Starting Your Analysis in R

Joseph Nathan Cohen
Department of Sociology
City University of New York, Queens College
Fall 2019

Introduction

In this module, we will learn how to begin an analysis in R. We will learn about:

- Installing R and RStudio (pg. 1)
- The RStudio user interface (pg. 1)
- Performing basic calculations (pg. 2)
- Working with objects (pg. 3)
- First steps in starting an analysis (pg. 6)
- Loading data tables (pg. 8)
- Cleaning data tables (pg. 10)
- Naming variables (pg. 14)
- Saving data tables

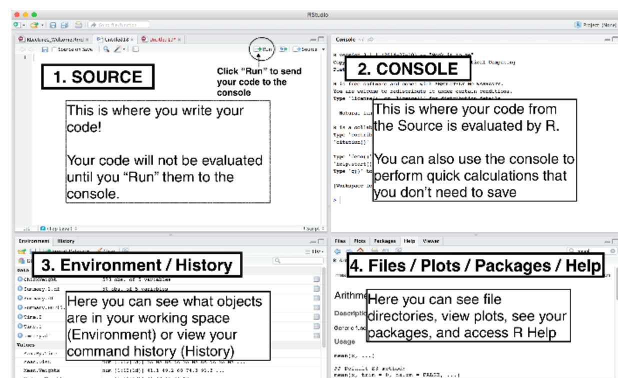
Installing R and RStudio

To perform an analysis in R, you need to install the program. Your first step is to download R from the Comprehensive R Archive Network (CRAN) at https://cran.r-project.org/

Next, install *RStudio*, a program that creates a user-friendly environment for using R. Download and install the program from https://www.rstudio.com/products/rstudio/download/

The RStudio User Interface

Throughout our lessons, we will routinely talk about the RStudio user interface. The interface is described in the figure to the right.



- ***Source***. Write scripts for your analysis
- ***Console***. Can use R interactively here. Displays the results of processed R commands.
- ***Environment***. An environment is a set of associated objects (data, functions, etc.) loaded into R. For more on environments, click here
- ***History***. A list of commands that have been run in your R session.
- ***Files***. Lists files in a folder.
- ***Plots***. Shows graphs that you've created.
- ***Packages***. Packages are libraries of commands. You can install and load packages in this window.
- ***Help***. For information on packages and commands.
- ***Viewer***. Shows local web content.

Basic Calculations in R

NOTE: Do not use commas in numbers. R will interpret them as character strings.

Addition

```
105 + 25

## [1] 130
```

Subtraction

```
75 - 40

## [1] 35
```

Multiplication

```
20 * 35

## [1] 700
```

Division

```
2250 / 5

## [1] 450
```

Exponentiation

```
5^3

## [1] 125
```

Square Root

```
sqrt(150)

## [1] 12.24745
```

Logarithms

```
log(3)

## [1] 1.098612
```

Natural exponents

```
exp(1.0986)

## [1] 2.999963
```

Complex Operations

```
3 + 5 * 6

## [1] 33

(3 + 5) * 6

## [1] 48
```

Exercise. Using R interactively, perform the following calculations:

1. 225 plus 67
2. 394,500 divided by 5
3. 2 to the exponent 16
4. The logarithm of 400,000
5. There are twenty-one players on Team A. There are twenty-five players on Team B. Combined, the two teams scored 2,100 points this season. On average, how many points did the players on Team A and B (combined) score?

Working with Objects

In R, you can store data into **objects**, and then manipulate those objects. You can store many things as objects, including numbers, names, lists, and data tables.

You define objects using arrows (<-) or (->). For example:

```
months <- c("January", "February", "March", "April", "May", "June",
            "July", "August", "September", "October", "November", "December")
```

Here, we created an object called "months" and attached a list of months to it. This object is a *vector*, a list comprised of multiple numbers or characters.

Whenever we evoke the object name "months", it will return the list:

```
months

##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"
```

Note that we define a list using the *collector operator* – **c()** – a small-case "c" with parentheses directly attached to it. Whenever you are listing more than one object in a sequence, use the collector operator. Also note that, when we list words or characters, we enclose them in quotation marks.

