Introduction to R – session 1

Indrek Seppo and Nicolas Reigl

Febr 9, 2018

Preparation of these materials was supported by HITSA – Estonian Information Technology Foundation for Education.

Introduction

The aim of the first sessions is to give you the very basic knowledge of R, about the building blocks of R. This should be almost enough for you to be able to start working in R on your own. It should cover the basics that is assumed to be known by help pages and specific tutorials on the internet – I expect you to be able to more or less follow them after the first couple of session of our course.

We will then continue towards the modern workflow in R, using some more recent add-ons (or packages, as they are called in R). We will cover data wrangling, convenient aggregation, filtering, slicing. We will look at data visualisation with grammar of graphics and how to create professional looking reports and articles straight from R[1].

R will feel different at first if you have never used a scripting language, and it does have a learning curve. We will ascend the steepest part of it during the first sessions. The worse will be over by then and we can continue with stuff that is much more fun, reaching the elegant features of modern R that have made R the language of choice in many fields.

[1] This handout was written entirely in RStudio.

Brief intro to R

R is open source software – free to use for whichever purpose (including commercially). You have additional right to copy it for a friend, and you can look at the current code base and propose your own changes to it. R will stay free forever – it has a quazillion of developers, all of whom should approve the change of license – not gonna happen.

The development of R started in 1993, but it started as an open source implementation of commercial S-language, first version of which was created at Bell Lab's in 1976. This legacy has it's downsides. So does the fact that R has been developed by statisticians rather than programmers (R is a collaborative effort, especially as most of the work is not done with the "base R", but with user-developed additional packages). This means that R can be inconsistent and it definitely has its quirks.

R started to quickly gain popularity during the 2000-s, and is by now the most used special purpose data analysis environment or language. It is actually so popular that has in recent years been in the top 20 of Tiobe index, which tracks the visibility of programming languages in the internet – quite a feat for a domain-specific language.

There are other languages emerging to the same niche:

- Julia[2] – domain-specific (specifically written for data analysis tasks, just like R) language, which is cleaner and more consistent than R.
- Python – very popular general purpose programming language, used quite a lot for scientific computing. Has in recent years vastly expanded it's statistical capabilities (following closely the logic of R). Again – way more consistent, faster, without some of the R-s limitations.

[2] http://julialang.org/

This said – R is the *lingua franca* of statistics. New statistical methods get typically implemented first in R. Over ten thousand user-created libraries mean that however specific your problem is, you will probably find a package that does what you need, without the need to implement it in matrix algebra yourself. R is the language all of the math-stats students are learning, it is pretty much a standard in many fields of science (and more and more the language of choice in statistcs courses at universities), it is written specifically for people who just want to do some data analysis, it tries to make it as easy as possible. You would usually use Python for large machine learning projects and for implementing some statistics into your data-driven products; you would usually use R for research and exploratory data analysis that do not have the highest performance requirements, and most of the data scientists use both. As Python and R have been borrowing from each other a lot (the statistical tools in Python are mostly modelled from R) most of what you will be learning in R can be translated very easily (sometimes "word-by-word") to Python.

I have not touched the programs you might have used to the most in your previous studies – commercial software like SPSS, Stata, SAS. They all have their strenghts and weaknesses. The one weakness they all share is the price-tag. But there are also stenghts: I have myself just recently used Stata for some dynamic panel VAR modelling and I think every aspiring economist should be able to use it – it is still the *de facto* standard in central banks (but R is already being used there as well); SAS can handle much larger datasets by default and statistic offices will continue to use it (well – they still use some ancient FoxPro stuff); MPlus is better at structural equation modelling (but R is not so far behind), thus might be for you if you do some very advanced or recent stuff in factor analysis. I would not consider the point and click user interface a particular strength, the moment you will start doing something real, you will be writing scripts in these programs as well. And the logic of reproducible research actually requires to have a mark left for every action you do.

Then there are a lot of business analytics tools, which try to automate some of the tasks as much as possible. Many of them have R bindings – you can write straight R code into them if you want them to do some advanced tricks.

RStudio

RStudio is the most used user interface for R (there are others, and one can even do without – R itself has a rudimentary script editing environment). It is

distributed as open source software[3], just like R, so one can download and use it for any purpose – for personal, academic or commercial gain. The company developing RStudio was bought by Microsoft in 2015, but there are no signs that the licensing model would change, on the contrary – the development has accelerated.

RStudio works both as a standalone program in users computer, or – if you ever find yourself working with really huge data sets, you can install it on a server[4] and use it over a web interface.

[3] You can buy a commercial license, which gives you an assurance that someone will respond your emails in 8h, for a $1000 a year. It is highly unlikely that you ever find yourself in need for that.

