Getting started with R

Jasleen Grewal

23 December, 2020

Contents

Getting Started

Please make sure you have read the "Seminar 0a - Getting Started.docx" guide before reading this document.

If you are already familiar with R markdown, you can skip to Section: Getting around in RStudio

First, let us orient ourselves with what we are seeing. This is normal written text. Normally, when we want to comment our code, we have to append a '#' in front of a new line. In an **R markdown (RMD)** document, we can write text as is, and save and execute code in 'chunks' within the same document.

An **R script**, on the other hand, only contains code and comments. If you are in R Studio, you can initialize a new R script by going File > New File > Rscript.

For the remainder of this tutorial, we will be working from within R markdown. If you want to **run just the code in this document**, you can either:
- Open the .Rmd document in RStudio, and 'run' each chunk.
- Copy over the code you see in this document, into the RStudio console.
- Copy over the code you see in this document, into an Rscript file, and run it from there.

What's going on in this RMD?

If you have opened the .Rmd document in RStudio, you will notice a few things initially.
First off, there is a little section at the top flanked by '—'. This is the document header and tells 'knitr', the package we use to generate readable versions from the .Rmd, what the default title, author, format etc. are.

```
---
title: "Getting started with R"
author: "Jasleen Grewal"
date: "`r format(Sys.time(), '%d %B, %Y')`"
output:
  pdf_document: #The following indented defaults apply to a pdf version of this document
    toc: true #Do we want a table of contents?
    toc_depth: 3 #What 'depth' of subheadings do we want to display the TOC until?
  html_document:
    toc: true
    toc_depth: 5
---
```

You will notice there is no text formatting bar. You can define various levels of headings in this document, simply by following markdown guidelines . You should familiarize yourself with this guide, as it will be quite handy when you write your reports and assignments for this class (and other projects in real life).

How do I write code in here?

R code is written within a 3-tick boundary, as shown in this **code chunk**:

```
# This is an R-code chunk
# In here, all the text I write has to be preceded by '#'
# But the code doesn't!
print("hello world")
```

There are several flags that can accompany this **code chunk**. *eval* and *echo* are 2 flags that are immediately useful to us. 1. By setting the *eval* flag to FALSE, we can tell R markdown to skip executing the code in a chunk. 2. By setting the *echo* flag to FALSE, we can tell R markdown to skip printing the actual code in a chunk.

In the chunk above, we had set the eval flag to FALSE, and the echo flag to TRUE.

1. Setting eval to TRUE, and echo to TRUE

```
print("hello world")
```

```
## [1] "hello world"
```

2. Setting eval to TRUE, and echo to FALSE

```
## [1] "hello world"
```

As you see, setting eval to TRUE returns the output from the command, appended by a '##' in the front. If you run the same command in your RStudio **Console**, you will see something similar to entering that chunk in the console.

What will be the expected output when we set eval to FALSE, and echo to FALSE?
Try this in your R markdown. You can press 'Knit' after you have written your code chunk, to see the difference it makes in the output pdf.

Loading Packages

R lets us do many types of statistical analyses. These methods are implemented in *packages* or *libraries*. A lot of packages come pre-installed in RStudio, and we can see a list of these by running the command, **library()**.

```
library()
```

We can also install any additional libraries we may need. For example, the 'geometry' package is not installed by default in RStudio. Thus, when we try to load it into the current environment, we might get this error:

```
library("geometry")
```

*Error in library("geometry") : there is no package called b

In this case, we shall go ahead and install the required package, and then try to import the library again.

```
install.packages("geometry",repos = "http://cran.us.r-project.org")
```

```
## Installing package into 'C:/Users/mtello/Documents/R/win-library/3.6'
## (as 'lib' is unspecified)
```

```
## package 'geometry' successfully unpacked and MD5 sums checked
##
## The downloaded binary packages are in
##  C:\Users\mtello\AppData\Local\Temp\RtmpuANkUR\downloaded_packages
```

```
library(geometry)
```

Use the **HELP** function to learn more about a package or function. You can access Help from the commandline by adding '?' as a prefix to your query, like this:

```
?setwd
```

```
## starting httpd help server ... done
```

```
?geometry
```

```
## No documentation for 'geometry' in specified packages and libraries:
## you could try '??geometry'
```

If there is no exact match for your query, R will return an error like the one above. If there is a match, it will open up a link to it in the 'Help' view on the bottom right corner in your RStudio window.

Getting around in RStudio

Now that you are a bit familiar with how to run chunks of code within an R markdown document, let us play around with some features in R.

Understanding paths

We can define the path to a file relative to where we currently are (**relative path**) or based on the path relative to the base directory "/", called the **absolute path**. Directories in Windows follow a different convention than in unix-based operating systems (such as OS X and Linux). Here the base directory is typically *C:/*. However unix path variables will typically correspond to those used in windows (i.e. "~" corresponds to your home directory and "/" corresponds to your base directory).