We can also create objects made of numbers:

```
days <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
```

Again, when we involve the object name, we get the numbers.

```
days

##  [1] 31 28 31 30 31 30 31 31 30 31 30 31
```

Once the object is defined, we can perform calculations with it. A mathematical operation performed on an object will be performed on all of its elements:

```
hours <- days * 24
hours

##  [1] 744 672 744 720 744 720 744 744 720 744 720 744
```

And we can also perform statistical operations on the entire vector:

```
mean(days)

## [1] 30.41667

sd(days)

## [1] 0.9003366
```

Data Frames

Once special kind of object with which we will work often is the *data frame*, a data table. Let's create a data frame using our months, days, and hours variables. We'll add minutes and seconds for fun.

```
minutes <- hours * 60
time <- data.frame(months, days, hours, minutes)

#Create a new variable in the data frame demarcating seconds
time$seconds <- time$minutes * 60

time

##       months days hours minutes seconds
## 1    January   31   744   44640 2678400
## 2   February   28   672   40320 2419200
## 3      March   31   744   44640 2678400
## 4      April   30   720   43200 2592000
## 5        May   31   744   44640 2678400
## 6       June   30   720   43200 2592000
## 7       July   31   744   44640 2678400
## 8     August   31   744   44640 2678400
## 9  September   30   720   43200 2592000
## 10   October   31   744   44640 2678400
## 11  November   30   720   43200 2592000
## 12  December   31   744   44640 2678400
```

Each of our vectors becomes a row on the data table. To reference a column (or variable) from a data table:

```
time$months

##  [1] January   February  March     April     May       June      July
##  [8] August    September October   November  December
## 12 Levels: April August December February January July June March ... Sept
ember
```

Perform statistical operations on a column of a data table:

```
mean(time$days)

## [1] 30.41667

table(time$days)

##
## 28 30 31
##  1  4  7
```

You can call out cells, rows, or tables. Using square brackets, you can call out the row or column numbers. The first number in the square bracket is the row number, and the second is the column number.

```
time[3,]

##   months days hours minutes seconds
## 3  March   31   744   44640 2678400

time[,3]

##  [1] 744 672 744 720 744 720 744 744 720 744 720 744

time[3,3]

## [1] 744
```

And you can perform calculations:

```
time[3,3] * 100

## [1] 74400
```

Exercise. Using R interactively:

1. Create a vector containing the name of New York's five boroughs: Manhattan, Queens, Brooklyn, Bronx, and Staten Island.

2. Reproduce this table as a data frame:

Borough	Population	Murders
Manhattan	1,538,096	41
Brooklyn	2,465,689	128
Queens	2,229,394	47
Bronx	1,332,244	91
Staten Island	443,762	21

3. Calculate the murder per capita rate. Which borough had the highest murder per capita rate?

First Steps in an Analysis

To begin an analysis:

1. Begin a script (pg. 6)
2. Clear R's memory (pg. 7)
3. Set the working directory (pg. 7)

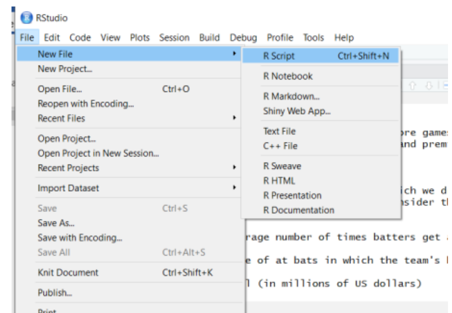
Step 1: Begin a Script

A *script* is a text file that contains a series of R commands. When you execute the script, R runs all the commands in the text file.

Scripts allow you to:

- Remember the commands you used in an analysis
- Reproduce your results
- Retrace your steps in order to find errors
- Reuse your code instead of entering it interactively over and over again

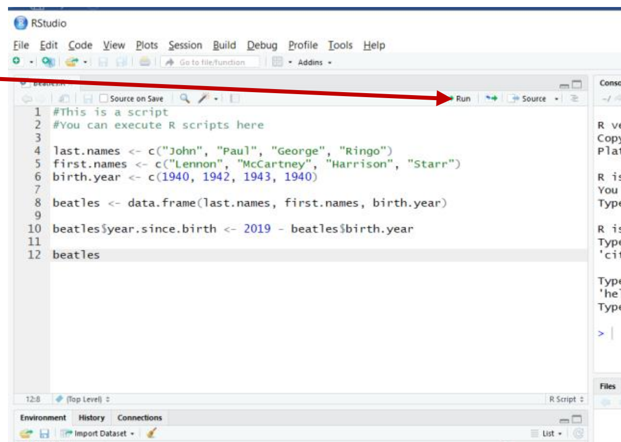
To start a script in R:



1. Go to the Menu bar
2. Click **File > New File > R Script.** Alternatively, you can use the Windows hotkeys: CTRL + SHIFT + N
3. Immediately save the file: **File > Save As…**.

