Programmieren mit R: Seminararbeit 2

Daniyar Akhmetov (5127348)
Marcelo Rainho Avila (4679876)
Xuan Son Le (4669361)

Abgabedatum: 19.12.2017

Contents

1 Part I: *Functions* (15 points)

1.1 Functions I:

Define a function which given an atomic vector x as argument, returns x after removing missing values

```
dropNa <- function(x) {
  # takes an atomic vector as an argument and returns it without missing values
  x[!is.na(x)]
}
```

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

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

1.2 Functions II:

Part I

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

```
meanVarSdSe <- function(x){
  # takes an numeric vector as argument and returns a named numeric vector
  # containing its mean, variance, standard deviation and standard error
  c(mean = mean(x),
    var = var(x),
    sd = sd(x),
    se = sd(x) / sqrt(length(x))
  )
}

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

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

Part II

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

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

The code returns NA values for each statistic computed, which is the output of each function when using the default (FALSE) argument for `na.rm`.

```
meanVarSdSe <- function(x, ...){
  # computs mean, variance, standard deviation and standard error
  #
  # Args:
  #   x: a numeric vector
```

```
  #
  # Returns:
  #   mean, variance, standard deviation and standard error of input vector
  c(mean = mean(x, ...),
    var = var(x, ...),
    sd = sd(x, ...),
    se = sd(x, ...) / sqrt(length(which(!is.na(x))))
  )
}

# test
meanVarSdSe(x, na.rm = TRUE)
```

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

Part III

Write an alternative version of meanVarSdSe in which you make use of the function definition **dropNa** from the above exercise.

```
meanVarSdSe <- function(x, dropMissing = TRUE){
  # computs mean, variance, standard deviation and standard error while using
  # dropNa function per default
  #
  # Args:
  #   x: a numeric vector
  #
  # Returns:
  #   mean, variance, standard deviation and standard error of input vector
  if (dropMissing) {
    x <- dropNa(x)
  }
  c(mean = mean(x),
    var = var(x),
    sd = sd(x),
    se = sd(x) / sqrt(length(x))
  )
}

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

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

1.3 Functions III:

Write an infix function %or% that behaves like the logical operator |

```
"%or%" <- function(x,y) {
  # logical operator OR:
  # TRUE OR TRUE = TRUE
```

```
  # TRUE OR FALSE = TRUE
  # FALSE OR TRUE = TRUE
  # FALSE OR FALSE = FALSE

  ifelse(x == TRUE, TRUE,
         ifelse(y == TRUE, TRUE, FALSE))
}

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

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

2 Part II: *Scoping and related topics* (15 points)

Scoping I

Explain the results of the three function calls

```
x <- 5
y <- 7
f <- function() x * y
g <- function(x = 2, y = x) x * y
```

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

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

Function `f()` does not require any arguments and is defined as the product of `x` and `y`. It raises an error if `x` or `y` are not are not defined in the global (or any other parent) environment.

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

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

Function `g()` takes two arguments, which are `x` with a default value of 2 and `y`, which per default is assigned to the value of `x`. The function returns the product of `x` and `y`. In call 2, function `g()` is called without any arguments. In this case, x is set to 2 and y is equal to `x`. At first glance, it is unclear if R's interpreter will take the globally assigned value for x (here: 5) or if should take the local variable `x`, which is 2 per default. It turns out that the default arguments are evaluated in the local environment (that is, *inside* the actual function). Therefore, when calling `g()` with default arguments, y is set to equal the *local* x value, which is set to the default value of 2.

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

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

In *call 3*, the argument for `x` is not passed, so `x` will get the default value 2. Further, `y` is assigned to `x`. Differently to *call 2*, the value of `y` is evaluated when calling the function. Thus `y` is explicitly assigned to the *global* x (in this case 5).

Scoping II

Why and how does the following code work?

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

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

Here the variable `t` is defined to be a 2 by 3 matrix in the *Global Environment*. However, `t()` is also a function from base R for computing the transpose of a matrix or data.frame. When calling `t(t)`, the parentheses indicate that R should first look for a function`t()` and skip non-function objects, and then apply this function to a object called `t`. R searches for a function `t()` and finds it in the `package:base` environment and calculates the transpose of the previously defined matrix `t`. The call `t(t)` only works because these objects with the same name are not created within the same environment, even when taking into consideration that one is a function and the other is a matrix object.

Scoping III

Why do the results of t(T) and t(t) differ?

```
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
```

In this scenarion, `t()` is a function defined in the global environment that takes *any* arguments and passes them onto the base R `matrix()` function, which creates a matrix from a given set of values. Further, `T` is a 2 by 3 matrix. However, applying `t()` to `T` is the same as calling `matrix(T)`. This returns, somewhat surprisingly, a 6 by 1 matrix. This dimensionality distortion is due to the fact that a matrix object in R is, under the hood, a *long* one-dimensional atomic vector with a `dim` attribute indicating the number of rows and columns. The default value for the number of rows and columns in `matrix()` function is 1. Therefore, when calling `matrix()` on a matrix object (and not defining a different number of rowns or columns), the matrix `T` gets "unwrapped" into the underlying one-dimensional row vector (or a six-dimensional column vector).

The ordering [1,4,2,5,3,6] rather than [1,2,..,6] is due to `byrow = TRUE` argument when constructing the matrix. The same matrix could, for instance, be created by changing the dimensions of an atomic vector, as following,

```
aMatrix <- c(1,4,2,5,3,6)
dim(aMatrix) <- c(2,3)
```

which exemplifies the underlying nature of a matrix object in R.

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

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

In the above scenario, the first line basically maps the base function `matrix()` to `t()`. In the second line, `t()`, while still mapped as a function, creates a 2 by 3 matrix, which is also mapped to `t`, overwriting the previous definition, since we cannot have multiple objects with the same name within the same environment. In the third line. When calling `t(t)`, R searches for the **function** `t()` and finds the transpose function from base R and not the already overwritten function in the Global Environment.

Dynamic Lookup

Explain the results of the five function calls and why the rm function in line 1 is important.

```r
rm(list = ls(all.names = TRUE))
f <- function(x, y = x + 1) x + y
x <- 3
f(2) # call 1
## [1] 5
x <- 5
f(2) # call 2
## [1] 5

f <- function(y = x + 1) x + y
x <- 3
f(2) # call 3
## [1] 5
x <- 5
f(2) # call 4
## [1] 7
f()  # call 5
## [1] 11
```

Call 1 and 2 This function requires one argument, `x`, but also accepts a second argument, `y`. It returns the sum of `x` and `y`, with `y` equals `x + 1` per default.
This exemplifies the lazy evaluation property in R. By the time the function is created `x` doesn't exist and only when the function `f(2)` is called, 2 is passed as an argument for `x` and and `y` gets assigned to the expression `x + 1` from the newly created function environment.) The global value of `x` does not affect the function. Thus, call 1 and 2 returns the same result. One can affirm that `f()` is (weakly) self contained. That means, the values from global or parent environments don't affect the output of the function

Call 3 and 4 This function accepts only one argument, `y`, but a second variable, `x`, is required for it to work. Since this variable is not created within the function environment, R will *go up* the search path looking for a variable called `x`. Even when passing the same argument for `y`, different values of `x` will yield different results, which can be seen on the results of call 3 and call 4. This function is, therefore, not self-contained.

Call 5 Since the default value for `y` is `x + 1` and no argument is passed in the last call, the function returns the value for `x + (x + 1)`, where `x` is the defined in the global environment.