Computer lab 1

Måns Magnusson

13th August 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/lab1.yml"
set_assignment(lab_path)

Assignment set:
Lab1 :  Advanced R programming, computer lab 1
```

1.1 Vectors

1.1.1 `my_num_vector()`

Create a function called `my_num_vector()` **without** parameters. The function should do the following calculations and return the vector below.

$$\left(\log_{10} 11, \cos\left(\frac{\pi}{5}\right), e^{\pi/3}, (1173 \mod 7)/19\right)$$

In the example the example below the values has been rounded to fewer decimals. Your functions should return "all" decimals.

```
> my_num_vector()

[1] 1.04139 0.80902 2.84965 0.21053
```

1.1.2 `filter_my_vector(x, leq)`

Create a function called `filter_my_vector()` with the argument `x` and `leq` (less or equal to). The function should take a vector `x` and set all values larger or equal to `leq` to missing value (`NA`).

See an example below.

```
> filter_my_vector(x = c(2, 9, 2, 4, 102), leq = 4)

[1]  2 NA  2 NA NA
```

1.1.3 `dot_prod(a, b)`

Create a function called `dot_prod()` that computes the dot product between two vectors, `a` and `b`. The dot product is calculated in the following way:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

This should be done using only vector arithmetics and statistical functions. More information on the dot product can be found **here**.

```
> dot_prod(a = c(3,1,12,2,4), b = c(1,2,3,4,5))

[1] 69

> dot_prod(a = c(-1,3), b = c(-3,-1))

[1] 0
```

1.1.4 `approx_e(N)`

The constant e can be described with the following infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

We use this series to approximate e by taking an arbitrarily large value N:

$$e = \sum_{n=0}^{N} \frac{1}{n!}$$

Create a function called `approx_e()` with the parameter `N` to approximate e. Test how large `N` need to be to approximate e to the fifth decimal place.

```
> approx_e(N = 2)

[1] 2.5

> approx_e(N = 4)

[1] 2.7083

> exp(1)

[1] 2.7183
```

1.2 Matrices

1.2.1 `my_magic_matrix()`

Create a function called `my_magic_matrix()` without any parameters that creates and returns the following magic matrix.

$$\begin{pmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{pmatrix}$$

Can you see what's magic about it?

```
> my_magic_matrix()

     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6
```

1.2.2 calculate_elements(A)

Create a function called `calculate_elements(A)` that can take a matrix of an arbitrary size and calculate the number of elements in the matrix.

See examples below:

```
> mat <- my_magic_matrix()
> calculate_elements(A = mat)

[1] 9

> new_mat <- cbind(mat, mat)
> calculate_elements(A = new_mat)

[1] 18
```

1.2.3 row_to_zero(A, i)

Create a function called `row_to_zero(A, i)` that can take a matrix of an arbitrary size and set the row indexed with `i` to zero.

See examples below:

```
> mat <- my_magic_matrix()
> row_to_zero(A = mat, i = 3)

     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    0    0    0

> row_to_zero(A = mat, i = 1)

     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    3    5    7
[3,]    8    1    6
```

1.2.4 add_elements_to_matrix(A, x, i, j)

Create a function called `add_elements_to_matrix(i)` with parameters `A`, `x`, `i`, `j` . The function should take a matrix `A` of an arbitrary size and add the value `x` to the parts of `A` indexed by `i` and `j`.

See an example below:

```
> mat <- my_magic_matrix()
> add_elements_to_matrix(A = mat, x = 10, i = 2, j = 3)

     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5   17
[3,]    8    1    6
```

```
> add_elements_to_matrix(A = mat, x = -2, i = 1:3, j = 2:3)

     [,1] [,2] [,3]
[1,]    4    7    0
[2,]    3    3    5
[3,]    8   -1    4
```

1.3 Lists

1.3.1 `my_magic_list()`

Create a function called `my_magic_list()` without parameters that creates and returns a list with three list elements. The first should contain a text element with the text ‘‘`my own list`’’. The second element should be the vector generated by the function `my_num_vector()` above and the third element should be the matrix generated by `my_magic_matrix()` above.

The first list element should be named `info`.

This is how the list should look.

