# Module 2

*Get: Importing, Scraping and Exporting Data with R*

*Dr. Anil Dolgun*

*Last updated: 09 April, 2018*

# Overview

## Summary

All statistical work begins with data, and most data are stuck inside files and databases. Data are arriving from multiple sources at an alarming rate and analysts and organisations are seeking ways to leverage these new sources of information. Consequently, analysts need to understand how to get data from these sources. Module 2 will cover the process of importing data, scraping data from web, and exporting data. First we will cover the basics of importing tabular and spreadsheet data (i.e., .txt, .xls, .csv files). Then, we will cover how to acquire data sets from other statistical software (i.e., Stata, SPSS, or SAS) and databases. As the modern data analysis techniques often include scraping data files stored online, we will also cover the fundamentals of web scraping using R. Lastly, the equally important process of getting data out of R, in other words, exporting data will be covered.

## Learning Objectives

The learning objectives of this module are as follows:


- Understand how to get data from tabular and spreadsheet files
- Understand how to get data from statistical software and databases
- Learn how to scrape data files stored online
- Learn how to export to tabular and spreadsheet files
- Learn how to save R objects

# Reading/Importing Data

The first step in any data preprocessing task is to "**GET**" the data. Data can come from many resources but two of the most common format of the data sources include text and Excel files. In addition to text and Excel files, there are other ways that data can be stored and exchanged. Commercial statistical software such as SPSS, SAS, Stata, and Minitab often have the option to store data in a specific format for that software. In addition, analysts commonly use databases to store large quantities of data. R has good support to work with these additional options. In this section, we will cover how to import data into R by reading data from text files, Excel spreadsheets, commercial statistical software data files and databases. Moreover, we will cover how to load data from saved R object files for holding or transferring data that has been processed in R. In addition to the commonly used base R functions to perform data importing, we will also cover functions from the popular `readr`, `xlsx`, `readxl` and `foreign` packages.

## Reading Data from Text Files

Text files are a popular way to hold and exchange tabular data as almost any data application supports exporting data to the CSV (or other text file) formats. Text file formats use delimiters to separate the different elements in a line, and each line of data is in its own line in the text file. Therefore, importing different kinds of text files can follow a fairly consistent process once you have identified the delimiter.

There are two main groups of functions that we can use to read in text files:

- **Base R functions**: The Base R functions are the built-in functions that are already available when you download R and RStudio. Therefore, in order to use Base R functions, you do not need to install or load any packages before using them.
- **`readr` package functions**: Compared to the equivalent base functions, `readr` functions are around 10× faster. In order to use `readr` package functions, you need to install and load the `readr` package using the following commands:

```
install.packages("readr")
library(readr)
```

## Base R functions

`read.table()` is a multi-purpose function in base R for importing data. The functions `read.csv()` and `read.delim()` are special cases of `read.table()` in which the defaults have been adjusted for efficiency. To illustrate these functions let's work with a CSV (.csv comma separated values) file called iris.csv which is located in our data repository. Before running any command note that we need to save this data set into our working directory or we need to explicitly define the location of this data set.

In the first example, let's assume that we have already downloaded iris.csv data and saved it in our working directory. Then, the following command will read iris.csv data and store it in the `iris1` object in R as a data frame:

```
# The following command assumes that the iris.csv file is located in the work
ing directory

iris1 <- read.csv("iris.csv")
```

Now we can view the header of the `iris1` object using `head()` function as follows:

```
head(iris1)
```

```
##   X Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1 1          5.1         3.5          1.4         0.2  setosa
## 2 2          4.9         3.0          1.4         0.2  setosa
## 3 3          4.7         3.2          1.3         0.2  setosa
## 4 4          4.6         3.1          1.5         0.2  setosa
## 5 5          5.0         3.6          1.4         0.2  setosa
## 6 6          5.4         3.9          1.7         0.4  setosa
```