For example, If our datafile.txt is in */home/user1/data/datafile.txt*, and our script is at */home/user1/scripts/script1.Rmd*, we can define the path for our data file as either:

An ABSOLUTE PATH
dat_path="/home/user1/data/datafile.txt"

or **A RELATIVE PATH**
dat_path="../data/datafile.txt"

The nice thing about defining an absolute path is that now, even if you were to move your script somewhere else, say, "/home/user1/script1.Rmd", your data file will still be read in from "/home/user1/data/".

Note: Notice how I have the backward-slash before the quotes in the paths above. This is to prevent R Markdown from render the double quotes at "", and to instead render them as "". This is important since, if you copy and paste the double-quotes address from the rendered pdf, you will get an error like this:

```
> dat_path=b
Error: unexpected input in "dat_path=??"
> dat_path="../data/datafile.txt"
```

Directory structures

Before we read/write anything, we must know **where we are**. The **getwd()** function can give us this information.

```
getwd()
```

You can also change the current working directory using **setwd()**. You can play with this in your own time.

Methods and datatypes

Vectors and Lists

We can run some basic commands to get a sampling of the various functions available in R.

```
log(64, base=2)
```

```
## [1] 6
```

```
c("Hi","this","is","a","vector","with",8,"elements")
```

```
## [1] "Hi"       "this"     "is"       "a"        "vector"   "with"     "8"
## [8] "elements"
```

```
length(c("Hi","this","is","a","vector","with",8,"elements"))
```

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

Notice how we have made a **vector** of length 8 in this example. In R, any entity you define based on its value is stored by default as a **vector**.

A vector has a specific class associated with it. These are the 5 different 'classes' in R:

- Logical (boolean): TRUE or FALSE
- Integers: Exactly what you think
- Numeric: These can be integers or floating point numbers, and may hence be confusing.
  - 2, 2.0, pi are all numeric. 2 can be treated as an integer by writing it as `as.integer(2)` in your code. + calling `as.integer()` on a number that can't be represented as a 32 or 64-bit integer results in `NA`, including negative integers that are too low or positive integers that are too large
- Complex: These are numbers with an imaginary component.
- Character: These are strings, and can be any combination of alpha-numeric characters. 2 can be treated as a character by writing it as `as.character(2)` in your code. The elements of this class generally have double-quotes around them when you print them.

Here is an example of a vector of length 2. We will start by having it as a numeric vector with 2 elements, 1 and 5.2

```
my_vector <- c(1,5.2)
print(my_vector)
```

```
## [1] 1.0 5.2
```

```
typeof(my_vector)
```

```
## [1] "double"
```

Now let us change the elements of this vector to characters, and see what changes:

```
print(as.character(my_vector))
```

```
## [1] "1"   "5.2"
```

And as integers. . . :

```
print(as.integer(my_vector))
```

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

What's different in these inputs? You can learn more about the various data types and structures in R by **following this link**.

Vectors can be of variable length, and so can lists. However, in a vector, all the elements must be of the **same class**. This is not necessary in a list. Consider the following example:

```
vector_random = c(FALSE, 1,2,4,7.34, TRUE)
print(vector_random)
```

```
## [1] 0.00 1.00 2.00 4.00 7.34 1.00
```

PAUSE AND THINK
What's happened here?!

Vectors implicitly coerce all their elements based on the following hierarchy: logical > integer > numeric > complex > characters. In the example above,

- not all elements could be converted to TRUE/FALSE (notice how TRUE=1.00, and FALSE=0.00)
- not all elements could be converted to integers, without losing some information (ex. 7.34)
- all elements could be converted to numeric with floating point though (notice how TRUE is 1.00, and FALSE is 0.00, and 1 becomes 1.00)

This is problematic if you want to keep multiple types of elements in the same object. **Enter lists**, which can be defined as a collection of elements without any restriction being placed on the class of the elements or the length of each element.:

```
list_random =  list(FALSE, 1,2,4,7.34, TRUE)
print(list_random)
```

```
## [[1]]
## [1] FALSE
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 7.34
##
## [[6]]
## [1] TRUE
```

Here, all the elements remain the way we had defined them. However, lists print out a new line for each element. Elements are indexed by double brackets, and single brackets denote a new list. This lets us define 'nested lists' (lists inside lists inside lists...) in R.

Lists are a very flexible data object in R. If you are not careful about how you assign your objects, they may default to lists (this is often the case with 2-dimensional data, like dataframes and tables). While that might seem like a non-issue, it can cause errors when you are using packages that assume a particular type of input (for ex., ggplot).

