

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:
#   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`:

```

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`:

```

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 NAs:

```

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 `length` differs.

Let's confirm the result:

```
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`.

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

```

# We 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:

```
f()
```

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

```
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 $5 * 7 = 35$.

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 $2 * 2 = 4$.

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:

```
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, which then defaults to the value 2. When we simply call `g()`, the `y = x` argument also defaults to a value dependent on `x`. But by inputting `y = x` manually, we send the search path to the global environment where `x` takes the value of 5. Thus the function returns $2 * 5 = 10$.

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 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 name new objects in the workspace using previously defined functions from the base. As long as there is no overlap *within* an environment, nothing will be overwritten. In the following example we see again why the search path is so important when defining objects:

```
# We define a matrix t
# 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:

```
# Print t(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 follows the search path from the global environment to the parent, which is simply `package:base`. In the base environment, the function `t()` returns the transpose of the given matrix.

Scoping III

```
t <- function(...) matrix(...)
T <- t(1:6, ncol = 3, byrow = TRUE)
t(T)
```

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

```
t <- function(...) matrix(...)
t <- t(1:6, ncol = 3, byrow = TRUE)
t(t)
```

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

Dynamic lookup