A blank file will appear on in the Source window. You enter commands in that window, and save them once entered.

- To run the commands in the script, click the **Run** button.



To leave notes in a script, add a hashtag (#) at the beginning of the line. R will not run lines preceded by a hashtag.

Step 2. Clear R's Memory

I begin each script by purging R's memory. Doing so ensures that your script will work from scratch. There are two commands to clear R's memory: **rm()** and **gc()**.

```
rm(list = ls())
gc()
```

The **rm()** command is used to remove objects from the memory. For example, the command `rm(time)` would remove the object `time` from R's memory.

The **ls()** command lists all the objects currently in R's memory. So the command `rm(list=ls())` asks R to remove all objects listed by the command **ls()**

The **gc()** command is the "garbage collector". It purges R's memory. After you remove all of the objects in R's memory, the garbage collector will remove any remaining objects from R's memory.

Note that these commands will not clear the console window. To do that, click the brook icon to the top right of the Console.

<u>Structure of R Commands</u>

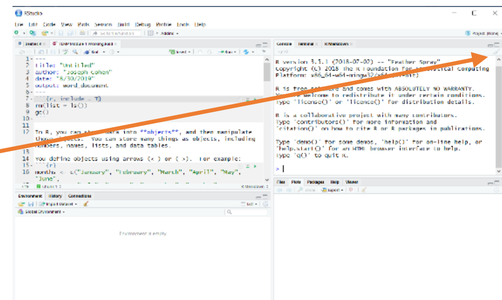
R commands are structured as words immediately followed by parentheses.

You enter the command's options within the parentheses.

To learn more about a command's options, a question mark (?) and the command's name.

Example for the command **mean()**:

```
?mean
```



Step 3. Set the Working Directory

Your computer has thousands of file folders. You must specify which directory will contain your analysis' source files and output. To set the working directory, use the **setwd()** command, followed by your working directory's path enclosed in quotation marks:

```
setwd("C:/Users/jncohen/Dropbox/Teaching/DATA 712/Week 2")
```

Remember that Windows paths generally have a forward slashes (\), whereas R uses backslashes (/). ***You must reverse the slashes from those used in Windows!***

Loading Data Tables

The next step in an analysis is to load the data you are using. You create an object with the data table attached to it, and then work with that object in R.

The appropriate commands depend on the format in which the data is stored. The table to the right describes commands that can be used to import common data types. We will focus on importing these data types here.

Format	Command	Package
CSV	read.csv()	*Base*
Excel	read_xlsx()	readxl
Stata	read.dta()	foreign
SAS Transport	read.xport()	foreign
SPSS	read.spss()	foreign

Pulling Data from the Web. If you are interested in web scraping, the package ***rvest*** offers useful tools. You can find a tutorial on how to use rvest here.

To query APIs, check out the packages ***httr*** and ***jsonlite***. A tutorial on using APIs is here.

Data from CSVs

CSVs are *comma-separated value* files. They store data tables in plain text files. Each new row on the data table is demarcated by a line in the text file. Columns are demarcated by commas.

To load CSV data, we use the **read.csv()** command. Below, we load the data from a file called "OECD Better Life Data.csv" in our working directory:

```
data <- read.csv("OECD Better Life Data.csv")
```

When you data into memory, you can check the first few rows by using the **head()** command on the object to which you attached your data.

```
#To Look at the top few rows of your data:
head(data)
```

```
##          country nofacilities houseexp rooms dispinc finwealth labinsec
## 1      Australia           NA       20    NA   32759    427064      5.4
## 2        Austria          0.9       21   1.6   33541    308325      3.5
## 3        Belgium          1.9       21   2.2   30364    386006      3.7
## 4         Canada          0.2       22   2.6   30854    423849      6.0
## 5          Chile          9.4       18   1.2      NA    100967      8.7
## 6 Czech Republic          0.7       24   1.4   21453        NA      3.1
```

Loading Data from Excel

To load data from Excel, we use the command **read.xlsx()** from the ***realxl*** package.

```
library(readxl)
```

Then create an object called "data" with the data from the first sheet of our Excel file:

```
data <- read_xlsx("OECD Better Life
Data.xlsx", sheet = 1)
```

For our current purposes, you might convert the object into a data frame, and check its first few rows:

```
data <- data.frame(data)

head(data)
```

A Super-Quick Primer on Packages

Many R commands aren't available when you first load the program. To use them, you must install and load libraries. To load a library (e.g., ***readxl***):

```
library(readxl)
```

You do not have to re-load packages in a single session.

