Introduction to R

Andrew McAdam

2012-09-30

Contents

Welcome 1

Opening and Navigating RStudio 2
- Console 2
- R as a Calculator 2
- Storing Information 3
- Adding Comments to R Code 4

Working Directory 4
- Setting the Working Directory 4

Vectors 5

Creating random numbers 6

Functions 7

Workspace 8
- Saving the Worskpace 9
- Good Coding Practices 10

Writing Scripts 10

Getting Help 11

Reference to R 11

Packages 12
- Dependencies 13

Welcome

Welcome to R! This document will provide you with an introduction to R. If you follow along and perform the various tasks as you move through the document you will get a decent introduction to R. Feel free to fiddle around as you go. Making mistakes is a great way to learn!!

In this document you will find R code as well as the output from R as it will appear on your screen. This document will provide in a lot of cases both the input and output of the R session. I recommend that as you work your way through the document, you type the code into R, instead of using copy-paste. This will allow you to get used to typing R code, and get used to trouble-shooting any error messages you will get.

One of the free programs that has been written to help you use R is called RStudio (http://rstudio.org). RStudio is just a wrapper (convenient and prettier interface) for R. We will use it in this course but know that this is not a different statistical program. It is just a way of interacting with the statistical program R.

I will provide both a pdf copy of this document, but it has been created using RMarkdown in RStudio. This program allows you to create PDFs or HTML like this of your work in R. RStudio is a very useful interface for R and producing PDF documents like this will allow you to keep very careful notes about what you have done and be able to communicate these (to your supervisor, perhaps) so that it is perfectly clear what you have done. As a supervisor, I require all of my students to leave me with an archive folder of all of their data and R code. Having a document like this would then allow me to see exactly how they performed their analyses and to recreate their analyses. This sort of paper trail is essential for the reproducability of scientific findings and will save a lot of time in the long-run if you get used to working in this format.

Opening and Navigating RStudio

Console

Open RStudio. It is in the Applications folder on the computer. When RStudio first opens there will be a window on the left called the 'Console' and one on the right called 'Environment' or 'History" You can ignore these ones on the right for now.

The Console represents your interface with R. This is a Question and Answer type interface where you submit a command and often receive some sort of output from R. The ">" symbol indicates that R is ready to receive an input from you.

R as a Calculator

One of the simplest thing that R can be used for is simple calculations. For example we can add two numbers together by typing

```
5+18
```

```
## [1] 23
```

Note first that the output from R in this document is preceded by "##". You will not see this in your console when you type in 5+18. Second, the output of this command in the console will be preceded by the number 1 in square brackets. This is just an index for keeping track where the answer was put. It actually means that it is the first value in a vector. We will come back to that later.

We can also do other types of math within R. . .

```
10*230
```

```
## [1] 2300
```

```
(10+10)/5+6
```

```
## [1] 10
```

```
#versus

(10+10)/(5+6)
```

```
## [1] 1.818182
```

```
sqrt(81)
```

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

Note that in this document my code (what I have typed into R) is in the grey box and the output from R is right below this.

```
log(10)
```

```
## [1] 2.302585
```

```
#Note that this is different from

log10(10)
```

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

```
exp(1)
```

```
## [1] 2.718282
```

Note that the log and exp functions refer to base e and not base 10.

Storing Information

In addition to receiving the output directly, the result of a calculation can be stored as a new variable.

```
x<-5+18

#or

x=5+18
```

Note that it looks like nothing happened here, but this was because we didn't ask R to respond. All we asked it to do was create a new variable called x and make it equal to 5+18. If we want to know what x is we just type in 'x' and R will tell us what it is.

```
x
```

```
## [1] 23
```

We can call a variable pretty much whatever we want…

```
answer<-5+18

 answer
```

```
## [1] 23
```

```
 y=10*4

y
```

```
## [1] 40
```

We can also use our variables within other calculations or to define additional vartiables.

```
 z=x*y

z
```

```
## [1] 920
```

Note that "=" or "<-" assigns a particular value to a variable. We can also ask R a question but we need slightly different notation for this. Suppose we were interested in knowing whether our variable z was equal to x. To do this we would use…

```r
z==x
```

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

The "==" asks the question "is z equal to x?". Here you can see that the answer is false.