[4] Only Linux servers are currently supported.

Managing projects in RStudio

It is highly recommended to organize your work in projects in RStudio. Each project is associated with its own working directory from where it tries to read and save the files by default. You can check the directory on the titlebar as well as on the Files-tab in the bottom right corner of RStudio[5]. When you close your project, RStudio will automatically save the state of the project – all of the open files, all of the data objects you have created, the history of every function you have used etc – these will be read in when you open the project again. You can have many projects for all the stuff you are working on without one interrupting the other.

[5] As long as you have not browsed around – you can quickly go back to the working directory by clicking More -> Go to working directory.

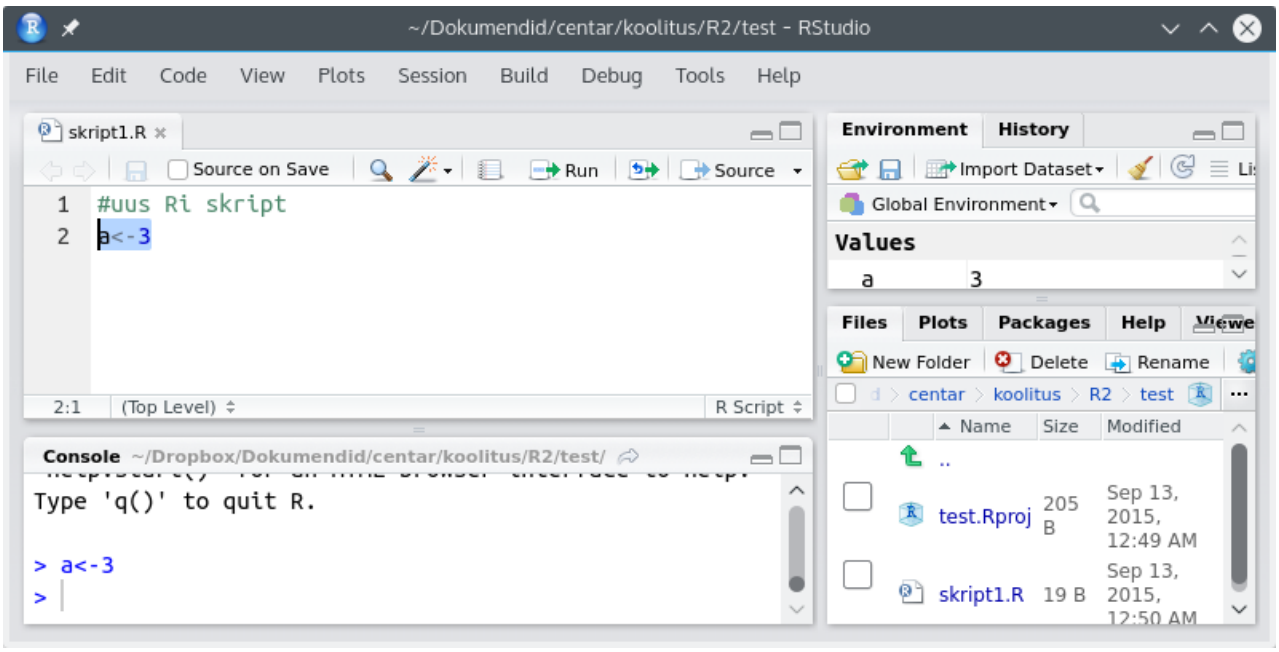

Figure 1: RStudio

You can create a new project from the menu 'File -> New Project', open a recent project with 'File -> Recent Projects' or 'File -> Open Project'. For a new project you will usually create a new directory, but it needs to know where to

create it[6].

After the project is created, you will need to create a script file. Chooose 'File - > New File -> R Script' from the menu. You can now save it ('Cntrl+S', 'File->Save or using the traditional floppy-disc icon). You can have many script-files open in parallel, they will be opened in different tabs.

[6] NB! If your computer's home directory is on a network drive, so that it's address is something like `\\domain\username`, then R will need some additional steps to work. A solution is to map the drive under some letter (*e.g.* `X:/`) and tell Rstudio the path using this letter (`X:/myproject`), not the `\\domain\username\myproject` path. This is the case with using the university computers here.

Working in RStudio and some basic arithmetics in R

Working in R means writing to a script file and then sending your commands to be executed in the console. For this select the part of the script you want to execute and either press Ctrl+Enter or click on the Run-button above. If nothing is selected the row where the cursor is will be run.

Lets try it by using R as a common calculator:

Multiplication, dividing, adding and substracting

```
4 * 5
4 * (2 + 4)  #NB! you need the explicit * before the brackets
8/2
3 - 4
3 + 5
```

Exponentiation, logarithms

```
2^3
sqrt(16)
16^(1/3)
log(10)  #NB! log() takes natural logarithm in R!
```

YOUR TURN

Your turn:

- Write some random calculations to the script file and send them to be executed in console.

