Computer lab 2

Måns Magnusson

August 24, 2015

Instructions

- This lab should be done individually
- It is ok to discuss the problems with each other, but to copy code is **NOT** allowed.
- The lab should be turned in through **LISAM** as a documented `.R` file.
- In the `.R` file two text element (objects) is mandatory:
  - `name` (your name)
  - `liuid` (your LiU-ID)
- The deadline for the lab be found at the course **webpage**

Contents

Automatic feedback with `markmyassignment`

As a complement to get fast feedback on your lab assignments the package markmyassignment has been written. This makes it possible to get automatic feedback on specified assignments, using any computer (altough internet access is required).

To install markmyassignment the package devtools is needed. To install devtools and markmyassignment just run the following code:

```
> install.packages("devtools")
> devtools::install_github("MansMeg/markmyassignment")
```

To get automatic feedback on your assignment you need an assignment path from your teacher. To set the assignment just run the following code

```
> library(markmyassignment)
> set_assignment("[assignment path]")
```

where `[assignment path]` äis the path given to you by your teacher.

To see which tasks that are included in the lab you can use the function `show_tasks()` in the following way:

```
> show_tasks()
```

To get automatic feedback you use the function `mark_my_assignment()`. To get feedback on all assignments just run the function.

```
> mark_my_assignment()
```

Remember that the functions need to be in the global environment in R.

You can also get feedback on specific tasks in the following way:

```
> mark_my_assignment(tasks="foo")
> mark_my_assignment(tasks=c("foo", "bar"))
```

It is also possible to correct an .R file in the following way:

```
> mark_my_assignment(mark_file = "[my search path to file]")
```

where `[my search path to file]` is the search path to the file to get feedback on.

Note! When an .R-file is checked, the global environment needs to be empty. Use `rm(list=ls())` fto clean the global environment.

Chapter 1

Lab assignments

To use `markmyassignment` run the following code:

```
library(markmyassignment)

Loading required package:  methods
Loading required package:  yaml
Loading required package:  testthat
Loading required package:  httr

lab_path <-
"https://raw.githubusercontent.com/MansMeg/AdvRCourse/master/Labs/Tests/lab2.yml"
set_assignment(lab_path)

Assignment set:
Lab2 :  Advanced R programming, computer lab 2
```

1.1 Conditional statements

1.1.1 `sheldon_game(player1, player2)`

In the series The big bang theory the character Sheldon comes up with a new version the game Rock-paper-scissors, called Rock-paper-scissors-lizard-spock. The detailed rules of the game can be found **here**.

Create a function you call `sheldon_game()` that should take two arguments, `player1` and `player2`. Each player should choose one of the choices `rock`, `paper`, `scissors`, `lizard` or `spock`. If another choice is made, the function should be stopped (with the `stop()` or `stopifnot()` function).

The function should return either `‘‘Player 1 wins!’’`, `‘‘Player 2 wins!’’` or `‘‘Draw!’’`.

Tip! This can be solved by using the cyclical structure of the game and the modulus operator.

```
> sheldon_game("lizard", "spock")

[1] "Player 1 wins!"

> sheldon_game("rock", "paper")

[1] "Player 2 wins!"
```

1.2 `for` loops

1.2.1 `my_moving_median()`

Create a function that will create a rolling median from a given numeric vector. Call the function `my_moving_median()`. The functions should accept a given numeric vector `x`, and a numeric scalar `n`.

The function should check if the arguments is a numeric vector (`x`) and a numeric scalar (`n`) and if not, it should stop the function (with the `stop()` or `stopifnot()` function). Otherwise the function should return the following vector:

$$y_t = \text{median}(x_t + \cdots + x_{t-n})$$

where y_t is the tth elementet. It should also be possible to send argument to the `median()` function (as `na.rm`).

```
my_moving_median(x = 1:10, n=2)

[1] 2 3 4 5 6 7 8 9

my_moving_median(x = 5:15, n=4)

[1]  7  8  9 10 11 12 13

my_moving_median(x = c(5,1,2,NA,2,5,6,8,9,9), n=2)

[1]  2 NA NA NA  5  6  8  9

my_moving_median(x = c(5,1,2,NA,2,5,6,8,9,9), n=2, na.rm=TRUE)

[1] 2.0 1.5 2.0 3.5 5.0 6.0 8.0 9.0
```