NOTE THAT THIS IS VERY DIFFERENT FROM:

```r
z=x

#or z<-x, where here we have ASSIGNED z to be equal to x

z
```

```
## [1] 23
```

```r
x
```

```
## [1] 23
```

```r
z==x
```

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

Adding Comments to R Code

You can see from some of my examples above that I sometimes added some notes or comments to my code. Whenever I did this I always preceeded my comments with the # character. This tells R that what follows the # is a comment and should be ignored. Everything you write after the # will be ignored by R until you hit enter to begin another line of code. At this point R will pay attention again.

```r
#I want to add comments here.  This is OK.
```

If you forget to enter the # character before a comment then R will get very confused and give you an error because it doesn't understand what you mean.

Working Directory

The working directory is the place on your computer where you are currently working. This is place where you R will ask for files that you request and where R will write output. You can see this in the 'Files' tab in the bottom right corner of RStudio, but you can also ask R for this information

```r
getwd()
```

```
## [1] "/Users/andrewmcadam/Documents/McAdam/Research/R datafiles/EBIO 4410"
```

You can see that I keep my R files organized within a series of folders for each project that I am working on.

If you use R projects in R Studio, which I would suggest, then the project will make sure are in the correct working directory for that project.

Setting the Working Directory

You can change your working directory in a few ways. First, you can use the menus, but this is where Mac/Windows/Unix interfaces might differ. On my Mac version of RStudio there is a menu "Session" and within this there is the option to "Set Working Directory". This will open a browser where you can select your working directory. Alternatively, you can use the command interface to do this directly.

```
setwd("/Users/andrewmcadam/Documents/McAdam/Research/R datafiles/EBIO 4410")
```

Note that R is case-sensitive so that Setwd and setwd are not equivalent!

Vectors

Instead of creating a variable we can also create vectors.

```
variable <- c(12, 33, 45, 101, 65, 30, 55, 99, 70, 84)
```

The c is short for concatenate

```
variable
```

```
## [1]  12  33  45 101  65  30  55  99  70  84
```

Note again that the [1] just helps us keep track of the fact that the value 12 is the first element in the vector.

```
one.to.ten <- 1:10

one.to.100 <- 1:100

one.to.100
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```
#also

one.to.ten <- seq(1,10)

one.to.ten
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
#or

one.to.ten <- seq(1,10, 2)

one.to.ten
```

```
## [1] 1 3 5 7 9
```

Note that the vector now wraps around more than one line and so the [] values at the left help us to keep track of where we are in the vector.

We can refer to a specific element in a vector by specifying its location in square brackets after the object name. For example the 3rd element in the vector called one.to.ten is simply…

```
one.to.ten <- seq(1,10)
one.to.ten
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
one.to.ten[3]
```

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

Just like variables, you can also do math with vectors

```
one.to.100*5
```

```
##   [1]   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
##  [19]  95 100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175 180
##  [37] 185 190 195 200 205 210 215 220 225 230 235 240 245 250 255 260 265 270
##  [55] 275 280 285 290 295 300 305 310 315 320 325 330 335 340 345 350 355 360
##  [73] 365 370 375 380 385 390 395 400 405 410 415 420 425 430 435 440 445 450
##  [91] 455 460 465 470 475 480 485 490 495 500
```