If you are given a new object in R, how do you check if it is a vector, or a list, or a factor (we will see these

later on in the course)?

```
typeof(vector_random)
```

```
## [1] "double"
```

```
typeof(list_random)
```

```
## [1] "list"
```

PAUSE AND THINK

If you are following along with the RMarkdown, pause and take a look at your RStudio environment at this point. We have created many new objects, and you'll see these listed in the Environment tab in the top right corner of your RStudio window. You can also switch the tab to 'History', where it will show a list of all the commands you have run recently.

How might this RStudio window be useful when you are trying to debug an issue?

Accessing 2D data - Lists and dataframes

Now let us move on to two-dimensional objects (with rows and columns).
When undertaking analyses with relational data (associating something in the rows with something in the columns), we would ideally like to keep our data in a rectangular form that maintains these relationships automatically. For this purpose, there are 2 types of data structures in R - dataframe (a list of variables), and matrix.

We will start by loading some sample data that comes pre-installed with RStudio.

```
mydat = datasets::PlantGrowth
typeof(mydat)
```

```
## [1] "list"
```

As you see, by default, this object is a dataframe (defined as a list). We can take a quick look at what it looks like. The **head()** command prints the first 6 'elements' (in this case, rows) in this object.

```
head(mydat)
```

```
##   weight group
## 1   4.17  ctrl
## 2   5.58  ctrl
## 3   5.18  ctrl
## 4   6.11  ctrl
## 5   4.50  ctrl
## 6   4.61  ctrl
```

As you see, we are able to define two different types of columns here - a 'double', and a 'factor'.

What is a factor?
On occassion, instead of having strings in a column simply as text, we might want to imbue some hierarchy in their definition. For example, in the example above, we can associate a different level to every string term in the column 'group'. What does this look like?

```
print(mydat$group)
```

```
##  [1] ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl trt1 trt1 trt1 trt1 trt1
## [16] trt1 trt1 trt1 trt1 trt1 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2
## Levels: ctrl trt1 trt2
```

As you see, we have 3 levels for this column - 'ctrl', 'trt1', and 'trt2'. This means that these are 3 separate categorical variables now, with the 'ctrl' group being our *reference group* since it is the first level. This will become particularly relevant when you are using your data to fit linear models and compare different experimental groups, later on in the course.

Note: A lot of default functions that we will be using for reading in data, automatically convert strings to factors. This can be tweaked with the 'stringsAsFactors' flag in the respective commands. Use the **Help** function for the `read.table` command to see an example.

Being able to define different data types for different columns can be a curse or a blessing, depending on what you want to do.

When you want **all your columns to be of the same type** - In this case, it is best to ***coerce*** your dataset into a matrix, like so:

```
mydat_mat = as.matrix(mydat)
typeof(mydat_mat)
```

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

```
head(mydat_mat)
```

```
##      weight group
## [1,] "4.17" "ctrl"
## [2,] "5.58" "ctrl"
## [3,] "5.18" "ctrl"
## [4,] "6.11" "ctrl"
## [5,] "4.50" "ctrl"
## [6,] "4.61" "ctrl"
```

- In this case, the type of the data structure is a **character matrix**
- Matrices are also usually the required type of input when you need to undertake any linear algebra on your dataset.

However, if **all your columns need to be of different types** (numeric, character etc) - Stick with a dataframe, - This is generally the default data structure your data will be read into, if you use commands like 'read.table', or 'read.csv'. - Here is our current example, as a dataframe:

```
mydat_df = mydat
typeof(mydat_df)
```

```
## [1] "list"
```

```
head(mydat_df)
```

```
##   weight group
## 1   4.17  ctrl
## 2   5.58  ctrl
## 3   5.18  ctrl
## 4   6.11  ctrl
## 5   4.50  ctrl
## 6   4.61  ctrl
```

We can also look at the dimensions of these objects,

```
dim(mydat)
```

```
## [1] 30  2
```

```
dim(mydat_df)
```

```
## [1] 30  2
```

You can quickly summarize the information in each column of a dataframe using the **summary()** command.

```
summary(mydat_df)
```

```
##      weight       group
##  Min.   :3.590   ctrl:10
##  1st Qu.:4.550   trt1:10
##  Median :5.155   trt2:10
##  Mean   :5.073
##  3rd Qu.:5.530
##  Max.   :6.310
```

Viewing different aspects of the data

We can get the column names for our dataframe.

```
colnames(mydat)
```

```
## [1] "weight" "group"
```

We can select the first row in the dataframe,

```
mydat[1,]
```

```
##   weight group
## 1   4.17  ctrl
```

or the first column

```
mydat[,1]
```

```
##  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17 4.41 3.59 5.87
## [16] 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

We can also select all the rows, with a particular column only. For example, here we see all the entries in our dataset, where the value of the column *group* == 'ctrl'.

PAUSE AND THINK

- Notice the use of '==='. This expression is used to evaluate whether A is equal to B. Execute the command `2==3` in the console and see what happens. Now enter `2==2`. This expression always returns a 'TRUE' or 'FALSE' (i.e. a *boolean value*).

- What happens when you execute the command, `2=3`? What is different here?

Now let us go ahead and subset our data, where the group *is equal to* 'ctrl'.

