Summer2019_Session1

June 18, 2019

1 Data Analytics Summer School

2019 Edition (Jesse Harrison, Anni Pyysing)

2 1. Hands-on Session: Basics of Data Analysis

R is one of the most widely used programming languages for data analysis. It is available for free under the GNU General public license. In this session, we will go through a number of practical tasks to equip you with a basic grasp of how R works. This will also come in handy in the subsequent sessions. Don't worry, no prior programming experience is expected!

R interfaces: details for the interested

R code can be written using essentially any software. However, some methods are more practical than others. The most common method is to use RStudio (see rstudio.com). If you want to start using R on your own computer later on, you can start by downloading R and RStudio on your computer. Both are free. Today we will not download anything - instead, we will work with R code using Jupyter.

2.1 Let's begin!

What you are currently using is Jupyter, a web-based ("cloud") interface that can run R. While not as powerful as a full-fledged R installation, it enables us to focus on some of the basics of reading, creating and editing R code.

Jupyter notebooks consist of cells. For example, this text block is a cell that you can edit by double-clicking it (try this!). Now that you have entered the *edit mode*, you can *run* the cell by clicking the Run above, or by using `Ctrl-Enter`.

You can edit everything in your notebook (for example, delete stuff). Also, you can add cells in the "Insert" menu, run a cell in the "Cell" menu, and save and download this notebook in the "File" menu. **Saving your work** is possible by using checkpoints (go to the File-menu and click Save and Checkpoint).

2.2 Code cells and simple calculations

Jupyter notebooks have two important types of cells: **Markdown** and **Code** cells. This cell is a markdown cell and it contains text. The cell below that has the marking `In[ ]` next to it is the first code cell. R code can be executed in code cells.

You can write, for example, 1+1 in the cell below and run the calculation. Test it! You can erase the calculation, modify it, and Run it again. Also try some additional calculations using the list of

arithmetic operators below. One important observation: for decimal points, R accepts the full stop symbol (`.`) instead of a comma (`,`).

To get used to how code cells work, also try typing different calculations on separate rows. You will notice that R treats each row individually (with the calculation results appearing as a list under the code cell).

R code	Explanation	Example
`+`	addition	`1+1`
`-`	subtraction	`10-5`
`*`	multiplication	`10*2`
`/`	division	`10/2`
`^`	exponential	`10^2`
`.`	decimal point	`3.14159`

```
In [ ]: 5 + 10
        14 * 7
        10 ^ 2
```

2.3 Objects (and removing them)

You can store info in *objects*. This is useful when you want to use your object later (once created, they remain in R's memory). The standard way to assign a value to an object is `object <- value`. Here we deal mostly with numeric values, but lines of text (such as "Text") can also be used. Further to individual numbers or text labels, you can assign the results of different calculations to objects. Some examples are given below:

```
price <- 140
carPrice <- 1
piRounded <- 3.14159
name <- "Text"
cookiePrice <- 2
complexCalculation <- 5 + 5
```

To get a hang of creating objects in R, try making up a few of your own using the code cell below: one containing a single number, another containing a line of text, and one using a calculation. Note that to actually see what is inside an object, you will need to write the name of the object separately after performing the assignment. Otherwise it may appear as if nothing happens when you run the code. For example:

```
price <- 140
price
```

Sometimes you might also want to remove an object after creating it. This can be done using `rm()` (with the name of the object inserted inside the brackets). After creating the three objects, let's also try removing them!

Note: there is also a command for completely clearing up your working environment in R. This can be done as follows: `rm(list = ls())`. Use it with some caution because R doesn't come with an undo button!

```
In [ ]: obj1 <- 5
        obj2 <- "pancakes"
        obj3 <- 5 / 4

        obj1
        obj2
        obj3

        rm(obj1)
        rm(obj2)
        rm(obj3)

        # rm(obj1, obj2, obj3)
```

2.4 Calculations using objects

Once data have been assigned to objects, they are convenient to use for calculations:

```
discount <- 20
finalPrice <- price - discount
finalSale <- (price - discount) * 0.5
```

Now you know how very basic math operations can be done in R, and it is time to put these skills to use! The code cell below has already some code written into it, and you can just modify it as needed.

