Session Five - Importing and Exporting Data

By Hellen Gakuruh

May 26, 2016

Table of Contents

Session Goal

The main goal of this session is to equip you with data importation and exportation skills.

What we shall cover

By the end of this session you should:

- Be able to import various data structures
- Appreciate some of the challenges in importing data into R and how to address them
- Know how to export data
- Have skills to quickly inspect data for conformity or quality

Importing Data into R

Importing or what is commonly referred as `reading` data into R can be quite a challenge. One of the easiest ways to overcome these challenges is to convert import data into delimited files like text files (.txt) and comma separated files (.csv). Delimited files belong to the free format file type meaning they have no formatting and therefore they are easy to read into R.

However, in most cases one is not able to convert data files into delimited format, in this case there some key aspects about the file to be imported and its content you need to know before you can read it in. These are:

- **Encoding**: To understand encoding and its importance, think back to a time when you received some unreadable or corrupted text (common on emails), this could have been due to the character encoding used or simply the conversion rules applied to convert characters (human readable) to bits (computer language - 0s and 1s). There are many types of encoding applicable for one set of text or another, for example, Japanese or other exotic language would have their own encoding as do English and other languages. It's vital to know the correct encoding and whether your platform (underlying computer system that runs your application e.g. Windows, Mac or Linux) supports it. A list of all supported R encoding can be sources with a call to `iconvlist()`. Please read this article, it's a great-read even for non-programmers.
- **Heading**: If data set to be imported has a header as its first row, then you need to tell R to treat that row as header otherwise R will read it in as actual data values.

- **Row names**: Similarly, if import data has row names either as a column in the data set or another character vector, you can let R know this to avoid automatic numbering of rows.
- **Separator**: It's important to be aware of how text or data values in the import data separated. Separation here means the distinguishing character used between columns. These characters are referred to as `delimiters` some of which are comma, tab, semi-colon, and back ticks.
- **Decimal**: Continuous data of type double can have either a period (4.23) or a comma (e.g. 4,23) depending on locale, R used a period, hence any other decimal point needs to be declared for R to recognize it.
- **Quotation**: Characters or text data are denoted with either double or single quotation marks in R, if this is not the same for the data to be imported, then you need to tell R.
- **Comment**: Most programming or scripting languages have their own way of adding useful or guiding notes to scripts in the form of comments. R uses "#" as do some other programs, any other character needs to be explicitly declared.

- **Missing data**: Missing data in R is represented by a single logical value NA (Not Applicable), any other number or charter should be stated
- **Qualitative data**: Qualitative data or categorical data in R is represented by a class `factor`. By default, R reads in character data as factors, if you do not what this to happen, more so if some of the variables (column) need to be character variables, then you can import data as characters and later convert the necessary variables to factor variables.

Given the foregoing discussion on what needs to be known before importing data into R, it's important to realize that data can be imported from a local drive (file in your computer), a network like the internet, or from a database. For local and internet files, importation can be done through Graphical User Interface (GUI) or the command line. Database files can only be read in through the command line as they require some database techniques.

Using GUI for delimited files

From R Studio's menu bar; go to tools > Import Data set; or from the global environment pane on the tool bar click on import data set. From there select to import a local file or a web Uniform Resource Locator (URL)/internet resource address.

To import a local file, GUI will take you to your current working directory where you either choose a file or move to another directory with the needed file. Once that is done, an import window will appear. From the right side, you can preview the data both as it is on the original source and how it will look like in R. Use the left side to change the options for reading in the data and then click to import the data set.

When data is read into R, the code used will be shown on the console and a viewing window will be opened (see tab on the top left hand side next to your scripts).

To import a web file, click on "From web URL" and then enter the web address with the data file.

Using command line

Using GUI to import data is quite easy, but there is a limit to what it can do, therefore command line (code form) can be use. Using code, you can specify all the details about the import file and its content (just like GUI). Base R and utils packages can be used to import free-formatted files like tab and comma delimited files while packages foreign and haven can be used to import other proprietor kind of files.

Let's look at how to import free formatted (delimited) files and then look at other data files.

Importing delimited files from command line

The core importing function in R is `read.table` function. This function has about 25 arguments all of which address the core elements we mentioned earlier.

The most important of these arguments (important as it has no default) is the file argument which indicates the file path to the data. Other arguments have default values but should be changed as need be; for example, if data has a header, then the argument header should be changed to TRUE, if it a tab delimited file then argument sep should be changed to "", if the locale uses a comma as its decimal point then argument dec should be ",".