In the second example, let's assume that the iris.csv data is located in another file path (i.e. on desktop under data folder) "~/Desktop/data/iris.csv". Now we need to provide a direct path to our .csv file depending on where it is located:

```
# The following command assumes that the iris.csv file is located in the "~/D
esktop/data/iris.csv" path

iris2 <- read.csv(file="~/Desktop/data/iris.csv")
```

Another suggested option is to set the working directory where the data is located. To illustrate, assume that the iris.csv is located on your desktop under data folder and you want to set this directory as the working directory. The `setwd()` function will set the working directory to the folder "data":

```
# Set the working directory to "~/Desktop/data"

setwd("~/Desktop/data")
```

Remember that you must use the forward slash / or double backslash || in R while specifying the file path. The Windows format of single backslash will not work.

After that you can read the iris.csv data using:

```
iris3 <- read.csv("iris.csv")
```

Let's check the header of the `iris3` object:

```
head(iris3)
```

```
##   X Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1 1          5.1         3.5          1.4         0.2  setosa
## 2 2          4.9         3.0          1.4         0.2  setosa
## 3 3          4.7         3.2          1.3         0.2  setosa
## 4 4          4.6         3.1          1.5         0.2  setosa
## 5 5          5.0         3.6          1.4         0.2  setosa
## 6 6          5.4         3.9          1.7         0.4  setosa
```

You can also compactly display the structure of an R object using `str()` function.

```
str(iris3)
```

```
## 'data.frame':    150 obs. of  6 variables:
##  $ X           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1
1 1 1 1 ...
```

Note that when we assess the structure of the iris data set that we read in, `Species` is automatically coerced to a factor variable with three levels.

However, we may want to read in `Species` as a character variable rather than a factor. We can take care of this by changing the `stringsAsFactors` argument. The default has `stringsAsFactors = TRUE`; however, setting it equal to FALSE will read in the variable as a character variable.

```
iris4 <- read.csv("iris.csv", stringsAsFactors = FALSE)

str(iris4)
```

```
## 'data.frame':    150 obs. of  6 variables:
##  $ X           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : chr  "setosa" "setosa" "setosa" "setosa" ...
```

Now, you can see that the variable `Species` is a character variable.

As previously stated `read.csv` is just a wrapper for `read.table` but with adjusted default arguments. Therefore, we can use `read.table` to read in this same data. The two arguments we need to be aware of are the field separator ( `sep` ) and the argument indicating whether the file contains the names of the variables as its first line ( `header` ). In `read.table` the defaults are `sep = ""` and `header = FALSE` whereas in `read.csv` the defaults are `sep = ","` and `header = TRUE` .

Therefore, we can also use the `read.table` function to read the iris.csv data. The extra thing we need to specify is the separator and the header arguments. As the data is comma separated and the first line contains the names of the variables, we will use `sep = ","` and `header = TRUE` options:

```
# provides same results as read.csv above
iris5 <- read.table("iris.csv", sep=",", header = TRUE, stringsAsFactors = FA
LSE)
```

Sometimes, it could happen that the file extension is .csv, but the data is not comma separated; rather, a semicolon (;) or any other symbol is used as a separator. In that case, we can still use the `read.csv()` function, but in this case we have to specify the separator.

Let's look at the example with a semicolon-separated file named iris_semicolon.csv which is located under our data repository. After downloading and saving this data file in our working directory we can use:

```
iris6 <- read.csv("iris_semicolon.csv", sep=";", stringsAsFactors = FALSE)
```

Similarly, if the values are tab separated (.txt file), we can use `read.csv()` with `sep= "\t"` . Alternatively, we can use `read.table()` . The following is an example:

```
iris7 <- read.csv("iris_tab.txt",sep="\t")

# provides same results as read.csv above

iris8 <- read.table("iris_tab.txt",header=TRUE)
```

Notice that here when we used `read.table()` , we had to specify whether the variable name is present or not, using the argument `header=TRUE` .