As an exercise, calculate what is the final price when the discount is **a)** 40, or **b)** 15% of the starting price.

```
In [ ]: discount2 <- 40
        finalPrice2 <- price - discount2
        finalPrice2

        discount3 <- 0.15 * price
        finalPrice3 <- price - discount3
        finalPrice3

        finalPrice4 <- price - (0.15 * price)
        finalPrice4
```

2.5 Vectors and functions in R

2.5.1 What vectors are

While the examples above dealt with individual numbers, this is not how R is typically used. Usually you have plenty of data (many items and many prices, for example). The way to store a sequence of numbers (or other types of data) is to use a vector. When we are dealing with numeric data, the vector is called a *numeric vector*. When dealing with text data, the vector is called a *character vector*.

We'll learn more about vectors soon, but for now let's go ahead and create a vector called *testVector*. See what comes out by running the cell!

```
In [ ]: testVector <- c(10, 30, 1, 4)
        testVector

        sort(testVector)
```

Ok, now we have created a numeric vector and sorted it. You can try to figure out how to make a character vector by yourself. If you can not figure it out, do not worry, we will learn it soon.

2.5.2 What functions are

A surprise! We have already used functions in this session, including `sort()` and `c()`. The first is a function that orders the values in a vector and the latter is a function that combines its arguments to form a vector. In the case of `testVector <- c(10,30,-20,-1)`, the arguments are 10, 30, -20, and -1. The command for removing objects, `rm()`, is also a function.

Some useful functions include convenient ways to calculate things, such as `mean()`, which calculates the mean of a numeric vector. The function call `mean(testVector)` will give you 4.75 as the output, since it treats the vector `c(10,30,-20,-1)` as a whole. Another example of a useful function is `median(testVector)`. Let's try both:

```
In [ ]: testVector <- c(10, 30, -20, -1)

        mean(testVector)
        median(testVector)

        mean(testVector) - median(testVector)
```

Some useful functions are listed next. You don't have to memorize any functions, but it is useful to know some that are available. If you are unsure how a function works, you can always open the help (see below), or you can search the internet or R documentation for many more functions.

- Most mathematical functions also work *element by element* and return multiple values `exp()`, `log()`
- Some functions treat the vector as a whole and return a single value, like the length of a vector `length()`, `sum()`, `mean()`, `median()`, `sd()`, `var()`, `min()`, `max()`
- Other functions tell several things at once `summary()`, `str()`
- It is also possible to ask for help and open R documentation. `help()`, or `?function` (e.g. `?mean`) will both work
- A default R installation can also be used to create simple graphics. `plot()`, `barplot()`

2.6 Annotating your code

Annotating your code is done by using the `#` character in R. This is especially necessary when writing longer projects. Comments are used to explain what you are doing in your code. The `#` will turn the following text into "non-code" and it will be ignored by the interpreter that turns the R-code into action.

```
In [ ]: # This is a comment
```

```
help(summary)
summary(testVector)

# See what happens if you erase the # character below
# mean(testVector)
```

2.7 Next some exercises!

Use the `help()`, search the internet, or just quess the correct commands. Writing something "wrong" does not break anything, it will just give an error message. Find the mistake, fix the mistake, and run the cell again!

Remember: if you store the value of some calculation in an object, you can read its content by simply writing its name.

Exercise 1: Learning to use external resources Find out the variance of the following numbers: 1, 4, 9.5, 10, and -2. *Hint!* There is a function for it :)

In [ ]:
```
# Exercise 1 answer below

variance <- var(c(1, 4, 9.5, 10, -2))
variance
```

Exercise 2: Finding out how a function works Learn what the function `seq()` does and what arguments it needs to work. Test it. Do not be afraid to get an error message!