Okay, let's import some .txt and .csv files called "tips" located in the data folder. Tips is a data set from "Reshape2" package.

```
tips <- read.table(file = "../../Data/tips.txt", header = TRUE, sep = "\t")
```

Read.table() function has wrapper functions that have default values, for example write.csv has default values for "header", "sep", "dec", "fill" and "coment.char". Wrapper functions are more efficient than their main function in terms of values needed to make a function call.

```
# Using read.csv
tips2 <- read.csv("../../Data/tips.csv")

# Using read.delim
tips3 <- read.delim("../../Data/tips.txt")

# Check datasets are all the same
all.equal(tips, tips2)
## [1] TRUE
all.equal(tips, tips3)
## [1] TRUE

# Tidy up
rm(tips2, tips3)
```

If the file to be downloaded is from a webpage, then "file" will be a web address (URL) for which read.table or its wrapper functions will download and immediately reading in the file.

```
# Reading a .csv from a webpage
urldataset <- read.csv(file = "http://www.hc-sc.gc.ca/data-donnees/por-
rop/cdn-attitudes-healthcare_attitudes-canadiens-system-soins.csv")
```

If you want a copy of the URL file, begin by downloading the file before reading it in. The download.file() can achieve this for any type of file (not necessarily .txt or .csv).

```
# Downloading url data and its metadata
download.file(url = "http://www.hc-sc.gc.ca/data-donnees/por-rop/cdn-
attitudes-healthcare_attitudes-canadiens-system-soins.csv", destfile =
"../../Data/urldataset.csv")
download.file(url = "http://www.hc-sc.gc.ca/data-donnees/cpab-dgcap/cdn-
attitudes-healthcare-data-dictionary-eng.txt", destfile =
"../../Data/urldatadictionary.txt")

# Read in downloaded data
urldataset <- read.csv("../../Data/urldataset.csv")

# Tidying up
rm(urldataset)
```

Importing other files formats from command line

As mentioned, importing csv and text files is rather easy in R, but most often we receive data in other formats like excel, SPSS, Stata or SAS. Base R does not have functions to read these types of files, but there are a number of packages suitable for importing one or more of these files. Most of the functions in these packages eventually call base R's read.table function.

Two of the widely used packages are "foreign" and "haven". In this session, we will have a go at haven (please install it first if you do not have it i.e. install.packages("haven").

```
# Loading required package
library(haven)
```

In the data folder are SPSS (.sav) and Stata (.dta) files. Let's import them into R with the functions read.sav/spss and read.dta respectively.

```
# Importing SPSS data
tipsSav <- read_sav("../../Data/tips.sav")

# Importing Stata
tipsStata <- read_dta("../../Data/tips.dta")

# Tidy up
rm(tipsSav, tipsStata)
```

For URL files, download them as we did with free-format files and then read them in as we have just done.

When importing spreadsheet files like excel, it's highly recommended you convert them to a free-format file like csv due to the fact that an excel file can have many spreadsheets, formula and or macros.

In the event you cannot do so, then I recommend you read R's Data Import/Export manual chapter 9 "Reading Excel spreadsheets". It's extensive and well written, but for this session we will explore one of the suggested packages for reading excel 2007 files, the "xlsx" package, we can use it to read in tips.xlsx.

Small pointer, this package is dependent on Java, please install it first, then install the package.

```
# Installing package and its dependencies
install.packages("xlsx")

# Load installed package
library(xlsx)
```

Read in excel sheet with "read.xlsx" function.

```
tipsExcel <- read.xlsx(file = "../../Data/tips.xlsx", sheetName = "tips")

# tidying up
rm(tipsExcel)
```

Importing Data Base files

There are quite a number of packages that offer linkages between R and databases. Most of these packages can import part or the full database as data frames using database semantics like select, from, by, query, join e.t c.

Database and indeed database management (DBM) is an extensive area and unless you work with databases, this might be a bit out of scope for this session, otherwise please read R's Data Import/Export chapter 4 on Relational databases, it's a great read.

Importing data with Scan function

A more efficient function for importing large matrices is the `scan` function. This function reads in vectors and lists detailing the expected data type. Explicitly specifying data type during importation reduces memory usage and thereby making it more efficient.

Let's read in scan.txt, we will use the argument `what` to specify expected data type. We can also import five line of this supposed large matrix.

```
scan("../../Data/scan.txt", what = character(), sep = "\t", nmax = 5)
## [1] "      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]"
## [2] " [1,] c  f  l  t  n  g  y  f  f  w   s  "
## [3] " [2,] m  a  d  y  k  f  m  v  r  p   p  "
```

```
## [4] " [3,] r  y  v  b  y  f  q  o  y  j  d  "
## [5] " [4,] f  k  z  j  x  f  g  n  n  a  c  "
```

Scan() has quite a number of arguments that control how data is imported. I found this great exercise that will get you reading and thinking about scan function as well as making function calls (arguments and values), please have a go at it.