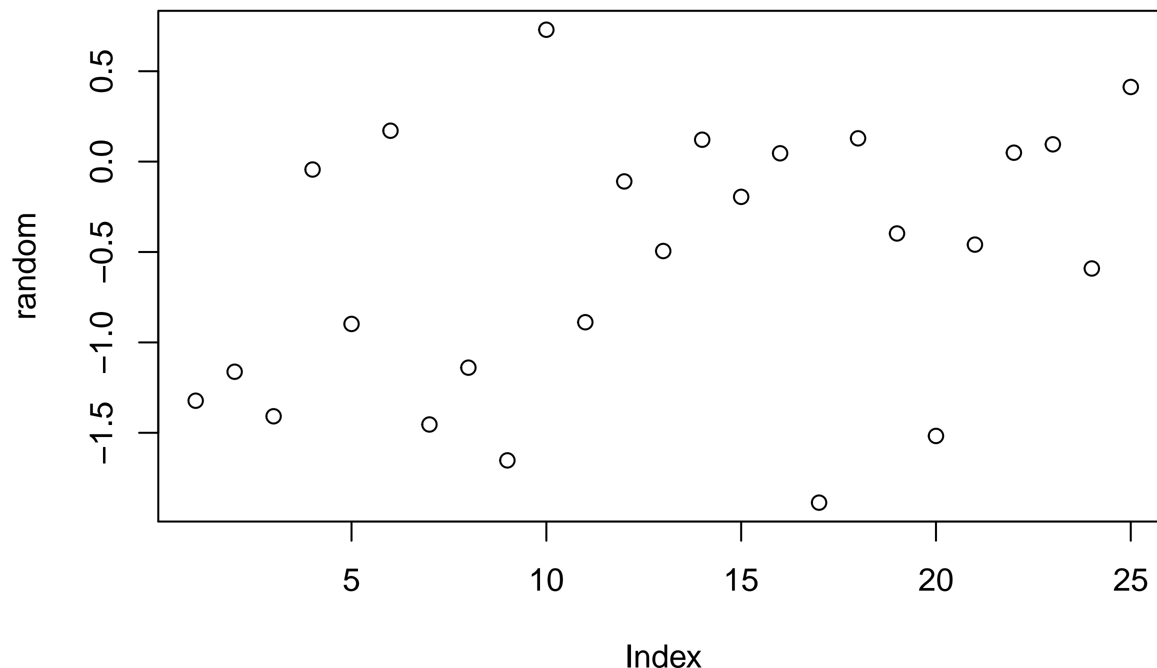
Creating random numbers

Because R is a statistical program you can also easily generate random numbers from a known distribution

```
random<-rnorm(25, mean=0, sd=1)
random
```

```
##  [1] -1.32249524 -1.16221756 -1.40852689 -0.04388863 -0.89793187  0.17083007
##  [7] -1.45401814 -1.13965221 -1.65256320  0.72925920 -0.88835472 -0.10962413
## [13] -0.49462495  0.12100252 -0.19499594  0.04601081 -1.88621182  0.12879877
## [19] -0.39745975 -1.51732895 -0.45921136  0.04930536  0.09561455 -0.59109446
## [25]  0.41270009
```

```
plot(random)
```



The plot function above generates a new graphical interface. R has a great deal of flexibility and power in how it generates graphics. You can do fairly simple things as we did above, but you can also get quite detailed in your graphics. I will spend more time talking about graphics later.

We can also check that this vector has 25 elements, a mean of zero and a sd of one.

```
length (random)
```

```
## [1] 25
```

```r
sum(random)/length(random)
```

```
## [1] -0.5546671
```

```r
mean.random<-sum(random)/length(random)
```

```r
sqrt(sum((random-mean.random)^2)/(length(random)-1))
```

```
## [1] 0.7298084
```

or simply

```r
mean(random)
```

```
## [1] -0.5546671
```

```r
sd(random)
```

```
## [1] 0.7298084
```

Functions

We have already encountered a number of R functions that come pre-loaded in R as part of the core package (e.g., log(), mean(), sqrt(), etc.). In addition to creating objects, variables and vectors, we can also write functions in R that will help to simplify life later. Remember that we already looked at:

```r
log(exp(10))
```

```
## [1] 10
```

As we saw there is also a log10(x) function, but no exp10(x).

If we type in… exp10(10)

We get… Error: could not find function "exp10"

So no function called exp10(). So let's write one!!

```r
exp10<-function(x){
#This function accepts a value, x, as input and returns the value of 10 raised
#to the power of x
10^x
}
```

We specify that this is a function using the function command and that it will receive only one entry as a variable, called x. All of the function is then contained within the braces. Note that I wrote a comment within the function to explain what the function is doing and asked R to ignore this text by preceeding the comment with the # character.

We can now recall our function

```r
exp10
```

```
## function(x){
## #This function accepts a value, x, as input and returns the value of 10 raised
## #to the power of x
## 10^x
## }
```

This does not run the function but instead just shows what is written within the function. It will only run if we specify the variable x in parentheses. X is the number or object that we want to be considered by the function.

As a test of our function…

```
exp10(3)
```

```
## [1] 1000
```

This function only contains one argument but we can easily make changes so that it contains several

```
 exp10<-function (x, y){
#This function is bogus but calculates 10 raised to the power of x but adds some other value, y, to it}
 10^x+y
 }

 exp10(3, 4)
```

```
## [1] 1004
```

Note that if I fail to include a necessary parameter then I will get an error

If I type… exp10(3)

I will get the error… Error in 10^x + y : 'y' is missing

This isn't a very useful function so I am going to change it back

```
 exp10 <- function(x){
#This function accepts a value, x, as input and returns the value of 10 raised to the power of x
 x <- 10^x
x
 }
```

Note that we can easily retrieve prior commands in R by simply hitting the up arrow on your keyboard. If you hit the up arrow on your keyboard and change your mind just hit esc. This will abandon your command without running it and return you to the input prompt.

