CompStat/R - Paper 2

Group 2: Carlo Michaelis, Patrick Molligo, Lukas Ruff

21 June 2016

Part I: Functions

Functions I

Below we define a function `dropNa` which, given an atomic vector `x` as an argument, returns `x` after removing missing values.

```
dropNa <- function(x) {
  # Expects an atomic vector as an argument and returns it without missing
  # values
  #
  # Args:
  #   x: atomic vector
  #
  # Returns:
  #   The atomic vector x without missing values

  # To remove the NAs, we use simple logical subsetting
  y <- x[!is.na(x)]

  # Return y
  y
}
```

Let's test our implementation with the following line of code:

```
all.equal(dropNa(c(1, 2, 3, NA, 1, 2, 3)), c(1, 2, 3, 1, 2, 3))
```

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

As we can see from this positive test, our implementation was successful.

Functions II

Part I Below we define a function `meanVarSdSe` which, given a numeric vector `x` as an argument, returns the mean, the variance, the standard deviation and the standard error of `x`.

```
meanVarSdSe <- function(x) {
  # Expects a numeric vector as an argument and returns the mean,
  # the variance, the standard deviation and the standard error
  #
  # Args:
  #   x: numeric vector
  #
  # Returns:
```

```r
  #   a numerical vector containing mean, variance, standard deviation
  #   and standard error of x

  # We check if x is numeric vector
  # If not: stop and throw error
  if( !is.numeric(x) ) {
    stop("Argument needs to be numeric.")
  }

  # Create vector object
  y <- vector()

  # Calculate mean, variance, standard deviation and standard error
  # and save it in y
  y[1] <- mean(x)
  y[2] <- var(x)
  y[3] <- sd(x)
  y[4] <- y[3]/sqrt(length(x))

  # Set names to vector entries
  names(y) <- c("mean", "var", "sd", "se")

  # Return the numeric vector y
  y
}
```

To test the function, we define a numeric vector, which contains numbers from 1 to 100, and use it as an argument for our function `meanVarSdSe`:

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

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

Finally we can confirm that the result is of class `numeric`:

```r
class(meanVarSdSe(x))
```

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

Part II Now we will have a look at the case below. We would expect that the function will return a vector with `NA`s:

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

```
## mean  var   sd   se
##   NA   NA   NA   NA
```

The reason for the result is that the functions `mean()`, `var()` and `sd()` use `na.rm = FALSE` as default, which means that missing values are not removed. If the vector `x` contains a missing value, the `mean()` function (as well as `var()` and `sd()`) will just return `NA` to inform about missing values. In the case of calculating standard error we use the result from our `sd()` function and calculate an `NA` value with some other numeric values, which will ultimately result in `NA` again.

To solve the problem, we can add `na.rm = TRUE` to these three functions. To make this optional, we will improve the `meanVarSdSe` function from above as follows:

```
meanVarSdSe <- function(x, ...) {
  # Expects a numeric vector and flag to handle missing values as an argument
  # and returns the mean, the variance, the standard deviation
  # and the standard error
  #
  # Args:
  #   x: numeric vector, na.rm: boolean
  #
  # Returns:
  #   a numerical vector containing mean, variance, standard deviation
  #   and standard error of x

  # We check if x is numeric vector
  # If not: stop and throw error
  if( !is.numeric(x) ) {
    stop("Argument needs to be numeric.")
  }

  # Create vector object
  y <- vector()

  # Calculate mean, variance, standard deviation and standard error
  # and save it in y
  y[1] <- mean(x, ...)
  y[2] <- var(x, ...)
  y[3] <- sd(x, ...)
  y[4] <- y[3]/sqrt(length(x) - sum(is.na(x)))

  # Set names to vector entries
  names(y) <- c("mean", "var", "sd", "se")

  # Return the numeric vector y
  y
}
```

We define the function with an ellipse `...`. Our function can now receive multiple arguments after the first input `x`. These arguments are used in `mean()`, `var()` and `sd()`. If we want to remove missing values in all of these functions (to get a result in the case of missing values), we can pass `na.rm = TRUE` as another argument, such as here: `meanVarSdSe(x, na.rm = TRUE)`. We just have to be aware of `length(x)` in this case. If we want to have the same result as above we have to remove the sum of `NA` values from the length of `x`. Otherwise the function will calculate a different result than in Part I, because then lentgh differs.