In [ ]:
```
# Exercise 2 answer below

mysterySeq1 <- seq(1, 10, 1)
mysterySeq1

# needs 'from', 'to' and 'by' arguments.
# above: from 1 to 10 "by" 1 (the "by" argument specifies the interval)

mysterySeq2 <- seq(1, 10, 2) # "by" 2
mysterySeq2
```

Exercise 3: Thinking on your own Create a vector `t` containing the numbers from 0 to 10 with 0.1 interval, and plot the `sin()` of the vector `t` at the points contained in the `t` vector.

In [ ]:
```
# Exercise 3 answer below

t <- seq(0, 10, 0.1)
y <- sin(t)

plot(t, y)
plot(y, t)
```

Extra exercise for the fast ones Find out how to write a title into the plot and change the scatter plot from exercise 3 into a line plot. *Hint!* The `plot()` function can take more arguments than two. Do not worry about finishing this exercise if you run out of time.

```
In [ ]: # Answer for the extra exercise

        # for more info, see help(plot)!

        plot(t, y, type = "l", main = "a smart title")
```

2.8 Data types

2.8.1 Numeric vectors vs. character vectors

Now that you can write simple R code and search the R documentation or the Internet for information about functions, it is time to learn something new! As you might remember, there are *numeric* and *character* vectors, which are used to store numbers and text. A character vector can obviously contain numbers as well, but in text format. Character vectors can be created using quotes `" "` or single quotes `' '`.

For example, what is the difference between the following commands?

```
numVector <- c(1, 2, 3, 4)
chVector <- c("1", "2", "3", "4")
```

Try running the cell below and inspect the message.

```
In [ ]: # Demonstrating the difference between character and numeric vectors

        numVector <- c(1, 2, 3, 4)
        chVector <- c("1", "2", "3", "4")

        sum(numVector)
        sum(chVector)
```

2.8.2 Logical vectors and logical operators

In addition to numeric and character vectors, there are *logical* vectors. Logical vectors have only two possible values, `TRUE` and `FALSE`. They are answers to questions such as "Is the price above 50?" or "Is the price exactly 10?".

A logical vector can be created similarly to other vectors, for example:

```
answers <- c(TRUE, FALSE, TRUE, TRUE)
```

Logical vectors (notice that a single value is a vector with the length of one) are often created as the output of functions. For example, the function `is.numeric()` tests if its argument is numeric and returns a logical value.

```
In [ ]: # Run this cell to see what happens
        is.numeric("some character string")
        is.numeric(50385)
```

Logical vectors are not only created as function outputs, but also as the result of *logical operations* that are written in an intuitive way, very similarly to the arithmetic operators.

The *logical operators* in R are listed below with some use examples:

R code	Explanation	Example	Output
`<`	less than	`10 < 15`	`TRUE`
`>`	greater than	`3 > 3`	`FALSE`
`<=`	less or equal	`3 <= 3`	`TRUE`
`>=`	greater or equal	`5 >= 6`	`FALSE`
`==`	equal	`7 == 7`	`TRUE`
`!=`	not equal	`2 != 2`	`FALSE`

It is also possible to combine different rules by using logical operators, which are:

- and `&`
- or `|`
- not `!`

Try to play with logical operators a bit in the code cell below. After trying out what's already written inside the cells, try some examples of your own.

```
In [ ]: # Test some logical operators to see how they work
        age <- 15

        # One rule
        age < 18

        # Combining rules
        age < 18 & age > 16

        # Own examples

        oldage <- 99
        oldage <= 100

        oldage != 100
```

Now for an exercise to flex your brain with logical operators.

Exercise: Book auction Let's say that you run a small company that sells boxes of books in an auction. The amount of books in each box varies, and the price of each box is set in the auction.

