Introduction to RStudio – Further Reading – Week 1

1 Introduction

In the Workshop 1 on Monday you used RStudio for the first time and learnt some basic principles R.

This document introduces a couple of important aspects of R, for which you do not necessarily have to try out all the examples yourself. Unlike in the Workshop all the R-output is included in the examples. But first a bit of background.

1.1 A Quick History of S-Plus and R

The S Language was initially developed in the 1970's by John Chambers of Bell Labs USA, and was substantially updated in 1988 (*The New S Language*). The Commercial Software *S-Plus* was developed based on the S Language for multiple Platforms and was one of the main statistical software programs in th 1990's and early 2000's and was widely used in academic and industrial research environments where advanced statistical methods are required.

In 1995 the first Version of R was developed implementing the S Language as part of a free and open source program. Other computational Statisticians joined the R-Project, notably Bill Venables and Brian Ripley, who had already produced several books and courses on S and S-Plus.

A major benefit with the open source approach was that many statisticians could contribute their own code as *packages* which can be seamlessly integrated into the R environment. By the mid 2000's R overtook S-Plus, which today is rarely used.

Today R has become (one of) the most popular statistics programs for professional statisticians working in business or university.

2 Lists and Data Frames

2.1 Lists

Lists are useful for collecting a variety of related data types under one object. Individual items can be recalled by the `$` sign.

In the next example, the first command is split over two lines. R recognises that the first line is incomplete and responds with a '+' prompt rather than the usual '>', to indicate that it expects the command to be continued.

Example

```
> fredList <- list(name="Fred",wife="Mary",
+  num.children=3, child.ages=c(4,7,9))
> fredList
$name
[1] "Fred"

$wife
[1] "Mary"

$num.children
[1] 3

$child.ages
[1] 4 7 9
```

Note: some R functions return a list result. Probably the first that you will encounter is `lm()` for a linear Model (regression).

2.2 Data frames

The `iris` data set is an example of a data frame (or data matrix). Vectors can be put together to make flexible data structures called *data frames*. A data frame is a collection of column vectors, each of the same length. The vectors may be numeric, factor, or logical. Each particular column and row of a data frame is given a name which can be chosen by the user, or assigned a default by R.

Example using characters from Aardman Animations (Wallace and Gromit)[1]

```
> name<- c("Wallace","Gromit","Shaun","Feathers_McGraw")
> type <- factor(c("human","human","animal","animal"))
> x<-c(72,8,5,NA)
> aardman <- data.frame(name, type, age=x)
> aardman
             name   type age
1         Wallace  human  72
2          Gromit  human   8
3           Shaun animal   5
4 Feathers_McGraw animal  NA
> aardman$type
[1] human  human  animal animal
Levels: animal human
```

Notice the value `NA` for Feathers McGraw's age. This is a special value used by R to denote a missing data value (NA = 'not available').

Data frames allow data to be accessed in a very flexible way.

Example continued

```
> aardman$age
[1] 72  8  5 NA
> aardman$name[type=="animal"]
[1] Shaun           Feathers_McGraw
Levels: Feathers_McGraw Gromit Shaun Wallace
```

What is the mean age of the characters?

```
> mean(aardman$age)
[1] NA
```

Since Feathers McGraw's age is unknown, we don't know the mean age. However, we can force R to calculate the mean of the available values:

```
> mean(aardman$age,na.rm=TRUE)
[1] 28.33333
```

This translates as 'compute the mean of the values, removing the `NA`s'. Be careful using `na.rm=TRUE` it's easy to get biassed results.

[1] Feathers McGraw is the name of the criminal penguin in "The Wrong Trousers"

3 Use of Files

3.1 Current working directory

When you ask R to access information from a file, by default it assumes that the file is located in the *working directory*[2]). RStudio displays the current working directory next to the word **console** at the top of the console window. There is also an R command to access the current working directory.

```
> getwd()
```

It is helpful to change your working directory to where you want to access data files and save script files graphics etc. The easiest way to do this is via the menus *Session > Set Working Directory > Choose Directory*

However if you want to run your source code again at a later date, then it is better to do this using an R-Command

```
> setwd("H:/StatComp/")
```

The syntax here is somewhat annoying. For unimportant historical reasons R uses `/` and windows uses `\` to seperate the directory names in the path.

3.2 Reading data from files

Understanding data formats, data manipulation and importing data is an essential part of data science. You will learn much more on this topic in other courses and later on in this course. Today we cover just the basics so that we are not limited to data files which come with the R installation.

Which command you use to read in data depends on which format the data is in. For text files the usual functions are `read.table()` and `read.csv()`. Other possibilities are `read.csv2()`, `read.fwf()` or `scan()`. There are also functions to read in data in a format that was saved by another statistical software such as SPSS or SAS.

A good summary of reading in data for different formats can be found at R Data Import Tutorial.

Note that the external “text” files must be in ASCII format. Today most numerical data sets are created electronically rather than typed in with a keyboard. If you do need to create a dataset by hand then I suggest using Excel and saving it as a comma separated version `.csv`. Word is not recommended as it is too easy for the layout to go wrong.