To install a new library:

```
install.packages("readxl", repos =
"https://cloud.r-project.org")
```

You can change the "repos" setting to query a different source.

```
##           country nofacilities houseexp rooms dispinc finwealth labinsec
## 1       Australia           NA       20    NA   32759    427064      5.4
## 2         Austria          0.9       21   1.6   33541    308325      3.5
## 3         Belgium          1.9       21   2.2   30364    386006      3.7
## 4          Canada          0.2       22   2.6   30854    423849      6.0
## 5           Chile          9.4       18   1.2      NA    100967      8.7
```

From Other Statistical Programs

You can load data from SPSS, Stata, or SAS using commands from the ***foreign*** package. For example, one loads a Stata *.dta file using the **read_dta()** command:

```
library(foreign)

data <- read.dta("OECD Better Life Data.dta")
```

For SPSS, we use the **read.spss()** command. For SAS Transport files we use **read.xport()**

Cleaning Data

Cleaning data involves ensuring that our analysis variables are properly coded, purged of errors, and ready to analyze. For our purposes, we will focus on three kinds of jobs:

1. Ensuring that R recognizes data as their correct types (pg. 11)
2. Recoding unrealistic or missing values (pg. 12)
3. Creating derivative variables

Here, we are going to use a "dirty" sample data set "Sample Data.csv". It is comprised of three variables:

- **age:** The respondent's age (in years)
- **college:** Whether the respondent has a college degree (=1) or not (=0)
- **rating:** The respondent's rating for our restaurant, from 1 (worst) to 5 (best)

We load the data, check out the first few rows, and look at a summary of the data:

```
sample.data <- read.csv("Sample Data.csv")
head(sample.data)

##   age college rating
## 1  33       0      2
## 2  35       0      3
## 3  57       0      2
## 4  37       0      3
## 5  61       0      4
## 6  42       0      4

summary(sample.data)

##       age          college          rating
##  41     : 13   Min.   :0.00   Min.   :1
##  34     : 10   1st Qu.:0.00   1st Qu.:2
##  37     :  9   Median :0.00   Median :3
##  43     :  9   Mean   :0.34   Mean   :3
##  46     :  8   3rd Qu.:1.00   3rd Qu.:4
##  51     :  8   Max.   :1.00   Max.   :6
##  (Other):143
```

An examination of the variable summaries suggest a few problems. First, the variable "age" seems to have problems. It is a continuous variable, and R's **summary()** command usually summarizes these variables as "college" and "rating" are summarized above. Second, the "college" variable seems to be coded as continuous, as opposed to discrete. Third, the variable "rating" has a maximum of 6 (off its scale) and seems to be coded as a continuous variable, even though it is an ordinal one.

Let's tackle these problems.

Ensuring R Recognizes Correct Data Type

Continuous Variables Miscoded as Discrete

There are supposed to be three types of data in this set. “Age” is a continuous variable. “College” is a discrete variable. “Rating” is an ordinal variable. Other types are possible, like string variables (words).

Above, we got some indication that “age” is miscoded. We take a closer look at the variable, and this is what we see:

```
sample.data$age

##   [1] 33  35  57  37  61  42  26  35  51  37  49  54  139 49  45  28  43 
##  [18] 53  37  46  28  40  54  43  39  62  46  29  64  44  40  21  30R 50 
##  [35] 57  52  28  51  52  25  40  53  41  45  28  46  42  42  56  44  41 
##  [52] 43  36  32  43  37  43  58  63  38  22  24  44  32  47  53  32  21 
##  [69] 43  23  36  41  34  57  57  27  51  50  41  27  47  23  43  30  47 
##  [86] 52  34  58  62  46  40  47  39  59  19  36  43  35  34  37  42  48 
## [103] 41  40  34  45  33  52  46  60  33  65  62  59  23  41  29  50  28 
## [120] 24  41  34  41  49  35  38  34  41  21  54  53  42  37  41  42  31 
## [137] 32  31  34  60  54  39  34  36  46  51  31  51  36  56  55  52  39 
## [154] 51  41  29  68  52  30  30  34  38  51  36  66  46  28  40  47  38 
## [171] 59  39  43  58  46  53  26  40  37  25  63  33  45  51  41  42  37 
## [188] 25  36  52  37  54  27  53  61  32  41  24  34  45 
## 50 Levels: 139 19 21 22 23 24 25 26 27 28 29 30 30R 31 32 33 34 35 ... 68
```

It looks like someone accidentally added a character to one respondent’s age. This caused R to read this variable as a collection of labels, rather than numbers.