The object `prices` has the prices of the boxes that were sold each day, and `booksInBoxes` has the amounts of books that were in each box.

Create your own `prices` and `booksInBoxes` objects. Using what we've learnt so far, can you use a logical operator(s) to find out if the total sales was above our weekly target, 500?

```
In [ ]: # Answer here
```

```
prices <- c(100, 49, 10, 16, 88, 2, 51)
booksInBoxes <- c(30, 1, 12, 5, 8, 1, 14)
target <- 500

totSales <- sum(prices)
totSales

totSales > target
```

2.8.3 Choosing values

Sometimes we want to choose a part of the values from a larger dataset for further analysis. We can do this using square brackets `[ ]`. Observe the output of the cell below by running it. Feel free to modify the code and test how it works.

```
In [ ]: prices[1]             # choosing the first value
        prices[2:5]           # choosing the 2nd, 3rd, 4th and 5th value
        prices[prices > 40]   # choosing values based on logical operations
        prices[c(2,4,5)]      # choosing the 2nd, 4th and 5th value
```

2.8.4 The workspace

The workspace holds the objects the user (you!) have defined during the session. For example, the object `prices` should be in your workspace. Here are some useful commands to get you started with the workspace: (further to the `rm()` function): * `getwd()` prints the current working directory location * `setwd()` can be used to set a custom working directory (you will need to specify the full path) * `ls()` lists all the objects in the workspace

Now might also be a good time to create a checkpoint if you want to save the state of your work before moving on.

```
In [ ]: # Find out what objects you have in your workspace
        ls()
```

2.8.5 Data frames - the most important data structure in R

We moved on from having objects that were single values to objects that were single vectors. However, it would be useful if the vectors that are related to each other could also be represented by the same object. Now we will proceed to storing our data in even bigger entities, **data frames**. Data frames can include our previously very separate vectors in the same object.

Some of the functions will work with entire data frames, like `summary()`, but for some functions, like `mean()`, you will need to specify which data inside the data frame you want to use in the function.

Some important functions and commands that are useful for working with data frames include: * `data.frame()` creates a data frame of the arguments that it is given * the dollar sign `$` is used to take a column from a data frame * square brackets `[ ]` can be used to take a part of the dataframe * `subset()` returns a subset of a data frame

In the following examples we will learn how to create, modify, and use data frames in R. But first, let's create a couple of vectors and combine them into one data frame for storage.

```
In [ ]: # Here we create some data to put into a data frame

        amount <- c(2, 3, 5, 1)
        book <- c("Comics", "Novels", "Manuals", "Comics")
        prices <- c(7.50, 20.00, 66.0, 500)

        # Combining the vectors into a dataframe

        bookTypes <- data.frame(amount, book, prices)
        bookTypes
```

2.8.6 Factors

There is still one data type to discuss, **factor**. Factors can be described as categories, such as `Comics`, `Novels`, and `Manuals`. You can check that the class of `bookTypes$book` is actually factor by using the function `class()`. The class changed from character to factor because the function `data.frame()` turns character vectors into factors by default. The conversion can be avoided by the option `stringsAsFactors = FALSE`. Factors are necessary in statistical modeling, where different categories are often compared.

Let's create a new data frame, but this time, we'll put everything inside it immediately. Also, notice how we have split this longer section of code onto multiple rows. We learnt before that R will treat each line of code individually and this is true when each line contains a *separate* section of code (such as assignments for separate data frames). In the current case, arranging the code on multiple rows is possible since all the commands relate to the same task (creating a data frame).

```
In [ ]: # Creating a data frame called 'bookReview'
        # (Note how the code is split into different rows for easier reading)

        bookReview <- data.frame(storage = c(4, 1, 3, 2),
                    titles = c("Good book", "Excellent book", "Bad book", "Horrible book"),
                    cost = c(12.1, 20.0, 3.33, -10 ),
                    stringsAsFactors=FALSE)
        bookReview
```

Now if we want to get only the titles, we must specify from which data frame it is found. This is done by using the $ sign. You can also try to take the column without specifying the data frame, and see what happens.

```
In [ ]: bookReview$titles              # take a column from the data frame
        titles                        # take the variable 'titles'
```

2.8.7 Creating and modifying columns

Now we will create (and modify) some new columns into the data frame using the $ sign. See what happens to our `bookReview` and try to figure out how the column `storage.value` is created!