Let's confirm the result:

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

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

Part III Now we will use the function `dropNa` from Functions I to deal with missing values in `meanVarSdSe`.

```r
meanVarSdSe <- function(x) {
  # Expects a numeric vector as an argument and returns the mean,
  # the variance, the standard deviation and the standard error
  # it also removes missing values if x contains some
  #
  # Args:
  #   x: numeric vector
  #
  # Returns:
  #   a numerical vector containing mean, variance, standard deviation
  #   and standard error of x

  # We check if x is numeric vector
  # If not: stop and throw error
  if( !is.numeric(x) ) {
    stop("Argument needs to be numeric.")
  }

  # We check if x contains missing values
  # If so: remove missing values using dropNA
  if( sum(is.na(x)) > 0 ) {
    x <- dropNa(x)
  }

  # Create vector object
  y <- vector()

  # Calculate mean, variance, standard deviation and standard error
  # and save it in y
  y[1] <- mean(x)
  y[2] <- var(x)
  y[3] <- sd(x)
  y[4] <- y[3]/sqrt(length(x))

  # Set names to vector entries
  names(y) <- c("mean", "var", "sd", "se")

  # Return the numeric vector y
  y
}
```

We used the function from Part I and added a condition which checks if we have missing values in `x`, using `is.na`. If the sum of `NA` values is greater than 0 (i.e.,if there is one or more missing value), we use the function `dropNA` from the first exercise to remove all missing values. The remaining code of the function can remain as above in Part I.

We can confirm the result:

```
meanVarSdSe(c(x, NA))
```

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

Functions III

In this section we define an infix function `%or%`. This function should behave like the logical operator `|`.

```
# Define infix function %or%
`%or%` <- function(a, b) {
  # Check if vector a and b is logical
  if( !(is.logical(a) & is.logical(b)) ) {
    stop("a and/or b have to be logical vectors.")
  }

  # Use ifelse to calculate result and return it directly
  # If the sum of the entry of vector a and the entry of vector b
  # is greater than or equal to 1, set result to TRUE, otherwise to FALSE
  ifelse(a + b >= 1,  TRUE, FALSE)
}
```

First we check if we have logical vectors. If `a` and/or `b` are not logical, we leave the function and throw an error. Otherwise we can calculate the `or` operation using the `ifelse` function and return the result directly after calculation. Inside of the `ifelse` function, the first argument checks the condition if the sum of the values `a` and `b` are greater than or equal to 1, where `TRUE` is equal to 1 and `FALSE` is equal to 0.

To confirm the function, we test an example:

```
c(TRUE, FALSE, TRUE, FALSE) %or% c(TRUE, TRUE, FALSE, FALSE)
```

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

Part II: Scoping and related topics

Scoping I

The main concept behind this exercise is the *Search Path*, which `R` uses to locate objects when called upon. In order for `R` to carry out a command or calculation, it seeks the necessary information according to a hierarchical path of *environments*. Each environment has a *parent*, to which `R` moves if the required information is not yet found. The `R` workspace is known as the *Global Environment* and also has a parent, which is the most recently loaded package. If there are no longer any loaded packages, then the search path *ends* at the final parent environment, the base package (`package:base`) which just has the empty environment as parent.

Below we can observe the importance of the search path with a simple example:

```
# Assign numeric values to the vectors x and y in the workspace
# which we call the global environment
x <- 5
y <- 7
```

```
f <- function() x * y
  # With no specified argument inputs, the function f follows the search path
  # and locates values for x and y in the global environment
g <- function(x = 2, y = x) x * y
  # A new environment is created within the function g, where arguments for x and y
  # are clearly defined
```

Although both functions `f` and `g` depend on values for `x` and `y`, they return different results when called:

```
# call 1
f()
```

```
## [1] 35
```

```
# call 2
g()
```

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

Beginning with function `f`, if we follow the search path we begin in the environment within the function itself. Since there is no information regarding the values of `x` and `y`, `R` moves to the parent environment, which is the global environment in this case. In the global environment, `x` takes the value of 5 and `y` takes the value of 7. Thus, the function returns $5 \cdot 7 = 35$.