## `readr` package functions

Compared to the equivalent base functions, `readr` functions are around 10× faster. This will make a remarkable difference in reading time if you have a very large data set. They bring consistency to importing functions, they produce data frames in a data.table format which are easier to view for large data sets. The default settings for `readr` function removes the hassles of `stringsAsFactors`, and they are more flexible in column specification.

`read_csv()` function is equivalent to base R 's `read.csv()` function (note the distinction between these two function names!). According to Boehmke (2016), there are two main differences between `read_csv()` and base R 's `read.csv()` functions:

- `read_csv()` maintains the full variable name whereas, `read.csv` eliminates any spaces in variable names and fills it with '.'
- `read_csv()` automatically sets `stringsAsFactors = FALSE`, which can be a controversial topic.

Let's read the `iris.csv` file using `read_csv` function. Note that the `readr` package needs to be installed and loaded before using this function.

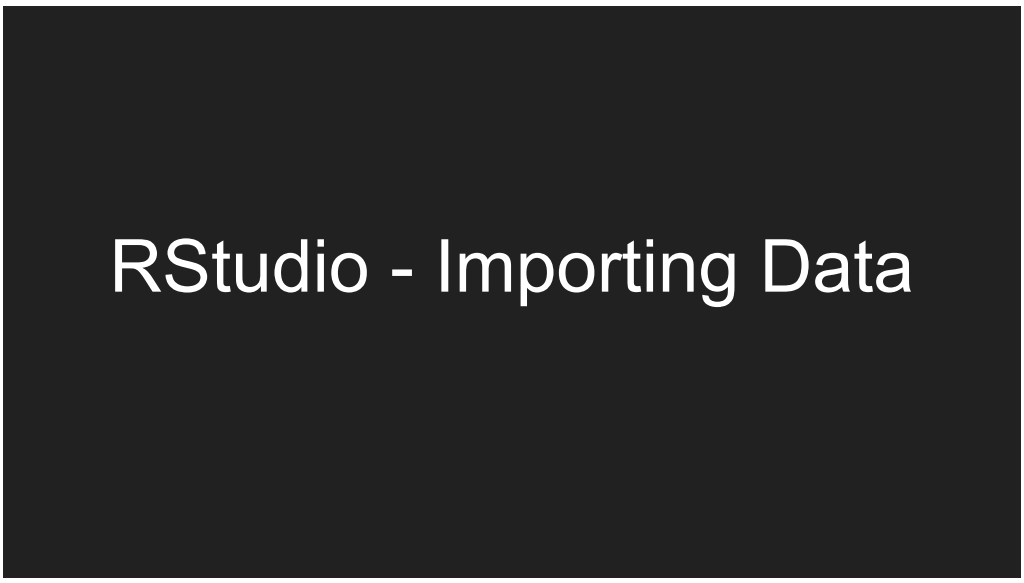
```
install.packages("readr")
library(readr)
```

```
iris9 <- read_csv("iris.csv")
```

```
## Parsed with column specification:
## cols(
##   X1 = col_integer(),
##   Sepal.Length = col_double(),
##   Sepal.Width = col_double(),
##   Petal.Length = col_double(),
##   Petal.Width = col_double(),
##   Species = col_character()
## )
```

When we use `read_csv` function, "Parsed with column specification" information will be reported. Note that this is not a warning and nothing is wrong with your code. You can use this information to check the variable types in your data set. If you want to make adjustments in the variable types, you can use additional arguments inside this function. For more information on `read_csv` function and its arguments type `help(read_csv)`.

The good news is RStudio has the built in "**Import Dataset**" dialog box on the upper-right "**Environment**" pane. You can also use this dialog box to import a wide range of file types including csv, Excel, SPSS, SAS and Stata data files. The following slides (taken from Dr. James Baglin's R Bootcamp notes (https://astral-theory-157510.appspot.com/secured/RBootcamp_Course_02.html#importing_and_exporting_data)) will briefly explain the process of importing a csv data set into RStudio.