```
> my_magic_list()

$info
[1] "my own list"

[[2]]
[1] 1.04139 0.80902 2.84965 0.21053

[[3]]
     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6
```

1.3.2 `change_info(x, text)`

Create a function that will take a list `x` (that must contain one element with name `info`) and change this element to the text argument given by `text`.

See an example below:

```
> a_list <- my_magic_list()
> change_info(x = a_list, text = "Some new info")

$info
[1] "Some new info"

[[2]]
[1] 1.04139 0.80902 2.84965 0.21053

[[3]]
     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6
```

1.3.3 `add_note(x, note)`

Create a function that will take a list `x` and add a new list element with the name `note`. This new element should contain text from the `note` parameter.

See an example below:

```
> a_list <- my_magic_list()
> add_note(x = a_list, note = "This is a magic list!")

$info
[1] "my own list"

[[2]]
[1] 1.04139 0.80902 2.84965 0.21053

[[3]]
     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6

$note
[1] "This is a magic list!"
```

1.3.4 sum_numeric_parts(x)

Create a function called **sum_numeric_parts()** that will take a list **x** and sum together all numeric elements in this list. In a simple implementation you will get warning messages seen below.

```
> a_list <- my_magic_list()
> sum_numeric_parts(x = a_list)

Warning in sum_numeric_parts(x = a_list):  NAs introduced by coercion

[1] 49.911

> sum_numeric_parts(x = a_list[2])

[1] 4.9106
```

1.4 data.frames

1.4.1 my_data.frame()

Create a function that generates a data.frame that has the following variables and elements.

```
> my_data.frame()

  id name income  rich
1  1 John   7.30 FALSE
2  2 Lisa   0.00 FALSE
3  3 Azra  15.21  TRUE
```

1.4.2 sort_head(df, var.name, n)

Create a function called **sort_head**() that takes a **data.frame** as parameter **df** and returns the **n** largest values for the given variable **var.name**. All variables should be returned.

Se below for an example of the function.

```
> data(iris)
> sort_head(df = iris, var.name = "Petal.Length", n = 5)

    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
119          7.7         2.6          6.9         2.3 virginica
118          7.7         3.8          6.7         2.2 virginica
123          7.7         2.8          6.7         2.0 virginica
106          7.6         3.0          6.6         2.1 virginica
132          7.9         3.8          6.4         2.0 virginica
```

1.4.3 add_median_variable(df, j)

Create a function called `add_median_variable()` should take a `data.frame` and a column id `j`. The function should compute the median for this variable and create a new variable called `compared_to_median` in the `data.frame`. All values that are greater than the median should have the text label ``Greater'', the values that are smaller should have the label ``Smaller''. The element that is the median (can happen) should have the label ``Median''.

Below is an example using the dataset `faithful`.

```
> data(faithful)
> head(add_median_variable(df = faithful, 1))

  eruptions waiting compared_to_median
1     3.600      79            Smaller
2     1.800      54            Smaller
3     3.333      74            Smaller
4     2.283      62            Smaller
5     4.533      85            Greater
6     2.883      55            Smaller

> tail(add_median_variable(df = faithful, 2))

    eruptions waiting compared_to_median
267     4.750      75            Smaller
268     4.117      81            Greater
269     2.150      46            Smaller
270     4.417      90            Greater
271     1.817      46            Smaller
272     4.467      74            Smaller
```

1.4.4 analyze_columns(df, j)

Create a function you call analyze_columns that should take a `data.frame` called `df` and two column ids in a vector `j` of length 2. These two columns should be analyzed and the results should be returned as a list with three elements. The first two should contained a named vector with the mean, median and the sd for each of the variables. The third element should contain the correlation matrix between the two columns.

The list should be named with the variable names (first two list elements) and the last element should be called ``correlation_matrix''. Below is a couple of examples:

```
> data(faithful)
> analyze_columns(df = faithful, 1:2)

$eruptions
  mean median     sd
3.4878 4.0000 1.1414
```

```
$waiting
  mean median     sd
70.897 76.000 13.595

$correlation_matrix
           eruptions waiting
eruptions    1.00000 0.90081
waiting      0.90081 1.00000

> data(iris)
> analyze_columns(df = iris, c(1,3))

$Sepal.Length
   mean  median      sd
5.84333 5.80000 0.82807

$Petal.Length
  mean median     sd
3.7580 4.3500 1.7653

$correlation_matrix
             Sepal.Length Petal.Length
Sepal.Length      1.00000      0.87175
Petal.Length      0.87175      1.00000

> analyze_columns(df = iris, c(4,1))

$Petal.Width
   mean  median      sd
1.19933 1.30000 0.76224

$Sepal.Length
   mean  median      sd
5.84333 5.80000 0.82807

$correlation_matrix
             Petal.Width Sepal.Length
Petal.Width      1.00000      0.81794
Sepal.Length     0.81794      1.00000
```