The other important thing to know about functions is that they are like Las Vegas. What happens in the function stays in the function. That is, we have defined a variable x within the function. This variable is used for the purpose of the function but does not affect any variable that we might have in our workspace called x

```
x <- 1
 temp <- exp10(3)
x
```

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

```
temp
```

```
## [1] 1000
```

Remember that we defined x in the exp10 function and so you might have thought that by running exp10(3) that we might have defined x to be 3. This did not happen because the definition of x in the function remains in the function and does not apply to the workspace object x.

Workspace

R stores variables, datafiles, functions, vectors, etc in what is called the Workspace. This contains all of the items that you can access directly within your R session. Note that these are not actual files in your working

directory, but are instead objects held with the R program. All of the items in your workspace are listed in the top-right panel in RStudio under 'Environment'. You can list all of the objects in your workspace using:

```
ls()
```

```
##  [1] "answer"      "exp10"       "mean.random" "one.to.100"  "one.to.ten"
##  [6] "random"      "temp"        "variable"    "x"           "y"
## [11] "z"
```

Let's say I didn't want to keep the variable vector around. I can delete it from the workspace using

```
rm(variable)
```

The object named variable is now reased from the workspace

If I type in… variable

I will get… Error: object 'variable' not found

We can check using…

```
ls()
```

```
##  [1] "answer"      "exp10"       "mean.random" "one.to.100"  "one.to.ten"
##  [6] "random"      "temp"        "x"           "y"           "z"
```

I can delete more than one object using

```
rm(answer, one.to.100, random)

ls()
```

```
## [1] "exp10"       "mean.random" "one.to.ten"  "temp"        "x"
## [6] "y"           "z"
```

Saving the Worskpace

It is a good idea to save your workspace so that variables, etc can be used in a later session

```
save.image()
```

This will save your workspace as your default workspace and it will be saved as a single file (an actual file) in your working directory. You can also save it to a particular file name

```
save.image("first workspace.RData")
```

If you are working in R and you want to load a particular workspace then you can load it using

```
load(".RData")

# Or

load("first workspace.RData")
```

Before quitting R you will be asked whether you want to save your workspace. This refers to the default workspace. You may or may not want to do this because it will over-write your default workspace file that currently exists. This is why it is always wise to save valuable workspaces as specific file names and not simply the default.

Good Coding Practices

It is always a good idea to save all of your data import, manipulation and analysis procedures as code rather than entering each line separately. You can then empty the entire workspace and recreate it by running your code from scratch whenever you want. If you can do this then this means that your code is reproducible.

Writing Scripts

There are three ways of interacting with R:

- Interactive mode. This is how we have used it so far. You type in one line of code and R receives it and perhaps gives you a response. The advantage is that you can check how things work on the fly. The disadvantage is that you do not have a record of what you have done, explicitly. You can save this by saving the console, by using the menus. In that case, you save the console as a .txt file, that you can open for instance in Word. Convert the font into Courrier or Courrier New to make sure that the alignment is preserved.
- Item script editor mode. In this case, you write all your code in a Script file. You can then copy and paste the code into the console or if you open the Script in RStudio you can run the Script by placing the cursor on the line of code you want to run and hitting 'command-enter'. You can also run an entire script using specific commends (see later). I find Scripts helpful for saving batches of code that I run often. For example, all of your code to import yoru data files and manipulate them prior to analysis.
- R Markdown. I have written this document in R Markdown and I ask all of my students to provide me with summaries/reports of their findings using R Markdown. It is a very handy way to mix a lot of text or commentary with some R code. When you create a new R Markdown file in RStudio it does a great job of providing you with a template, which is actually a very easy way to learn how to created and use R Markdown files. You can use these to create PDFs of your work as well as HTML, Word or even slides!