```
mydat[mydat$group=="ctrl",]
```

```
##    weight group
## 1    4.17  ctrl
## 2    5.58  ctrl
## 3    5.18  ctrl
## 4    6.11  ctrl
## 5    4.50  ctrl
## 6    4.61  ctrl
## 7    5.17  ctrl
## 8    4.53  ctrl
## 9    5.33  ctrl
## 10   5.14  ctrl
```

We can also compute some basic stats on this data subset, like so:

```
#Print the min, max, quartile ranges, median, and mean values
summary(mydat[mydat$group=="ctrl","weight"] )
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.170   4.550   5.155   5.032   5.293   6.110
```

```
#Log transform the data
log(mydat[mydat$group=="ctrl","weight"] )
```

```
##  [1] 1.427916 1.719189 1.644805 1.809927 1.504077 1.528228 1.642873 1.510722
##  [9] 1.673351 1.637053
```

Lastly, we can get the number of **unique** entries in a column, using the function `unique()`. First we select just the column 'group', and then we unique- the results, as shown here:

```r
unique(mydat$group)
```

```
## [1] ctrl trt1 trt2
## Levels: ctrl trt1 trt2
```

Accessing elements in a list (or data.frame)

In R, a `data.frame` is a special case of a `list`. We say that `data.frame` **inherits** from `list`. We can index elements of a list either using `[[]]` or the `$` operator, which have the same behavior:

```r
tmp <- list(
    elem1 = "one",
    elem2 = 2.2,
    elem3 = 3,
    elem4 = "4"
)
print(tmp$elem1)
```

```
## [1] "one"
```

```r
print(tmp[["elem1"]])
```

```
## [1] "one"
```

```r
print(tmp[[1]])
```

```
## [1] "one"
```

```r
print(tmp$"elem1")
```

```
## [1] "one"
```

Notice that above we have shown two things: 1. The difference in syntax between accessing named elements of a list 1. The use of initiallizing named elements of a list

`$` is a recursive operator, and accesses the next level down of a list, making the following command possible:

```r
tmp <- list(
  a = list(
    b = "c"
  )
)
print(tmp$a$b)
```

```
## [1] "c"
```

We can use the similar logic as above to access columns of a data.frame:

```r
tmp <- data.frame(
    elem1 = c("one", "two", "three"),
    elem2 = 2:4,
    elem3 = 3:5,
    elem4 = c("4", "5", "6")
)
print(tmp$elem1)
```

```
## [1] one   two   three
## Levels: one three two
```

```
print(tmp[["elem1"]])
```

```
## [1] one   two   three
## Levels: one three two
```

```
print(tmp[[1]])
```

```
## [1] one   two   three
## Levels: one three two
```

```
print(tmp$"elem1")
```

```
## [1] one   two   three
## Levels: one three two
```

```
print(tmp[,2])
```

```
## [1] 2 3 4
```

Input/Output

It is one thing to be able to transform data and subset it in different ways. However, sometimes we might want to save some of our transformed data and come back to it later, or send it to collaborators who want to open it in a different software (like Excel, for ex.). Here are some easy ways to do so:

```
write.table(x=mydat, file="test1.csv", sep="\t")
```

You can read this table back into your current R environment using a command like this:

```
mydat_reloaded = read.table(file="test1.csv", header=TRUE, sep="\t")
```

Notice how we are using relative paths for the output file here. How will you find out where these files have been stored? (**HINT:** think about how you are able to query the Current Working Directory in R).

Would this work?

```
mydat_reloaded =  read.table(file=paste(getwd(), "/test1_cwd.csv", sep=""), header=TRUE, sep="\t")
```

Where is this file saved? Is it saved at the same place as the first one? Often, the location defined by the relative path may be different than where the *current working directory* is. Look at the use of the `setwd()` function (`?setwd`) to first set the current working directory to your desired location.

Additional food for thought

- You might have noticed the interspersed `\pagebreak` in the R Markdown. This is a quick line that lets you start the next bit of your report on a new page, and will come in handy when you want to be able to separate different sections not just using the section headings but by different page ranges, later on in the course.

- Review the *Pause and Think* sections. Are you able to follow along with the discussion?

- You have successfully managed to save an output file, and load it back in, in this brief tutorial. You wrote to the current working directory using a *relative path* for your files. Try doing this using an absolute path instead. HINT: You know how to get the absolute address of the directory you are currently in. Look at how to use this in conjunction with the `paste()` function.

Yay, you have worked through this R Markdown successfully! Now press 'Knit' at the top in the Editor window - are you able to generate a .pdf version of this, similar to the one you were following?