You can, in fact, write directly to the console (which is below the scripts-panel), but the very essence of R is the **reproduducible research** approach – everything that is important will be recorded in the script file and we can easily rerun this script later, make some refinments if needed etc.

At the top-right of RStudio user interface we see a pane called History. History will record every line we use (even if we have deleted it from the script-file).

The basics of using R

We will start with how to deal with data objects. Data object can be a single value or whole dataset[7], which is usually in the form of *data frame* in R. *Data frame* itself consists of data columns or vectors. Then continue to how the functions work and how to access the help system.

[7] Or a picture or anything else.

Data objects and assignment

Assigning a value to a data object is usually done with the following operator[8]: "<-".

[8] That one of the main operators in R actually uses two characters is a historical curiosity. The creators of S language, an ancestor of R, used computers where there was actually a specific key for this sign. The key disappared, the sign that drives every modern programmer crazy, stayed.

```
a <- 3 #assign a value to a data object
print(a) # show the content of the data object
a #the same
a = 4 #Not a good style, but usually works
a <- 3 + 2 #the result will be written in the data object a
a <- a + 4 #the new value will be assigned to a
2 -> a #or we can do it this way
```

NB! R will not display you anything if it is written in the data object. **R follows what is called "silence is blissed"-philosophy – it will only let us know if something went wrong.** Nothing will be displayed if everything works. To see what is in the data object, we have to explicitly ask R to show it to us. The data objects that have been created will be displayed in the "Environment"-tab top right in the RStudio.

Requirements for object names.

Data objects in R **can not start with a number, include "-"-sign, or other arithmetic signs, it cannot contain spaces either** (well, you will later see, that it actually can :)). Pretty much everything else is allowed, *e.g.* `name.of.the.bear`, `teddys_color`. You can have numbers everywhere but the very beginning of the name (so `4bears` does not work, `bears4` does), you can use special characters, up to the hieroghlyphs but this can introduce problems if you are sending the file to someone else (especially in different platforms – going from Windows to Mac etc). In case you see some unknown symbols in the R code, you can change the text-encoding in RStudio (Tools -> Project Options -> Code Editing -> Text Encoding -> Change), UTF-8 is supported on all platforms (but Windows does not use it by default for some reason).

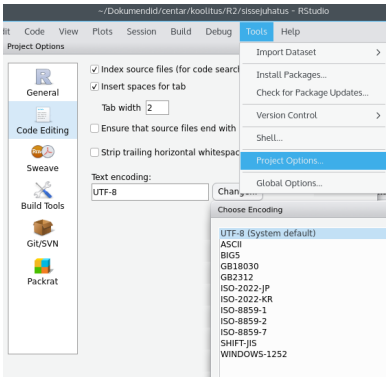

Figure 2: Changing the encoding of the text

R is case-sensitive – dataset $\neq$ dAtaset

```
nrofdogs <- 5
nrOfDogs <- 3
nrofdogs
```

```
## [1] 5
```

```
nrOfDogs  #two different variables!
```

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

You can use data objects just like the values they contain:

```
a <- 5
a + 3
```

```
## [1] 8
```

To remove a data object you can use the command `rm()`

```
rm(a)  #removing a data object
```

How to mange data objects should more or less clear now. We have assigned single values to them, but will later see that an object can actually contain pretty much anything in R – whole datasets, results of the analysis, graphs. The same logic works everywhere. If we want to write something into an R object, so that we could use it later, we can assigne it with the "<-" operator.

YOUR TURN

Your turn:

- create a variable called `nrOfChickens` ja give it a value of 4, create a variable called `nrOfDogs` and assign it a value of 5, create a variable called `nrOfAnimals` and assign it a value of `nrOfChickens + nrOfDogs`.
- remove the object `nrOfDogs`.

Data vectors

Data vectors

Vectors or data columns are the basic building blocks of R. Everything in R is a vector. A single value is just a vector of a size of one.

Creating a data vector is done in the following way (note that the **values are delimited by commas, and the decimal point is a point**):

```
c(3, 4, 5.6, 7)
```

```
## [1] 3.0 4.0 5.6 7.0
```

There are a number of shorthands, for example ":" creates a vector of adjascent numbers:

```
longvector <- c(3:65)
longvector
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
## [26] 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
## [51] 53 54 55 56 57 58 59 60 61 62 63 64 65
```

We can now see what the number in the beginning of rows does – it shows the position of the first element in this row in the data vector.

YOUR TURN

Your turn:

- create a data object called `length` and assign to it a vector containing the following values 5, 6, 8 ja 10.