Reading text data

R being a programming language can be able to read and analyse more than numeric data. Its extensibility opens new avenues like text analysis, for example, you can scrape text intensive sites like social media and perform textual analysis.

One of the key base R functions for reading in lines (mostly in textual form) from a connection (local files, URL, databases etc), is the readLines() function (different from readline() which we shall use in level 2).

This function can also be used to inspect certain issues about a data file before importing it. This is done by reading a few line to establish issues like delimiters and quotation marks used. This is more in line with what we will do when we read a few rows of data with read.table().

For now let's use this function to read in a text file with color names and their code.

```
# Reading in text data with readLines
colors <- readLines(con = "../../Data/Web Colors.txt")
```

We can then try and get all colors that have a certain word using regular expression. Regular expression are characters or strings used to search patterns in a given text. For example, here we are using one of R's regular expression to search for the word blue.

```
# Getting all colours that have the word blue using regular expression
grep("blue", x = colors, ignore.case = TRUE, value = TRUE)
##  [1] "    aliceblue:            \"#f0f8ff\","
##  [2] "    blue:                 \"#0000ff\","
##  [3] "    blueviolet:           \"#8a2be2\","
##  [4] "    cadetblue:            \"#5f9ea0\","
##  [5] "    cornflowerblue:       \"#6495ed\","
##  [6] "    darkblue:             \"#00008b\","
##  [7] "    darkslateblue:        \"#483d8b\","
##  [8] "    deepskyblue:          \"#00bfff\","
##  [9] "    dodgerblue:           \"#1e90ff\","
## [10] "    lightblue:            \"#add8e6\","
## [11] "    lightskyblue:         \"#87cefa\","
## [12] "    lightsteelblue:       \"#b0c4de\","
## [13] "    mediumblue:           \"#0000cd\","
## [14] "    mediumslateblue:      \"#7b68ee\","
## [15] "    midnightblue:         \"#191970\","
## [16] "    powderblue:           \"#b0e0e6\","
## [17] "    royalblue:            \"#4169e1\","
## [18] "    skyblue:              \"#87ceeb\","
```

```
## [19] "    slateblue:            \"#6a5acd\","
## [20] "    steelblue:            \"#4682b4\","
```

Importing Challenges

We have covered a few ways to read in data, however, its worthwhile to note some of the challenges experienced during data importation, more so when file information and it's requirements are unknown.

Take for example a case where you do not know how data is stored, in particular, you do not know if it has a header or any other type of formatting. Further, you do not have the program that generated the file, meaning you cannot check its formatting from its original program. How would you solve this problem?

One possible solution is to take a peek at the data by reading in just a few rows of data to determine the delimiter then determine if it has a header.

```
# Reading 5 rows  of sheet 1
read.xlsx(file = "../../Data/tipsUnknown.xlsx", sheetIndex = 1, endRow = 5)
##   X16.99 X1.01 Female No Sun Dinner X2
## 1  10.34  1.66   Male No Sun Dinner  3

# Reading 5 rows of sheet 2
read.xlsx(file = "../../Data/tipsUnknown.xlsx", sheetIndex = 2, endRow = 5)
## data frame with 0 columns and 0 rows
# Now with no header
read.xlsx(file = "../../Data/tipsUnknown.xlsx", sheetName = "header", endRow
= 5, header = FALSE)
##           X1  X2  X3     X4  X5   X6   X7
## 1 total_bill tip sex smoker day time size
```

Here we see that the file has some empty rows at the top since we asked for 5 rows and it returned only two, one of which became a header since it has no header. Checking the second sheet we see that it has no data, however, it has one row as the header which should have been in sheet one. We can now read in data from sheet one without a header and bring the header from sheet one.

```
# Reading in full data ?
tipsExcel <- read.xlsx(file = "../../Data/tipsUnknown.xlsx", sheetIndex = 1,
header = FALSE)

# Adding column names
names(tipsExcel) <- as.vector(as.matrix(read.xlsx(file =
"../../Data/tipsUnknown.xlsx", sheetName = "header", endRow = 5, header =
FALSE)))


# Confirming equality with original tips dataset
all.equal(tips, tipsExcel)
# [1] TRUE
```

```
# Tidy up
rm(tipsExcel)
```

If you have a large data set and wanted to speed up the process of reading it into R, then one way is to specify its column classes.

```
col.class <- c("numeric", "numeric", "factor", "factor", "factor", "factor",
"integer")
tips2 <- read.csv(file = "../../Data/tips.csv", colClasses = col.class)
# Check if both datasets are the same
all.equal(tips, tips2)
# [1] TRUE

# Tidy up
rm(tips2, col.class)
```

You can do this to files you are not sure of.