Slide 1

More information on `readr` package can be found here: https://cran.r-project.org/web/packages/readr/readr.pdf (https://cran.r-project.org/web/packages/readr/readr.pdf)

# Reading Data from Excel files

Excel is the most commonly used spreadsheet software. Therefore, it's important to be able to efficiently import and export data from Excel. Often, users prefer to export the Excel data file as a .csv file and then import into R using `read.csv` or `read_csv`. However, this is not an efficient way to import Excel files. In this section, you will learn how to import data directly from Excel using two different packages, the `xlsx` and `readxl` packages.

## The `xlsx` Package

If the dataset is stored in the .xls or .xlsx format, we have to use certain R packages to import those files; one of the packages is `xlsx`, which is designed to read files formatted as .xlsx.

To illustrate, we will use the same data from the previous section which is saved as an .xlsx file in our working directory. To import the Excel data we simply install and load `xlsx` package and use the `read.xlsx()` function:

```
install.packages(xlsx)
library(xlsx)
```

```
# read in xlsx worksheet using a sheet index or name

iris10<- read.xlsx("iris.xlsx", sheetName = "iris")
```

Often people make notes, comments, headers, etc. at the beginning or end of the Excel files which we may not want to include. If we want to read in data that starts further down in the Excel worksheet we can include the `startRow` argument.

```
# read in xlsx worksheet starting from third row.

iris11<- read.xlsx("iris.xlsx", sheetName = "iris", startRow = 3)
```

If we have a specific range of rows or columns to include we can use the `rowIndex` or `colIndex` arguments as follows:

```
# read in xlsx worksheet including columns 1-4 and rows 3-5.

iris12<- read.xlsx("iris.xlsx", sheetName = "iris", rowIndex = 3:5, colIndex
 = 1:4)
```

Another useful argument is `keepFormulas` which allows you to see the text of any formulas in the Excel spreadsheet. For more information on `read.xlsx` function and its arguments type `help(read.xlsx)`.

### The `readxl` Package

`readxl` was developed by Hadley Wickham and the RStudio team who also developed the `readr` package. This package works with both .xls and .xlsx formats. Unlike `xlsx` package, the `readxl` package has no external dependencies (like Java or Perl), so you can use it to read Excel data on any platform. Moreover, `readxl` has the ability to load dates and times, it automatically drops blank columns, reads in character variables as characters, and returns outputs as data.table format which is more convenient for viewing large data sets.

To read in Excel data with `readxl` you can use the `read_excel()` function which has very similar operations and arguments as `read.xlsx`. Here are some examples:

```
install.packages(readxl)
library(readxl)
```

```
# read in xlsx worksheet using a sheet index or name

iris13<- read_excel("iris.xlsx", sheet = "iris")
```

```
# read in xlsx worksheet and change variable names by skipping the first row
# and using col_names to set the new names

iris14<- read_excel("iris.xlsx", sheet = "iris", skip = 1, col_names = paste
 ("Var", 1:6))
```

More information on `read_excel` function and `readxl` package can be found here: https://cran.r-project.org/web/packages/readxl/readxl.pdf (https://cran.r-project.org/web/packages/readxl/readxl.pdf).

# Importing Data from statistical software

The `foreign` package provides functions that help you read data files from other statistical software such as SPSS, SAS, Stata, and others into R. To import an SPSS data file (.sav) into R, we need to call the `foreign` library and then use the `read.spss()` function. Similarly, if we want to import a STATA data file, the corresponding function will be `read.dta()`. Here is an example of importing an SPSS data file:

```
install.packages("foreign")
library(foreign)
```

```
# read in spss data file and store it as data frame

iris_spss <- read.spss("iris.sav", to.data.frame = TRUE)
```

Note that we set the `to.data.frame = TRUE` option in order to have a data frame format, otherwise, the defaults ( `to.data.frame = FALSE` ) will read in the data as a list.

More information on `foreign` package can be found here: https://cran.r-project.org/web/packages/foreign/foreign.pdf (https://cran.r-project.org/web/packages/foreign/foreign.pdf)

Remember that you can also use the "**Import Dataset**" dialog box on the upper-right "**Environment**" pane to import SPSS, SAS and Stata data files instead of using the `foreign` package.