To create a new source script using RStudio select File->New->R Script. In R (Mac interface) select File->New Document. Once you have created this new R script save it as "Day One Script.R" (File->Save As in RStudio). Then type a simple source bit of code…

#This sample script is just to show how a source script works x<-25+25 print(x) #The print command simply tells R to output the value of "x"

Save your sample script. Then run it in R using…

```
source ("Day One Script.R", echo=T)
```

```
##
## > x <- 25 + 25
##
## > print(x)
## [1] 50
```

Note that indicating that echo=TRUE (or simply T) tells R that we want to see what is happening in the source script (input and output). Also, we would have needed to exactly specify the path name if we were calling a script that was not already in our working folder (e.g. source("/Users/xxx/folder/subfolder/Day One Script.R", echo=T). Try running this same script without the echo.

```
source ("Day One Script.R", echo=F)
```

```
## [1] 50
```

Note that we still get the output because of the 'print' command but we don't see the inputs that were a part of the script.

Getting Help

There are a number of places where you can get help with R directly from the console.

You can type… ?lm

This brings up a description of the function "lm".

You can also search for help on a topic based on a word that might not be the name of the function, but is instead a topic referred to or contained in a function. For this you can use

help.search("lm")

or

??lm

This brings up all references to the lm function in packages and commands in R. We will talk about packages later.

Perhaps the best way to get help with R though is to simply Google the function or topic along with "R". There is a wealth of information out there!!!

Reference to R

You will often need to make reference to this statistical package (i.e. in your final project, thesis and publications). There is no manual to directly reference. Instead R provides a reference for you to use. In case you ever forget how to find it look at the reminded that is given to you above…''citation()"

```
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2020). R: A language and environment for statistical
##   computing. R Foundation for Statistical Computing, Vienna, Austria.
##   URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2020},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```

Packages

R is just one big collection of packages. Some of these are so essential to the functioning of R that they are automatically loaded when R boots up. Others have to be loaded when you need them. Still others don't come immediately downloaded with R. You will, therefore, need to download these packages before loading them

You can see a list of all of the available packages on the CRAN site

****** Packages need to be installed only once, but THEY MUST BE LOADED EACH TIME YOU OPEN R. ********

For example if I type… ISwR

I will get… Error: object 'ISwR' not found

ISwR is a package of datasets that was put together by Dalgaard to go along with a useful R book (http://www.springer.com/statistics/computational+statistics/book/978-0-387-79053-4) that I would recommend. We can load this package in the menus or with the library() command.

```
install.packages("ISwR")

# then

library(ISwR)
```

Note that ISwR hasn't added anything to my workspace, but functions and datasets within the package can now be accessed from the console. Also note that we installed ISwR first. This only needs to be done once. We then loaded the package using the 'library' command. This needs to be done each time we start-up R.

We can load a datafile called "ashina" from ISwR

```
data(ashina)

ashina
```

The data file ashina is now part of the workspace and can be modified or used for analyses just like any other object.

```
ls()
```

```
## [1] "exp10"       "mean.random" "one.to.ten"  "temp"        "x"
## [6] "y"           "z"
```

You can (and should!) also cite packages as well as R, if you use them.

```
citation("ISwR")
```

```
##
## To cite package 'ISwR' in publications use:
##
##   Peter Dalgaard (2020). ISwR: Introductory Statistics with R. R
##   package version 2.0-8. https://CRAN.R-project.org/package=ISwR
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {ISwR: Introductory Statistics with R},
##     author = {Peter Dalgaard},
##     year = {2020},
```

```
##     note = {R package version 2.0-8},
##     url = {https://CRAN.R-project.org/package=ISwR},
##   }
##
## ATTENTION: This citation information has been auto-generated from the
## package DESCRIPTION file and may need manual editing, see
## 'help("citation")'.
```

Dependencies

R functions often refer to functions from other packages. This is extremely helpful because it means that not everyone needs to reinvent the wheel they just need to refer to someone else's wheel. The problem is that if the ISwR package refers to a function in some other hypothetical package 'Arrg' then I will get an error when I try and run the function that depends on Arrg if I have not yet installed the Arrg package. This is called a dependency. One solution to this is to sequentially keep installing the needed packages, but for some complex packages there can be many dependencies, so this can take quite a while. Only recently (!) did I realize that you could simply install a new package using

```
install.packages("ISwR", dependencies=T)
```

which will isntall ISwR as well as all of the other packages that are needed to run the functions in ISwR. This can be a lot of packages, but this can save a lot of time.

If I were smart I would write a script that simply told R to install all of the packages that I typically require. This is because each time you update R to a new version you need to reinstall all the packages. I haven't done this so instead I install them all one at a time when I get error messages telling me that they haven't been installed yet!

Before we finish for the day we should save our workspace again

```
save.image("first workspace.RData")
```