If you call it, the result should look like this:

```
length
```

```
## [1]  5  6  8 10
```

Vector needs to contain the same type of data. *E.g.* only numbers, only text, only dates. You can concatenate vectors with `c()` to create a longer vector.

```
c(length, 10, 7)  #new vector: length and 10 and 7
```

```
## [1]  5  6  8 10 10  7
```

Note that it does not overwrite the data object `height` as we did not ask for it, it just shows us the result in the console.

Lets create another vector, with some names of animals this time:

```
animal <- c("cat", "dog", "bear", "elephant")
```

Note that the names are inside quotation marks – they are textual values (character type in R). Would they not be in quotation marks, R would assume that we want to concatenate data objects named cat, dog, bear and elephant.

Pretty much everyting in R is vectorized. If we add some number to the vector `length` it will be added to all of its elements. If we multiply it by some number, every element will be multiplied by it.

For example:

Couple of tips for R console – up and down arrow will allow you to move between earlier commands. Tabulator helps to finish the command (it will suggest the function names and objects starting with what was written)).

```
length
```

```
## [1]  5  6  8 10
```

```
length + 5
```

```
## [1] 10 11 13 15
```

```
length * 2
```

```
## [1] 10 12 16 20
```

If we add two vectors of the same length, the first element of first vector will be added to the first element of second, the second element to the second element etc. The same with multiplication.

```
length + c(1, 2, 3, 4)
```

```
## [1]  6  8 11 14
```

Vector reuse

One of the peculiarities of R is that vectors are "reused":

```
c(1, 2, 3, 4) + c(1, 2, 3)
```

```
## Warning in c(1, 2, 3, 4) + c(1, 2, 3): longer object length is not a multiple of
## shorter object length
```

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

When the second vector ends, we will start from the beginning again! At least it warns us here, but if the vector lengths are multiples of each other, we will not even be warned:

```
c(1, 2, 3, 4) + c(1, 2)
```

```
## [1] 2 4 4 6
```

Named vectors

Named vectors

One useful type of vector is a *named vector* – a vector where every element has a name. We can then address the elements by their names:

```
person <- c(name = "Masha", surname = "Mishka")
person
```

```
##     name  surname
##  "Masha" "Mishka"
```

```
person["name"]
```

```
##    name
## "Masha"
```

Some of the R functions want to get this kind of vectors as an input, we will look closer at these when the time is right.

Data frames

We dont usually use a single datacolumn, but have a number of them together in a table. This kind of data table is called a *data frame* in R.

We have previously created vectors **animal** and **length**. Lets tie them together with a command `data.frame()` and write it to a data object called **mydata**.

Creation of a data frame

```
mydata <- data.frame(animal, length)
mydata
```

```
##     animal length
## 1      cat      5
## 2      dog      6
## 3     bear      8
## 4 elephant     10
```

We have now created our first data frame, where every animal has its length! Kind of like a Excel work sheet.

We can look at the data frame by `View(andmed)` – it will be shown in a more convenient way.

We can think of the dataobjects (seen in the Environment pane in RStudio) like different worksheets in Excel – some have only a single value in them, some have a whole dataset, some can have multiple datasets (we will see list-type data later), some may even contain a graph.

Lets remove vectors **animal** and **length**, so that they will not disturb us later on:

```
rm(animal)
rm(length)
```

To access a single variable or column in the data frame, we will use the $-sign. Eg:

Accessing the variables in data frames

```
mydata$animal
```

```
## [1] cat      dog      bear     elephant
## Levels: bear cat dog elephant
```

```
mydata$length
```

```
## [1]  5  6  8 10
```

We can add additional columns to the data frame:

Adding variables to data frames

```
mydata$width <- c(3, 3, 1, 1)
mydata
```

```
##     animal length width
## 1      cat      5     3
## 2      dog      6     3
## 3     bear      8     1
## 4 elephant     10     1
```

or

```
mydata$lengthsquared <- mydata$length^2
```

Another way to access the variables is using square brackets, the name of the variable must be quoted:

```
mydata["animal"]
```

```
##     animal
## 1      cat
## 2      dog
## 3     bear
## 4 elephant
```

We can give it not one, but as many names as we want, by using a vector of names (note the `c()`!):

```
mydata[c("animal", "width")]
```

```
##     animal width
## 1      cat     3
## 2      dog     3
## 3     bear     1
## 4 elephant     1
```

We can create a new data object from the selected variables:

```
animalwidth <- mydata[c("animal", "width")]
```

YOUR TURN

Your turn:

- add a new variable called `volume` to the data frame `mydata`, so that it would be the product of the variables `length` and `width` from the same data frame.
- create a new data.frame (data object) named `animalvolume`, containing only two columns from the original data frame – `animal` and `volume`.