# Reading from Databases

A data set can be stored in any format whereas large-scale data sets are generally stored in database software. Commonly, large organisations and companies keep their data in relational databases. Therefore, we may need to import and process large-scale data sets in R.

One of the best approaches for working with data from a database is to export the data to a text file and then import the text file into R. According to Adler (2010), importing data into R at a much faster rate from text files than you can from database connections, especially when dealing with very large data sets (1 GB or more). This approach is considered to be the best approach if you plan to import a large amount of data once and then analyse. However, if you need to produce regular reports or to repeat an analysis many times, then it might be better to import data into R directly through a database connection.

There are some packages in order to connect directly to a database from R. The packages you need to install would depend on the database(s) to which you want to connect and the connection method you want to use. There are two sets of database interfaces available in R:

- `RODBC` : The `RODBC` package allows R to fetch data from Open DataBase Connectivity (ODBC) connections. ODBC provides a standard interface for different programs to connect to databases. Before using RODBC, you need to i) install the RODBC package in R, ii) install the ODBC drivers for your platform, iii) configure an ODBC connection to your database. Adler (2010) provides a comprehensive list of where to find ODBC drivers for different databases and operating systems.
- `DBI` : The `DBI` package allows R to connect to databases using native database drivers or JDBC drivers. This package provides a common database abstraction for R software. You must install additional packages to use the native drivers for each particular database(s).

The process of creating a connection is huge and beyond the scope of this course. Here, I will provide a list of additional resources to learn about data importing from these specific databases:

- MySQL: [https://cran.r-project.org/web/packages/RMySQL/index.html (https://cran.r-project.org/web/packages/RMySQL/index.html)](https://cran.r-project.org/web/packages/RMySQL/index.html (https://cran.r-project.org/web/packages/RMySQL/index.html)
- Oracle: https://cran.r-project.org/web/packages/ROracle/index.html (https://cran.r-project.org/web/packages/ROracle/index.html)
- PostgreSQL: https://cran.r-project.org/web/packages/RPostgreSQL/index.html (https://cran.r-project.org/web/packages/RPostgreSQL/index.html)
- SQLite: https://cran.r-project.org/web/packages/RSQLite/index.html (https://cran.r-project.org/web/packages/RSQLite/index.html)
- Open Database Connectivity databases: https://cran.rstudio.com/web/packages/ RODBC (https://cran.rstudio.com/web/packages/RODBC)

Also, R data import/export manual https://cran.r-project.org/doc/manuals/R-data.html (https://cran.r-project.org/doc/manuals/R-data.html) is a comprehensive source for configuring database connections and importing data from databases.

# Scraping Data from Web

As a result of rapid growth of the World Wide Web, vast amount of information is now being stored online, both in structured and unstructured forms. Collecting data from the web is not an easy process as there are many technologies used to distribute web content (i.e., HTML, XML, JSON). Therefore, dealing with more advanced web scraping requires familiarity in accessing data stored in these technologies via R.

In this section, I will provide an introduction to some of the fundamental tools required to perform basic web scraping. This includes importing spreadsheet data files stored online and scraping HTML table data. In order to advance your knowledge in web scraping, I highly recommend getting copies of "XML and Web Technologies for Data Sciences with R" (by Deborah and Ducan (2014)) and "Automated Data Collection with R" (by Munzert et al. (2014)).

## Importing Tabular and Excel files Stored Online

The most basic form of getting data from online is to import tabular (i.e. . txt , .csv) or Excel files that are being hosted online. Importing tabular data is especially common for the many types of government data available online.