Hint! The next exercise will be easier if you focus on this part.

```
In [ ]: # Creating some new columns
        bookReview$type <- "book"
```

```
bookReview$storage.value <- bookReview$storage * bookReview$cost
bookReview

# Modifying our new columns
bookReview$type[4] <- "trash"
bookReview
```

Exercise: Modifying the data frame We have received a shipment of new books and their information should be stored into our data frame. The shipment included two books of each type. Don't forget to update the storage value!

Luckily for us, the cost of the books stays the same, so we don't need to worry about that.

```
In [ ]: # Answer here

        bookReview # before changes

        bookReview$storage <- bookReview$storage + 2 # updating number in store
        bookReview

        bookReview$storage.value <- bookReview$cost * bookReview$storage # updating storage va
        bookReview
```

2.8.8 Choosing values from a data frame

As you might have already learned, there are many ways to achieve the same goal in R. Both `subset()` and `[ ]` can be used to choose columns and rows. "Unselecting" columns is done by the minus sign (-). Uncomment the method you want to test. Can you figure out how each method works?

```
In [ ]: # Create a column with a sequence of numbers
        bookReview$delete.column <- seq(1, 100, length = 4)
        bookReview

        # multiple methods to delete the 6th column in the data frame:

        # bookReview <- bookReview[1:4,-6]
        # bookReview <- subset(bookReview, select = -6)
        # bookReview["delete.column"] <- NULL

        bookReview
```

We can also set rules for the observations that we want to take from the data frame.

```
In [ ]: subset(bookReview, storage > 1 & cost > 5)
        subset(bookReview, type == "book")
```

Exercise: Finding observations from the data frame Find out the average cost of the books that are not horrible. *Hint*: logical operators are your friend!

```
In [ ]: goodReview <- subset(bookReview, titles != "Horrible book")
        goodReview

        mean(goodReview$cost)
```

Next, let's try to use some functions on our data frame. Run the cell below and observe the output!

```
In [ ]: summary(bookReview)  # see how this works on a data frame
        mean(bookReview)     # see how this does not work on a data frame
```

2.8.9 Error messages (and missing values)

Learning to read error messages is a good skill to have, because it will help you locate and fix errors in your code. Did you read the warning message that appeared when we tried to take the `mean()` from the entire data frame? The message says that the argument we gave the function is not in a suitable format. However, the function did still return something.

`NA` is short for "not available", which means that the answer is a missing value. In real life, datasets are full of missing values. You should remember that missing values can disturb some calculations, such as if you want to take the `mean()` from a vector that has `NA`s. Luckily, you can tell the function `mean()` to skip `NA`s in calculation by giving the function an extra attribute, `na.rm = TRUE`.

There is also another value, `NULL`, that could be confused with `NA`, but the difference between `NULL` and `NA` is a bit out of the scope of this session. Check out the R documentation if you are interested!

2.9 Getting data into R

For now we have dealt with very small data sets that we created by ourselves, but in the real world you would use data from external resources. R has many functions for data import, such as `read.csv()` to read .csv files and `read.table()` to read .txt files. Notice that your data might not be exactly in the format that the function expects! For example, if your data uses a different decimal point than the default `.`, you can include an extra argument to specify the separator. This would be written as `read.csv("filename.csv", dec = ",")`. Another alternative is to use `read.csv2()`, which uses the semicolon (`;`) as the separator.