Removing variables from the data frame

To delete a variable from the data frame we can assign it a value of `NULL`. For example – would we discover that length times width might give us a surface area of our animals (in the improbable case that we do have two-dimensional animals), but not the volume of them, then we would probably want to get rid of the variable:

```
mydata$volume <- NULL
```

Commands or functions

We have already used a number of function calls: `c()`, `data.frame()`, `rm()`. The commands or functions have always the regular brackets () after them. Inside these brackets we can write arguments and parameters – what is the data we want to use this function on, do we want to add some additional parameters etc. `c()` only wanted to know which vectors to concatenate.

Lets look at a few more functions. For example `mean()`, that – you might have guessed it – finds a mean.

```
mean(mydata$length)
```

```
## [1] 7.25
```

You can access the help for every command using a command `help(command)` or just `?command`. Lets have a look what does it say about the `mean`:

Help system

```
?mean
```

You will now find the help info appearing in the right-bottom pane of RStudio. You will find the description of the function and some usage information – `mean(x, trim = 0, na.rm = FALSE, ...)`. In the **Arguments** sections the parameters and their effect is described. The `mean()` function will first need an argument `x`, which has to be an R object (numeric, logical, date or time vectors are supported). There are some additional parameters, but they have default values – we can overwrite them, pass our own value, but we do not have to do it if the default is what we want. There is an argument called `trim`, asking which proportion of highest and lowest values to remove before it finds the average. The default is 0, but we can change it:

```
mean(mydata$length, trim = 0.25)
```

```
## [1] 7
```

The second parameter – `na.rm`, which is by default set to FALSE, is a parameter we are going to use often. It asks what to do in case of missing values. The missing values are denoted by `NA` (*not available*) in R.

In addition to 'NA' there are some other special values in R: NaN – not a number (eg 0/0), +Inf (eg 8/0) and -Inf (-3/0).

Lets create a vector with a missing value:

```
missing <- c(4, 5, 6, NA)
```

What is the average of 4, 5, 6 and a number that we do not know? We have no idea! And this is what R correctly tells us:

```
mean(missing)
```

```
## [1] NA
```

But we can tell R to ignore the missing values by setting the parameter na.rm to be TRUE:

```
mean(missing, na.rm = TRUE)
```

```
## [1] 5
```

Another function that we are going to use quite often is `read.csv()` that helps us to read in csv-files.

Take a look at the help:

```
?read.csv
```

We see that it needs a file name as an argument and allows us to change a number of additional parameters – *e.g* whether there is a header in the dataset (by default it expects to see one: `header=TRUE`). `read.csv()` and `read.csv2` are in fact convenience-functions – they are the same as `read.table()` with some of the parameters pre-filled[9].

[9] If you save a data set from Excel as a csv-file, either `read.csv()` ja `read.csv2` will usually read it in, depending on your locale settings of Excel.

YOUR TURN

Your turn:

- find from the help system how to tell the read.csv() function that in your data:

1) the decimal point is ","
2) the separator is ";"

Lets read in a file called "gdpestimate.csv"[10]

[10] This file is originally compiled by Kaspar Oja from Estonian Central Bank and shows the difference between initial estimate of Estonian GDP growth and the estimate after later revisions. It is a bit dated by now.

```
read.csv("gdpestimate.csv")
```

```
##   year quarter firstEstimate latestEstimate       date
## 1 2003       3         0.043     0.06819747  2003-8-15
## 2 2003       4         0.058     0.07403148 2003-11-15
```

It reads it in and prints the result into the console. Just like R does. But we do not want this! We want to read it into an R data object, so that we could use it in R later on. Lets write the result into an object called `gdp`:

```
gdp <- read.csv("gdpestimate.csv")
```

Initial overview of a dataset

After we have read a new dataset, we would like to get a quick overview of it. First thing to do is usually `summary()`. `summary()` is a generic function – it works on pretty much every type of R objects. If you summarise a regression model, it will give you a nice regression output etc.

```
summary(gdp, digits = 1)
```

```
##       year         quarter  firstEstimate    latestEstimate          date
##  Min.   :2003   Min.   :1   Min.   :-0.166   Min.   :-0.19   2003-11-15: 1
##  1st Qu.:2006   1st Qu.:2   1st Qu.: 0.009   1st Qu.: 0.02   2003-8-15 : 1
##  Median :2009   Median :2   Median : 0.040   Median : 0.05   2004-11-15: 1
##  Mean   :2009   Mean   :2   Mean   : 0.030   Mean   : 0.03   2004-2-15 : 1
##  3rd Qu.:2012   3rd Qu.:3   3rd Qu.: 0.071   3rd Qu.: 0.08   2004-5-15 : 1
##  Max.   :2015   Max.   :4   Max.   : 0.120   Max.   : 0.12   2004-8-15 : 1
##                                                             (Other)   :42
```

For a dataframe, it will give an overview of each variable. For numerical variables it gives their minimum, maximum, mean and 25.th, 50.-th and 75.-th percentile. For factors it will give the count of occurrences. We see that the data is from 2003-2015, that the first estimate has been varying between -16% to +12%. We can thus quickly see if something went wrong during the data import etc.