1.2.2 `for_mult_table()`

Write a function called `for_mult_table()` with arguments `from` and `to`. Both arguments can be assumed to be positive integers. The function should return a multiplication table as a matrix from the argument `from` to the argument `to`. The column names and row names should contain the factors specified by the arguments.

The function should stop (using `stop()`) if the arguments is not numeric scalars.

See an example below.

```
for_mult_table(from = 1, to = 5)

  1  2  3  4  5
1 1  2  3  4  5
2 2  4  6  8 10
3 3  6  9 12 15
4 4  8 12 16 20
5 5 10 15 20 25

for_mult_table(from = 10, to = 12)

    10  11  12
10 100 110 120
11 110 121 132
12 120 132 144
```

1.2.3 * `cor_matrix()`

Create a function called `cor_matrix()`. It should take a numeric `data.frame` with numerical variables of an arbitrary size as an argument `X` and calculate the correlation matrix for the given variables. If another object than an data.frame is used as argument, the function should return an error.

In this exercise you are not allowed to use the functions `var()`, `sd()` or `cor()` in your function.

See an example below.

```
data(iris)
cor_matrix(iris[,1:4])

         [,1]     [,2]     [,3]     [,4]
[1,]  1.00000 -0.11757  0.87175  0.81794
[2,] -0.11757  1.00000 -0.42844 -0.36613
[3,]  0.87175 -0.42844  1.00000  0.96287
[4,]  0.81794 -0.36613  0.96287  1.00000

data(faithful)
cor_matrix(faithful)

        [,1]    [,2]
[1,] 1.00000 0.90081
[2,] 0.90081 1.00000
```

1.3 while loops

1.3.1 find_cumsum()

Create a functions that takes two arguments `x` and `find_sum`. The function should calculate the cumulative sum of the numeric vector `x` and stop when the cumulative sume is greater than `find_sum` or if the vector `x` is traversed and return this value. The function should assert that x is a numeric vector and is a numeric scaler (with stop() or stopifnot()).

It is not allowed to use a `for` loop in this assignment.

```
find_cumsum(x=1:100, find_sum=500)

[1] 528

find_cumsum(x=1:10, find_sum=1000)

[1] 55
```

1.3.2 while_mult_table()

Create a functions that calulates a given mutiplication table similar to the exercise 1.2.2 but now you shoul use a (nested) while loop instead of a for-loop. The result should be identical in all other respect.

```
while_mult_table(from = 3, to = 5)

   3  4  5
3  9 12 15
4 12 16 20
5 15 20 25

while_mult_table(from = 7, to = 12)

    7  8   9  10  11  12
7  49 56  63  70  77  84
8  56 64  72  80  88  96
9  63 72  81  90  99 108
10 70 80  90 100 110 120
11 77 88  99 110 121 132
12 84 96 108 120 132 144
```

1.3.3 * trial_division_factorization()

Write a function that can factorize any integer number x inte its prime components. The function should return a vector of all the prime factors as a numeric vector. The idea is to simply test all integers (or prime numbers) from 2 to $\sqrt{n}$. For more detailed information on how to factorize primes using the trial division algorithm can be found **here**.

```
trial_division_factorization(x = 2^3 * 13 * 17 * 31)

[1]  2  2  2 13 17 31

trial_division_factorization(x = 47 * 91 * 97)

[1]  7 13 47 97
```

1.4 repeat and loop controls

1.4.1 repeat_find_cumsum()

Implement a function similar to `find_cumsum()` in 1.3.1 but using `repeat{}` instead of a `while` loop.

```
repeat_find_cumsum(x=1:100, find_sum=500)

[1] 528

repeat_find_cumsum(x=1:10, find_sum=1000)

[1] 55
```