To load a .csv file into R, you would have two options:

`read.csv("file.csv)` # when the file is in your working directory
`read.csv("pathtofile/file.csv")` # when the file is outside your working directory

R also comes with some built-in datasets that you can practice with. If you are interested, available datasets are listed by the `data()` function. The same function also loads the dataset when the name of the dataset is given as an argument. The *PlantGrowth* dataset is a good one to start with.

```
In [ ]: # An example of exploring a built-in dataset

        # first clear the workspace
```

```
rm(list = ls())

# load the PlantGrowth data set
data(PlantGrowth)

# explore the data
head(PlantGrowth)
str(PlantGrowth)
?PlantGrowth
```

2.9.1 Packages

Packages are the strength of R! You can extend R by installing new packages. There are several packages already installed with R, but installing new ones will probably be necessary if you are working with more than the very basic stuff. You only have to install a package once, but load it into the library for use every time you start a new session. Here are some useful functions to get you started!

- `installed.packages()` will list all the packages that are already installed
- `install.packages("packageName")` will install the package from CRAN
- `library()` will list all the packages that are available in the library and ready to use
- `library("packageName")` will load the package to be used during the session

The amount of packages available for R is enormous. See this article at rstudio.com to find a nice list of useful packages. All available packages at CRAN can be found at cran.r-project.org under Packages, and a useful listing of packages by topic is under Task Views.

Now you have the basic tools to move on to the next section of this Data Analytics School. If you want to go through these materials later on, go ahead and download this notebook as PDF!

2.10 Extra examples and exercises

Here are some extra examples and exercises for those who have some coding experience, have perhaps used R before, or are just very fast.

2.10.1 PlantGrowth data set

Let's continue with the *PlantGrowth* data set. This data set, while quite a small one, is still useful for our purposes. It contains information on plant yields (measured as dry weight) obtained under either control conditions or two experimental treatments.

Something we haven't yet tried (but will explore later on in more depth) is using R to visualize your data. In addition to offering many functions for data manipulation and statistics, R is also great for creating plots. Here, we can try looking at the control vs. experimental groups using one of the default functions in R called `boxplot()`. It produces quite a lot of information for just a single line of code:

```
In [ ]: boxplot(weight ~ group, data = PlantGrowth, col = "blue")
```

The box and whiskers plot shows the group medians, lower (25%) and upper (75%) quartiles and also the minimal and maximal values within each group. What conclusions do you think we could draw from the results?

One could also think about plotting the data slightly differently. For example, we might want to see all the individual data points rather than a summary. One way to do this is to type in `stripchart(weight ~ group, data = PlantGrowth)` (try it using the code cell below!). The resulting plot could be easier to interpret, however. Can you find a way to rotate it clockwise so that the groups are shown on the x axis and weight on the y axis? To find the answer, have a look at the R help file for this function.

In [ ]:
```
stripchart(weight ~ group, data = PlantGrowth)
```

```
stripchart(weight ~ group, data = PlantGrowth, vertical = TRUE)
```

2.10.2 A few words on formulas

You've probably noticed how the code in these examples is somewhat different to what we were working with before. The use of a the tilde symbol (~) in R is specific to *formulas*. The general way in which formulas in R are structured can be thought of as:

`function(y ~ x)` # first y, then x!

Another situation where you will encounter formulas is when you're doing statistical modelling with R. For example, in the future if wishing to fit a linear model to a data set, you could use the linear model function `lm()` as follows:

`lm(variable_y ~ variable_x, data = dataframe)`

As diving into the world of statistics is beyond the scope of this hands-on session, you may wish to look up the linear model function in your own time: `?lm`

2.10.3 Free exploration

Feel free to explore another built-in dataset in R, *ToothGrowth*.

In [ ]:
```
# Free exploration
```

```
?ToothGrowth
data(ToothGrowth)
head(ToothGrowth)
```