Another important command is `str()`:

```
str(gdp)
```

```
## 'data.frame':    48 obs. of  5 variables:
##  $ year          : int  2003 2003 2004 2004 2004 2004 2005 2005 2005 2005 ...
##  $ quarter       : int  3 4 1 2 3 4 1 2 3 4 ...
##  $ firstEstimate : num  0.043 0.058 0.068 0.06 0.062 0.058 0.07 0.1 0.108 0.105 ...
##  $ latestEstimate: num  0.0682 0.074 0.1025 0.0329 0.0601 ...
##  $ date          : Factor w/ 48 levels "2003-11-15","2003-8-15",..: 2 1 4 5 6 3 8 9 10 7 ...
```

It tells us how big is the dataset (we can see this in the Environment pane as well); it tells the type of each variable (year and quarter are integers, firstEstimate and latestEstimate are numerical, date is of factor type – we will come back to this), and shows the first values.

`View(gdp)` will show us the whole dataset, but it is usually enough to take a peak at the first couple of rows. First six rows will be shown with `head()`, the last six with `tail()`, both can take a parameter `n=x`, to show as many rows as we'd like:

```
head(gdp, n = 2)
```

```
##   year quarter firstEstimate latestEstimate       date
## 1 2003       3         0.043     0.06819747  2003-8-15
## 2 2003       4         0.058     0.07403148 2003-11-15
```

```
tail(gdp, n = 2)
```

```
##    year quarter firstEstimate latestEstimate      date
## 47 2015       1         0.012     0.01133824 2015-2-15
## 48 2015       2         0.019     0.02021796 2015-5-15
```

In R we can combine functions, write them inside another – the outer function will be applied to the result of the inner function:

```
View(head(gdp, n = 10))
```

Some additional functions for inspecting the data are: `dim(gdp)` tells us the nr of rows and columns in the dataset, `names(gdp)` gives us the names of the variables, `attributes(gdp)` gives us metainfo – names of the variables, names of the rows etc.

Your turn:

- create a new variable named **difference** into the data frame **gdp**, so that it would contain the difference between `gdp$firstEstimate` and `gdp$latestEstimate`.
- find the average differences over the period.
- create a new variable called **absdifference** into the data frame, containing the absolute difference between the two variables. You can find the absolute value by using function `abs()`, so `abs(gdp$difference)` would give you the required value after the difference column is created.
- find the average error during the period (average absolute difference)
- find the correlation between firstEstimate and latestEstimate. You should use the function `cor()` which takes two parameters - the variables for which the correlation is to be found.
- Read in the data file piaac.csv (available from `http://www.ut.ee/~iseppo/piaac.csv`[11]), writing it to an data object called **piaac**.
- how big is the data set (how many rows, how many columns)?
- what are the minimum and maximum values of the **income**-variable?
- Bonus exercise for the quickest: the function `quantile()` helps to find all kinds of percentiles. Find the 10-th, 50-th and 90.-th percentile of the variable **income** in the **piaac** data set[12].

[11] I will be talking about this dataset in the seminar.

[12] Note that the Usage part of the quantile help uses a function seq() that creates a sequence of numbers, you do not want to use it yourself.

Saving data

There are a number of different ways to save your dataset for external use. Usually you would save it as a comma separated value (csv) file using `write.csv()` (or `write.csv2()`).

Write the gdp-data to disk:

```
write.csv(gdp, file = "gdpnew.csv")
```

You will see the newly created file in the project folder. If you open the file, you'll see that there are row numbers included.

Your turn: YOUR TURN

- find from the help of write.csv which parameter should you add so that it would write the data without row numbers and save the data without the row numbers.

