202 0.898 0.945 ...
##  $ m_2: num [1:4, 1:2] 0.384 0.77 0.498 0.718 0.992 ...
##  $ m_3: num [1:4, 1:2] 0.1256 0.2672 0.3861 0.0134 0.3824 ...
##  $ m_4: num [1:3, 1:4] 0.186 0.827 0.668 0.794 0.108 ...
##  $ m_5: num [1:2, 1:5] 0.477 0.732 0.693 0.478 0.861 ...
```

```
sapply(matList, dim)
```

```
##      m_1 m_2 m_3 m_4 m_5
## [1,]   3   4   4   3   2
## [2,]   3   2   2   4   5
```

```
lapply(matList, dim)
```

```
## $m_1
## [1] 3 3
##
## $m_2
## [1] 4 2
##
## $m_3
## [1] 4 2
##
## $m_4
## [1] 3 4
##
## $m_5
## [1] 2 5
```

b) Along the lines of the previous task, we can simply replace the `dim()` function with our self-made function `is.diagonal()`.

```
is.diagonal <- function(x) nrow(x) == ncol(x)

sapply(matList, is.diagonal)
```

```
##   m_1   m_2   m_3   m_4   m_5
##  TRUE FALSE FALSE FALSE FALSE
```

```
lapply(matList, is.diagonal)
```

```
## $m_1
## [1] TRUE
##
## $m_2
## [1] FALSE
##
## $m_3
## [1] FALSE
##
## $m_4
## [1] FALSE
##
## $m_5
## [1] FALSE
```

12. This is a very nice exercise to demonstrate differences in performance of a `for` loop compared to, let's say, a `sapply()`. Already when doing simple computation in 10000 steps, the elapsed time with the `for` loop exceeds the time needed for the `sapply()`.

```
# for loop
res <- 100
system.time(
  for (i in 1:10000) {
    res <- c(res, res[1] + i * 50)
  }
)
```

```
##    user  system elapsed
##    0.19    0.02    0.21
```

```
# sapply
start <- 100
i <- 1:10000
system.time(res <- c(start, sapply(i, function(x) start + x * 50)))
```

```
##    user  system elapsed
##    0.02    0.00    0.01
```

14. Using `mapply()` we are able to reduce the rather lengthy expression above to a very simple statement:

```
## the smart way
lSmart <- mapply(rep, 1:4, 4:1)
lSmart
```

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

15. We are looking at the `airquality` dataset which contains variables about measures of airquality.

a) In order to calculate the mean temperature for each month, we can use the `tapply()` function.

```
tapply(airquality$Temp,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE)
```

```
##        5        6        7        8        9
## 65.54839 79.10000 83.90323 83.96774 76.90000
```

b) Calculating the mean ozone concentration for each month, of course, is not very different. All you need to do is exchange the variable `Temp` with the variable `Ozone`.

```
tapply(airquality$Ozone,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE)
```

```
##        5        6        7        8        9
## 23.61538 29.44444 59.11538 59.96154 31.44828
```

c) Internally, the `tapply()` function works on a list. Since the result so far is a named vector, some simplification must have taken place. And in fact, by adding the argument `simplify = FALSE`, a list is returned.

```
tapply(airquality$Ozone,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE,
       simplify = FALSE)
```

```
## $`5`
## [1] 23.61538
##
## $`6`
## [1] 29.44444
##
## $`7`
## [1] 59.11538
##
## $`8`
## [1] 59.96154
##
## $`9`
## [1] 31.44828
```