For function `g` the search path also begins in the environment within the function itself. However, in this case there is a defined value for `x`, as well as an expression defining a value for `y` based on `x`. The search path ends and the function returns $2 \cdot 2 = 4$.

By manipulating the arguments of a function, it is also possible to alter the original search path. We see this when calling the following function:

```
# call 3
g(y = x)
```

```
## [1] 10
```

Looking back at the code for function `g`, we see the two arguments `x` and `y`. When calling `g(y = x)` however, we are omitting the first argument `x`, which then defaults to the value 2, defined in the function.
When we simply call `g()`, the `y = x` argument also defaults to a value dependent on `x`. But by inputing the argument `y = x` manually while calling, we send the search path to the global environment where `x` takes the value of 5. Thus the function returns $2 \cdot 5 = 10$.

Scoping II

In this exercise we once again see the importance of understanding the search path and how `R` carries out tasks according to the environment in which it is working. Especially important is the *naming* of objects and functions. As discussed in the previous section, the ultimate parent environment to use is `package:base`, which contains the commonly used and most fundamental functions in `R`. Since the global environment (workspace) is separate from `package:base`, it is possible to name new objects in the workspace using previously defined functions from the base. As long as there is no overlap *within* an enviornment, nothing will be overwritten, it will just be masked. In the following example we see again why the search path is so important when defining objects:

```r
# Define matrix t, where the number of columns is selected as 3
# and the matrix is filled row-wise
t <- matrix(1:6, ncol = 3, byrow = TRUE)

# Print matrix t
t
```

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

As expected, printing `t` returns a 2×3 matrix filled by row using the numbers one through six. Let's see what happens if we treat `t` like a function:

```r
# Print t(t), which should transpose matrix t
t(t)
```

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

The result is a 3×2 matrix filled by column using the numbers one through six. In other words, we have printed the transpose of the original matrix `t`, which we had defined in the global environment. Since `t` is a defined matrix and not a function, `R` will ignore the `t` in the global environment while searching for function `t`. `R` follows the search path from the global environment to the earlier parents and finds function `t` in `package:base`. In the base environment, the function `t()` returns the transpose of the given matrix.

Scoping III

In the previous exercises we observed how `R` searches through a chain of environments to locate objects and information. In this next exercise, we investigate what happens when different objects are defined identically within the *same* environment. Here we are defining objects in the global environment (workspace):

```r
# Define a function t in the global environment
t <- function(...) matrix(...)

# Define a matrix T using the above t function
# with the desired input arguments
T <- t(1:6, ncol = 3, byrow = TRUE)

# Print result of T
T
```

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

As expected, printing `T` returns a 2×3 matrix filled by row using the numbers one through six. Now let's enter `T` into the function `t`:

```r
# Call defined function t with argument T
t(T)
```

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

Since `t` is a function we have defined in our workspace (global environment), `t()` takes `T` as an argument input and returns a column vector containing the numbers one through six (note: `t` and `T` are different objects, because `R` is case sensitive). The transpose function `t()` from the base environment is now masked by our own function, saved in the global environment and now is just an alias of `matrix()` (as we defined it). Since the `matrix()` function has the default value `ncol = 1` and we were not giving any argument, it creates a matrix out of the data from `T` (values 1 to 6) and put it in just one column.

Let's now see what would happen if we had defined `T` instead as `t`:

```r
# Define a function t in the global environment
t <- function(...) matrix(...)

# We now define t as the following matrix using the t function from above
t <- t(1:6, ncol = 3, byrow = TRUE)
# Although we used the function t to define the new matrix t
# both are defined in the global environment

# Call defined function t with argument t
t(t)
```

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

Since two objects (independent of their type) cannot have the same name within our global environment, the new matrix `t` overwrites the original function. In our global environment, `t` is now a defined matrix and no longer a function, it was replaced. The search path now moves down (is looking earlier in path), until it finds `t`, defined as the transpose function (seen earlier), in base environment. It is therefore clear why we receive the same result as in the previous exercise when printing `t(t)` here.

This entire concept can be referred to as *name masking*. We can think of the transpose function `t` in the base environment as the "original" function. Each time a new object `t` is created in later environments, the original is "masked", but not overwritten. So if the search path is led back to the base environment, the original function can still be located.

Dynamic lookup