Exporting Data

Base R and Utils exporting functions

Using base R and Utils package, data and other outputs can be exported using the `write` group of functions (type ?write on the console, RStudio will show you all the available functions that start with write).

One of the key function being write.table(), the equivalent of read.table function for importing data. Lets use it to export our Tips data set as a text and a csv file.

```
# Export data as a text file
write.table(x = tips, file = "../../Data/tipsExport.txt", sep = "\t",
row.names = FALSE)

# View exported data file
file.show("../../Data/tipsExport.txt")

# Export data as a csv file
write.csv(x = tips, file = "../../Data/tipsExport.csv", row.names = FALSE)

# View exported data file
file.show("../../Data/tipsExport.csv")
```

Another interesting function is "write" function used to export matrices to a file or console. This function is useful when you want to print output as it is and not as a vector with the row and column indices.

Do take note that matrices which are two dimensional need to be transposed before exportation, it's also a good idea to indicate the number of columns.

```r
# Write a matrix to a file
mat <- matrix(data = 1:12, nrow = 4)
mat
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12

# Printing to the console
write(t(mat), file = "", sep = "\t", ncolumns = 3)
## 1    5   9
## 2    6   10
## 3    7   11
## 4    8   12
```

Notice the difference between usual print and write? Now we can export it as tab delimited file.

```r
write(t(mat), file = "../../Data/matrixData.txt", sep = "\t", ncolumns = 3)
```

Other useful exporting functions:

- `save` or `dump` for writing R objects (those on your workspace),
- `sink` for diverting R outputs to a file, and
- `cat` for concatenating and printing to the console or file. This is similar to write as it prints text without it being vectorised.

```r
# Saving current objects with a .RData file
#-------------------------------------------------
# Current objects
ls()
# Number of current objects
length(ls())
# Storing current objects
save(list = ls(), file = "tmp.RData")

# Deleting all objects after saving a copy
rm(list = ls())

# Confirm there are no objects
ls()

# Restoring with load function
load("tmp.RData")

# Confirming all objects back
length(ls())

# Divert output to a file
```

```
#-----------------------
# Open diversion file
sink(file = "output.txt", append = TRUE)

# Output to be diverted
set.seed(294859)
x = 1:12
y = rnorm(12, 50, 15)
x * y
x/y
x + y
mean(c(x, y))

# Close diversion
sink(file = NULL)

# See the output file
file.show("output.txt")

# Concatenating and printing to a file connection
#-----------------------------------------------
cat("Tips dataset has", length(tips), "columns", file = "tipsCat.txt")

# View file
file.show("tipsCat.txt")

# Tidying up
unlink(c("../../Data/matrixData.txt", "tmp.RData", "output.txt",
"tipsCat.txt"))
```

Exporting with other packages

We can also use the Haven package to export data to SPSS and Stata

```
library(haven)
# Export to SPSS
write_sav(tips, "../../Data/tips.sav")

# Exporting data Stata
write.dta(tips, "../../Data/tips.dta")
```

Xlsx package can also export for us data into excel. Note, this package can also do further formatting, just read it's documentation (help(package = "xlsx")).

```
library(xlsx)

write.xlsx(tips, file = "../../Data/tipsExport.xlsx", sheetName =
"tipsExport", row.names = FALSE)

# See the excel data
file.show("../../Data/tipsExport.xlsx")
```

```
# A bit of tidying up
unlink(x = c("../../Data/tipsExport.txt", "../../Data/tipsExport.csv",
"../../Data/tipsExport.xlsx"))
```

Taking a Glimpse

When you have managed to import your data, one of the first things you do is to inspect it. R has some handy functions to take a glimpse of data like printing some few observations (at the beginning and at the end), open a data viewer window (to show entire data set but clipped to first 1000 rows) many others.