Csv-file will lose some of the metainfo. Any R object can be written to disk using save(): `save(object, file="filename.RDat")` and later load again using load(): `load(file="filename.Rdat")`. NB! the object will be loaded with the same name as it was saved.

```
save(gdp, file = "gdpnew.Rdat")
rm(gdp)
gdp
```

```
## Error in eval(expr, envir, enclos): object 'gdp' not found
```

```
load(file = "gdpnew.Rdat")
```

You can actually save multiple objects together (`save(object1, object2, file="filename.RDat`)

If you want to save an object so that it could be later read in with any name, you should use `saveRDS()`

```
saveRDS(gdp, file = "gdpnew.rds")
gdp2 <- readRDS(file = "gdpnew.rds")  #it will now be named gdp2
```

You may need `dput()` ja `dget()`. Those will write the R objects into a text file so that all the meta-data and attributes will be saved. If you ever ask help from online forums, then this is the way to save your example data.

```
dput(gdp, file = "gdp.txt")
```

Data types

We have used numerical and textual (character-type) variables until now. To get the type of data object, you can use `class(objectname)` or `str(objectname)`. The important ones are also integers, logicals and factors, but there are many others.

```
mypet <- "cat"  #textual data (character type) inside quotation marks
class(mypet)
```

```
## [1] "character"
```

```
mypet <- cat  #wont work! look at the error message
c <- TRUE  #logical data can be TRUE/FALSE or T/F
class(c)
```

```
## [1] "logical"
```

```
class(gdp$year)  #R has decided it is an integer
```

```
## [1] "integer"
```

```
class(gdp$date)
```

```
## [1] "factor"
```

If we give R a vector of data, it will choose the data type automatically so that everything in the data could be expressed with it (remember, vector could only hold one type of data):

```
c(3, 4, "cat", 5)
```

```
## [1] "3"   "4"   "cat" "5"
```

To convert between the data types, use functions like: `as.*datatype*()`. To convert something to character type we use `as.character()` etc:

```
a <- 4
a + 5
```

```
## [1] 9
```

```
a <- as.character(a)  #convert a to character
a  #in the quotation marks - character
```

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

```
a + 5  #could not work
```

```
## Error in a + 5: non-numeric argument to binary operator
```

```
as.numeric(a) + 5  #back to numeric
```

```
## [1] 9
```

```
as.numeric(TRUE)  #TRUE is 1, FALSE is 0
```

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

Factors

One of the most dangerous type of data in R are factors. They are in fact integers with labels.

Read in the file **factortest.csv** and take a look at it:

```
factortest <- read.csv(file = "http://www.ut.ee/~iseppo/factortest.csv")
head(factortest)
str(factortest)
summary(factortest)
```

It seems ok, just looking at it, but `str()` and `summary()` show us that R has read the length variable in as a factor. The reason being, that there is a "-" inside the numbers – a rather common occurence in the data. We remember, that one can convert between datatypes, wit as.datatype() function, so lets try to convert it to numeric:

```
factortest$numbers <- as.numeric(factortest$length)
head(factortest, n = 5)
```

```
##     animal length numbers
## 1      cat   21.2      19
## 2      dog   21.1      18
## 3 elephant   22.6      22
## 4      cat   22.9      23
## 5      dog     14       6
```

R does not give us any warnings, but obviously the result is not what we wanted. R just gives us an absolutely meaningless order of factors.

To convert the factor levels correctly to numbers you should first convert them to character type and then to numeric:

Note the warning – it says that not all of the levels can be converted to numeric, and thus some of the values are now NA-s.

```
factortest$correctnr <- as.numeric(as.character(factortest$length))
```

```
## Warning: NAs introduced by coercion
```

```
head(factortest, n = 5)
```

```
##     animal length numbers correctnr
## 1      cat   21.2      19      21.2
## 2      dog   21.1      18      21.1
## 3 elephant   22.6      22      22.6
## 4      cat   22.9      23      22.9
## 5      dog     14       6      14.0
```

NB! this mistake is a very common one – you need to be extra careful whenever you are dealing with factors and `as.numeric()`. R tends to convert text to factors by itself every now and then – when reading in csv-data (you can

explicitly tell it not to - by using stringsAsFactor=FALSE argument), when you are creating a data.frame etc, so you will have to deal with it frequently.

Factors are still useful and an important type of data, as this is the only way we can change the order of how text appears in our graphs and tables. To see the current order, use `levels()`

```
levels(factortest$animal)
```

```
## [1] "cat"      "dog"      "elephant"
```

They are in alphabetical order by default, we can change the order manually using the `factor()`[13]:

[13] This is the base-R way of doing it, we will learn a much better way of dealing with factor variables, provided by an addon called forcats later.

```
factortest$animal<-factor(factortest$animal,
                          levels=c("dog", "elephant", "cat"))
levels(factortest$animal)
```

```
## [1] "dog"      "elephant" "cat"
```

But! If we manage to make a mistake in writing some factor levels, then **R will not give us any error messages or warnings!**. It will just create NA-s for all the levels where not in our function call. And we can miss this easily if there are some NA-s in our data anyway. You should always check that everything went correctly.