What should we do? It is tempting to replace this data with the number 30. My preference is to replace these errors with missing data.

To purge a numeric variable of observations with non-numeric characters, ask R to read the variable labels, and then store any numeric labels as numbers using the commands **as.character()** and **as.number()**:

```
sample.data$age <- as.numeric(as.character(sample.data$age))

## Warning: NAs introduced by coercion
```

When we perform these operations, R recodes any observation with non-numeric values as missing. We now recheck the variable using the command **summary()**:

```
summary(sample.data$age)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##   19.00   34.00   41.00   42.49   51.00  139.00       1
```

It looks like we still have a problem: There is a maximum age of 139, which looks like an unrealistic value. In these situations, my preference is to recode these as missing. Let’s turn to resolving that problem next.

Discrete Variables Coded as Continuous

The opposite situation is occurring with the variable "college". It is a discrete or qualitative variable, that R is coding as continuous.

In R, qualitative or discrete data are called "factors". We can recode such a variable as a factor variable by using the **factor()** command:

```
sample.data$college <- factor(sample.data$college, ordered=F)
```

We use the "ordered = F" option to tell R that our variables aren't ordinal.

If we summarize the data again, we see the variable summarized as if it were a discrete variable:

```
summary(sample.data$college)

##   0   1
## 132  68
```

Recoding Unrealistic or Missing Values

Sometimes, data sets give missing values extreme scores, like "-99" or "999". R will think these are real numbers if we don't change them. Likewise, sometimes we come across variable values that look like data entry errors, as with the 139 year-old in our data set above. Both issues are resolved with the **ifelse()** command.

The syntax of this command is:

```
ifelse(condition, value if true, value if false)
```

So, for the "age" variable above, we might pick a realistic maximum variable, and recode any variables that are above that value as missing:

```
sample.data$age <- ifelse(sample.data$age >= 100, NA, sample.data$age)
```

When we summarize the variable, the values look more realistic:

```
summary(sample.data$age)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   19.00   34.00   41.00   42.01   51.00   68.00       2
```

We also had the sample problem with our "rating" variable:

```
sample.data$rating <- ifelse(sample.data$rating > 5, NA, sample.data$rating)
```

And "rating" is an ordinal discrete variable, so we have to recode it as the proper type (as done above):

```
sample.data$rating <- factor(sample.data$rating, ordered = T)
```

It works the same with missing value codes. Let's say we have a variable in which missing values are coded as -99:

```
fake.variable <- ifelse(fake.variable == -99, NA, fake.variable)
```

A summary of logical operators in R:

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
!is.na(*variable.name*)	Is not missing

Creating Derivative Variables

Another step in data preparation is to create *derivative variables* – variables created from other variables in our set.

For example, I might create a "seniors" variable that denotes senior citizens in my set:

```
sample.data$seniors <- ifelse(sample.data$age > 65, 1, 0)
sample.data$seniors <- factor(sample.data$seniors, ordered=F)
summary(sample.data$seniors)

##    0    1 NA's
##  196    2    2
```

I can also create continuous variables:

```
sample.data$age.months <- sample.data$age*12
summary(sample.data$age.months)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   228.0   408.0   492.0   504.1   612.0   816.0       2
```

Naming Variables

Use the **names()** command to see the names of the variables in the data set:

```
names(sample.data)

## [1] "age"     "college" "rating"
```

You can change the name of a particular variable in a set as follows:

```
names(sample.data)[1] <- paste("years")
names(sample.data)

## [1] "years"      "college"    "rating"     "seniors"    "age.months"
```

Or you can rename all variables like this:

```
names(sample.data) <- paste(c("year", "education",
                              "scores", "older", "months"))
head(sample.data)

##   year education scores older months
## 1   33         0      2     0    396
## 2   35         0      3     0    420
## 3   57         0      2     0    684
## 4   37         0      3     0    444
## 5   61         0      4     0    732
## 6   42         0      4     0    504
```

Exercise. Download the set "Practice Data.xlsx" in Excel format. The data is on sheet #1, and the codebook is on sheet #2.

- Clean the data
- Give the variables new names that reflect the contents of the variables

Saving Data

You can save your data as a CSV using the **write.csv()** command:

```
write.csv(sample.data, file = "My Sample Data.csv")
```

Or you can save it as an R format set:

```
saveRDS(sample.data, file = "My Sample Data.RDS")
```

To load that set:

```
new.data <- readRDS("My Sample Data.RDS")
names(new.data)

## [1] "year"      "education" "scores"    "older"     "months"
```