1.4.2 repeat_my_moving_median()

Implement a function similar to `my_moving_average()` in 1.3.1 but using `repeat{}` instead of a `while` loop.

```
repeat_my_moving_median(x = 1:10, n=2)

[1] 2 3 4 5 6 7 8 9

repeat_my_moving_median(x = 5:15, n=4)

[1]  7  8  9 10 11 12 13

repeat_my_moving_median(x = c(5,1,2,NA,2,5,6,8,9,9), n=2)

[1]  2 NA NA NA  5  6  8  9
```

1.5 Environment

1.5.1 in_environment()

Create a function called `in_environment()` that return the contents of an environment as a text vector. The function shoulde take the argument `env` that takes an environment name as a text element.

```
env <- search()[length(search())]
env

[1] "package:base"
```

```
funs <- in_environment(env)
funs[1:5]

[1] "!"          "!.hexmode" "!.octmode" "!="        "$"
```

1.5.2 * where()

This function is similar to the where() function described in Advanced R. But unlike the implementation in the book you are not allowed to use `parent.env()` to traverse the search path. The function should take a name and return the name of the environment where the object is found by following the search path.

If the function is not found, the function should return the text element ``[fun] not found!''.

Assert that `fun` is a character vector of length 1.

```
where(fun = "sd")

[1] "package:stats"

where(fun = "read.table")

[1] "package:utils"

where(fun = "non_existant_function")

[1] "non_existant_function not found!"
```

1.6 Functionals

1.6.1 cov()

This exercise is taken from the Advanced R book. Use `lapply()` and an anonymous function to find the coefficient of variation (the standard deviation divided by the mean) for all columns in a given `data.frame`. The function should return a vector of the coefficients of variation.

Assert that `X` is a `data.frame`.

```
data(iris)
cov(X = iris[1:4])

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
     0.14171      0.14256      0.46974      0.63555

cov(X = iris[3:4])

Petal.Length  Petal.Width
     0.46974      0.63555
```

1.7 Clojures

1.7.1 moment()

This exercise has been taken from the Advanced R book. Create a function that creates a function for the ith central moment for a given variable. Definitions of the central moments can be found here.

```
m1 <- moment(i=1)
m2 <- moment(i=2)
m1(1:100)
```

```
[1] 0

m2(1:100)

[1] 833.25
```

1.7.2 * mcmc_counter_factory()

In markov chain monte carlo methods we often want to use counters for the sampling algorithms. Create a counter factory that creats a clojure with a counter. The factory should take two arguments, thin and burnin. Each counter should contain three objects:

- iteration (numeric) scalar
- store_sample (boolean) scalar
- samples (numeric) scalar

Assume that the counter is called in each iteration so in each call of the counter (without argument) should bump up the iterations object.

The store_sample boolean indicates whether the current iterations should be stored. Iterations after the number of burnin iterations should be stored every thin iteration. So if we run 20 iterations with burnin = 10 and thin = 3 we would store iteration {13, 16, 19}. The samples scalar should keep track of the number of stored iterations (three as the example). In each iteration the values should be returned as a list.

Assert that `burnin` is a nonnegative value and that `thin` is a positive value.

Below is an example.

```
mcmccnt <- mcmc_counter_factory(burnin = 3, thin = 2)
mcmccnt()

[[1]]
[1] 1

[[2]]
[1] FALSE

[[3]]
[1] 0

mcmccnt()

[[1]]
[1] 2

[[2]]
[1] FALSE

[[3]]
[1] 0

mcmccnt()

[[1]]
[1] 3

[[2]]
[1] FALSE

[[3]]
[1] 0
```

```
mcmccnt()

[[1]]
[1] 4

[[2]]
[1] FALSE

[[3]]
[1] 0

mcmccnt()

[[1]]
[1] 5

[[2]]
[1] TRUE

[[3]]
[1] 1

mcmccnt()

[[1]]
[1] 6

[[2]]
[1] FALSE

[[3]]
[1] 1
```

.