We can also tell R that a factor is an ordered factor (e.g. if we have some ordered scales in the data - like "good", "better", "the best"). R will then consider the variable to be ordered nominal variable and will automatically use the correct analytical methods.

```
factortest$animal <- factor(factortest$animal, levels = c("cat", "elephant",
    "dog"), ordered = TRUE)
levels(factortest$animal)
```

```
## [1] "cat"      "elephant" "dog"
```

```
factortest$animal[1]
```

```
## [1] cat
## Levels: cat < elephant < dog
```

Additional packages

There are a number of functions or commands in the base R, but we can add functionality by installing some addon packages, of which there are currently over 10000. To do this, select the Packages tab on the right-down pane in R and click Install. Lets install a package called `ggplot2`[14].

To use the functions in this package we will need to load it in:

[14] You can do it by writing `install.packages("ggplot2")` as well.

```
library(ggplot2)
```

You need to install packages once (actually once every couple of years, depending on major R updates), but it is necessary to read it in every time you restart R.

The additional packages are developing quickly, you probably want to update them to the latest version every now and then. For this use the "Update"-button.

Sometimes the addon packages "break". Not frequently, but I have run into it a number of times. If your analysis depends on the specific version of a package (especially when you use some packages that are in heavy development phase and can change a lot) you can save the state of your packages in your project by using Packrat – it will save all the packages you currently have and even if you update your packages for other projects, they will stay the same for the current project[15]. Packrat helps you to upgrade specific packages one by one, to go down to previously working versions etc. It is good for archiving some work you have done.

[15] NB! As it downloads every single package that is currently in use, the data requirements can grow quickly, couple of hundred MB is quite common.

Some of the packages offer so called vignette's, that give you an overview of the usage of the package, quite regularly there is an article published describing the package and its use.

The addon packages offer new functions, but sometimes the names of the functions collide. R will automatically use the function from the package that was last read in (but rereading the package again does not help - you have to detach it first! This can be done by taking away the check mark befor the package name in the Packages tab.). If we want to use a specific function from a specific library, no matter at which order we have loaded the libraries, then we can do it by writing `packagename::functionname()`

YOUR TURN

Your turn:

- install and load package **dplyr**

You should see the following message:

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

What it tells you, is that by default R will now use the functions `filter()`, `lag()`, `setdiff()` etc from dplyr, not from the base packages. If we want to use the base version of `filter()`, we would need to call it in the following way:
`stats::filter()`.

Dates

There are other data types, e.g. Date:

```
today <- as.Date("8-09-2016")
class(today)
```

```
## [1] "Date"
```

Lets change the type of *date*-variable in our *gdp* data to Date[16]:

[16] This is automagically possible, because `gdp$date` follows ISO standard ("YYYY-MM-DD")

```
summary(gdp$date, maxsum = 4)  #it was a factor-type variable
```

```
## 2003-11-15  2003-8-15 2004-11-15    (Other)
##          1          1          1         45
```

```
gdp$date <- as.Date(gdp$date)  #change it to date
class(gdp$date)
```

```
## [1] "Date"
```

```
summary(gdp$date)  #summary can now compute mean etc
```

```
##         Min.      1st Qu.       Median         Mean      3rd Qu.         Max.
## "2003-08-15" "2006-07-23" "2009-06-30" "2009-06-30" "2012-06-07" "2015-05-15"
```

You will encounter date-variables in almost every dataset you'll ever see, and most of them do not follow the ISO standard, so let me give you a brief intro on how to handle dates and periods in R.

For this, we will need to install the **lubridate** package. Go to Packages -> Install, find lubridate, install and load it.

```
library(lubridate)
```

Lubridate offers us a number of functions, which look like: 'dmy()', 'mdy()', 'dmy_h()', 'dmy_hm()', 'dmy_hms()' etc. dmy expects the dates to be in the day-month-year order, mdy in month-day-year order etc. h means hour, m minutes, s seconds. So it covers pretty much everything:

```
dmy("02-01-2012")
```

```
## [1] "2012-01-02"
```

```
dmy("02/01 2034")
```

```
## [1] "2034-01-02"
```

It has also some nice functions for calculating durations, adding weeks or months to dates etc.

To get the current date, you can use the `today()`-function in R

```
today()
```

```
## [1] "2018-02-09"
```

YOUR TURN

Your turn:

- vignettes are one way to explain what do some packages do (they are not compulsory, so not every package has one, then – some have many). You can find the vignettes from the internet). Open the lubridate vignette (either from the internet or R help system typing `vignette("lubridate")`).
- find out how many days and weeks have you lived.
- for this you will first have to create a date variable with your birthday in it.
- then a date variable with todays date (you can use the function `today()`)
- then convert it to a time interval (there is a time intervals subsection in the vignette)
- then take a look at arithmetic with dates subsection and you should find a way to calculate how many days or weeks or seconds are there in this interval.