[2] directory=folder

Example

Suppose that your current working directory contains a file named `somedata`, which contains the following:

```
 2  100.0
 3   44.5
```

In R,

```
> x <- read.table("somedata",header=FALSE)
> x
  V1    V2
1  2 100.0
2  3  44.5
```

3.3 Reading commands from a file

Typically, many commands are required for an analysis. By storing them in a script file, the analysis can be repeated or modified later on.

While you're working out how to do the analysis, type the commands into the RStudio script file and run each line individually. To execute lots of commands in one go by highlighting the code you want to run and typing Ctrl+Enter or clicking on the Run button at the top of the source file.

Another way is to source the whole file. If you click on Source the script file will be saved and then run using the `source()` command. You can use also use `source(...)` directly at the command prompt or in a script file to run another script file.

For example, suppose the *current working directory* contains a file `somecommands.r`, whose contents are as follows:

```
dat <- c(3,3,3)
print(dat)
```

(with a carriage return after the last line in the file). You can run these commands by typing

```
> source("somecommands.r")
[1] 3 3 3
```

or

```
> source("somecommands.r",echo=TRUE)
> dat <- c(3,3,3)
> print(dat)
[1] 3 3 3
```

3.3.1 Commenting your programs

To make your programs readable by a human, it is helpful to insert lines that explain what's going on. These are called *comments* in programming jargon. To include comments in an R program, use the # character: R will ignore anything on a line after encountering this character. Here is a simple R program:

```
#
# This is a very simple program that illustrates the value of
# comments. It starts by asking the user to input their name ...
#
username <- readline(prompt="Please input your name: ")
#
# ... and then it says hello to them, personally.
#
cat(paste("Hello, ",username,"!\n",sep=""))
```

Even though you don't know all the commands, the program is (hopefully) perfectly intelligible because the comments explain what's going on.

Get into the habit of commenting your programs clearly. This applies not just to R but to all types of coding. If you have assessments where you need to write code (includung the tests in this course) you can only gain high marks if the lecturer can understand what you've written!

3.4 Saving output to a file

When you run commands in RStudio, results are by default written to the screen. The `sink()` command is used to direct output to a file instead of the screen: this file can then be printed and examined. If you have stored all of your R commands in a file called `somecommands.r`, this enables you to print a 'clean' copy of the results:

```
> sink("somecommands_output.txt")
> source("somecommands.r",echo=TRUE)
> sink()
```

Now you will not see anything on screen when you run `source("somecommands.r")`. The results are written straight to `somecommands.res`. The final `sink()` command tells R to stop writing to this file and return output to the screen. N.B. RStudio does not at the moment allow you to save the console window file so if you want to save the output `sink()` is the best way to do this.

3.5 Current Workspace

All assigned variables (or any other R *objects*) are automatically stored by the computer in memory, in your R *workspace* also known as *Global Environment*, until you finish your R session. When you finish, R gives you the option to save your workspace for future use.

Experienced R users recommended that you you do not save your workspace for two reasons. If you always save it, all objects will remain in the workspace until overwritten or explicitly deleted by the command `rm()` (for *remove*). This means that, after a semester or so, your R files can

take up a lot of memory, and can slow down R, and you end up with lots of objects, the purpose of which you have long forgotten.

A rarer but more serious problem of reloading your past objects is that your code could be relying on an object that was defined 6 months ago and not defined in your current source code. This means that when you do eventually run the code in a clean environment or if you pass the code onto another user then it will not do what you expect it to.

In the Environment Window, top right, you can see all the objects in the current "workspace". To output your workspace objects in the R console, type `objects()`.

Example continued

```
> x <- 8
> x
[1] 8
> y <- 3.1415
> objects()
[1] "x" "y"
> rm(x)
[1] "y"
```

4 Using Packages

There are hundreds of extensions to R called packages, which are publicly available. These packages have been contributed by R users, who have developed R functions for specific purposes.

An example is the package `survival`. This includes many functions used in the statistical field called "survival analysis" and was contributed by Terry Therneau, a statistician who has contributed many extensions to R.

You will probably need to use many R extensions in this degree Programme, but they are easy to install and use.

4.1 Installing a Package

The R code to install a package is

```
> install.packages("packagename")
```

This asks you to choose a mirror site (somewhere local i.e. in Germany is best). The installation might take a minute or so and install some other "dependency packages" but that should be all you need to do.

In some instances you might have problems with administrative rights not being able to write files to certain directories. If you are at home and on a windows PC and have this problem quit R, right click on the RStudio Icon and run as administrator. If you are on a Linux OS login as

super-user and run RStudio. When installing a package in the Beuth Maths-Lab as part of the Workshop, your lecturer will give you the instructions specific to that network.

You will only need to install specific package once on each computer.

To use the functions inside a package run the command
`> library(packagename)`
or `> require(packagename)`
Quotes (") around `packagename` are optional. The difference between `library()` and require() is not important at this stage.