Let's have glimpse at out "tips" data set

```
# Printing the first 6 rows of the dataframe
head(tips)
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
## 4      23.68 3.31   Male     No Sun Dinner    2
## 5      24.59 3.61 Female     No Sun Dinner    4
## 6      25.29 4.71   Male     No Sun Dinner    4

# Printing the first 3 rows of the dataframe
head(tips, 3)
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3

# Printing the last 6 rows of the dataframe
tail(tips)
##     total_bill  tip    sex smoker  day   time size
## 239      35.83 4.67 Female     No  Sat Dinner    3
## 240      29.03 5.92   Male     No  Sat Dinner    3
## 241      27.18 2.00 Female    Yes  Sat Dinner    2
## 242      22.67 2.00   Male    Yes  Sat Dinner    2
## 243      17.82 1.75   Male     No  Sat Dinner    2
## 244      18.78 3.00 Female     No Thur Dinner    2

# Printing the last 2 rows of the dataframe
tail(tips, 2)
##     total_bill  tip    sex smoker  day   time size
## 243      17.82 1.75   Male     No  Sat Dinner    2
## 244      18.78 3.00 Female     No Thur Dinner    2
```

To view the entire data set (at least up to 1000 rows), View() with a capital V is appropriate.

```
View(tips)
```

We can get all the variable names

```
names(tips)
## [1] "total_bill" "tip"        "sex"        "smoker"     "day"
## [6] "time"       "size"
```

Length of the data set in both directions (rows/case and columns/variables) can also be determined

```
# Number of variables
length(tips)
## [1] 7
ncol(tips)
## [1] 7

# Number of cases
nrow(tips)
## [1] 244
```

One very helpful function as far as data inspection is concerned is the str() function. This function gives detailed information about the data set in a compact way. It sort of combines information gotten from class() function, length and to some extent view.

```
# Objects class
class(tips)
## [1] "data.frame"

# Class of elements in the object tips
sapply(tips, class)
## total_bill        tip        sex     smoker        day       time
##  "numeric"  "numeric"   "factor"   "factor"   "factor"   "factor"
##       size
##  "integer"

# Now to see object class, elements class, length and some values
str(tips)
## 'data.frame':    244 obs. of  7 variables:
##  $ total_bill: num  17 10.3 21 23.7 24.6 ...
##  $ tip       : num  1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3
3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  2 3 3 2 4 4 2 4 2 2 ...
```

Missing data is a great concern in data analysis, it is therefore good to determine whether your data has missing values.

```r
# Since tips is a complete dataset, we shall randomly adding some missing
values
set.seed(8494)
tips[sample(1:nrow(tips), ceiling(nrow(tips) * 0.05)), sample(1:length(tips),
ceiling(length(tips) * 0.40))] <- NA

# Check for misssing values (NA) in the entire dataset
any(sapply(tips, is.na) == TRUE)
## [1] TRUE
```

Something extra extra: you can inspect objects in you workspace with a bit more information than ls() gives, by calling the function ls.str()

Gauge yourself

* Note: All data sets are located in Data/Exercises folder

Do you have the expected tools and skills from this session?

1. What is the first thing you should attempt to do before importing data into R?
2. Name the two methods of importing data into R
3. The core importing function in base R is? What class will the imported data be?
4. Write a code to read in diamonds.csv .
5. How can you download and read a webbased data file
6. Write a code to read in "diamonds.sav" and "diamonds.dta" into R
7. When reading in excel spreadsheets, what is the first thing to consider and is there a function in base R that can import it?
8. Write a code to read in diamonds.xlsx
9. What function is suitable for large matrices? Use this function to read in largeMatrix.csv
10. If you received a data file that you had no basic information about, what would you do to be able to read it into R?
11. Which base R function can give you a compact insight to a given object? Use this function to see diamonds structure

You are ready for the sixth session if you know these

1. To ease importation, the first thing to do is to convert data into a free format file like text or csv files.
2. Data importation can be done either from the graphical user interface (GUI) or from the command line (using code)
3. The core importing function in base R is read.table() which reads in data as a data.frame
4. Diamonds.csv can be imported with: read.csv("../../Data/Exercises/diamonds.csv", sep = "@", dec = ",")

5. To download a web based data set, download.file function is used and read with either write.table/csv/tab etc
6. Diamonds.sav and diamonds.dta can be read in with package foreign and haven among others. To read these two files with the haven package, we load it first, library(haven) then read them with read_sav("../../Data/Exercises/diamonds.sav") and read_dta("../../Data/Exercises/diamonds.dta")
7. The first thing to consider when importing excel files is to convert them to free-format like csv as base R has no functions to read them in, only contributed packages
8. Diamonds.xlsx can be read in with read.xlsx("../../Data/Exercises/diamonds") after loading "xlsx" package with library(xlsx)
9. Scan function is most suitable to read in large matrices. largeMatrix.csv can be read with the following code: scan("../../Data/Exercises/largeMatrix.csv", sep = ";")
10. In the case where there is no information about a file, a possible solution is to read a few rows using read.table or it wrapper functions or read a few lines using readLines function.
11. str function give compact information about an R's function. str(diamonds) is used to see diamonds structure