To illustrate we will use "Domestic Airlines - On Time Performance" data which is available online at https://data.gov.au/dataset/29128ebd-dbaa-4ff5-8b86-d9f30de56452/resource/cf663ed1-0c5e-497f-aea9-e74bfda9cf44/download/otptimeseriesweb.csv (https://data.gov.au/dataset/29128ebd-dbaa-4ff5-8b86-d9f30de56452/resource/cf663ed1-0c5e-497f-aea9-e74bfda9cf44/download/otptimeseriesweb.csv). This .csv file covers monthly punctuality and reliability data of major domestic and regional airlines operating between Australian airports.

We can use `read.csv` or `read.table` functions to read online data depending upon the format of the data file. In fact, reading online .csv or .txt file is just like reading tabular data. The only difference is, we need to provide the URL of the data instead of the file name as follows:

```
# the url for the online csv file

url <- "https://data.gov.au/dataset/29128ebd-dbaa-4ff5-8b86-d9f30de56452/reso
urce/cf663ed1-0c5e-497f-aea9-e74bfda9cf44/download/otptimeseriesweb.csv"
```

Next, as the online data is a .csv file, we can read this data file using `read.csv` function.

```
# use read.csv to import

ontime_data <- read.csv(url, stringsAsFactors = FALSE)

# display first six rows and four variables in the data

ontime_data[1:6,1:4]
```

```
##                 Route Departing_Port Arriving_Port      Airline
## 1   Adelaide-Brisbane       Adelaide      Brisbane All Airlines
## 2   Adelaide-Canberra       Adelaide      Canberra All Airlines
## 3 Adelaide-Gold Coast       Adelaide    Gold Coast All Airlines
## 4  Adelaide-Melbourne       Adelaide     Melbourne All Airlines
## 5      Adelaide-Perth       Adelaide         Perth All Airlines
## 6     Adelaide-Sydney       Adelaide        Sydney All Airlines
```

Importing Excel spreadsheets hosted online can be performed just as easily. Recall that there is no base R function for importing Excel data; however, several packages exist to import .xls and .xlsx files. One package that works smoothly with pulling Excel data from URLs is `gdata`. With `gdata` we can use `read.xls()` to import Excel files hosted online.

To illustrate, we will use the "Australians' Interest and Engagement with Science" data set available at http://www.data.vic.gov.au/data/dataset/5b9b9dd5-7250-49ef-8c02-dfd12a8c1821/resource/dc730e2d-f6af-47c9-844c-9cd6299ffb8e/download/.filesdbisciencereportsurveydata20120723.xlsx (http://www.data.vic.gov.au/data/dataset/5b9b9dd5-7250-49ef-8c02-dfd12a8c1821/resource/dc730e2d-f6af-47c9-844c-9cd6299ffb8e/download/.filesdbisciencereportsurveydata20120723.xlsx).

First, we need to install and load the `gdata` package, then we can read the online Excel file using `read.xls` function.

```
# first install and load the gdata package

install.packages("gdata")
library(gdata)
```

```
# the url for the online Excel file

url <- "http://www.data.vic.gov.au/data/dataset/5b9b9dd5-7250-49ef-8c02-dfd12
a8c1821/resource/dc730e2d-f6af-47c9-844c-9cd6299ffb8e/download/.filesdbiscien
cereportsurveydata20120723.xlsx"

# use read.xls to import

science_data <- read.xls(url)

# display the first six rows and eight variables in the data

science_data[1:6, 1:8]
```

```
##   ID AGE AGE.BAND AGE.BAND.2 POSTCODE State          SES GENDER
## 1  1  32  25-34yo   Under 35     3131   VIC   Middle SES      F
## 2  2  37  35-44yo        35+     2830   NSW     High SES      M
## 3  3  44  35-44yo        35+     5158    SA   Middle SES      M
## 4  4  30  25-34yo   Under 35     4005   QLD     High SES      F
## 5  5  42  35-44yo        35+       NA  #N/A Unclassified      M
## 6  6  32  25-34yo   Under 35     3128   VIC   Middle SES      F
```

Note that, you will get an error when you try to read from an URL starting with **https** as the `read.xls` function does not support **https**. If you simply replace the **https** with **http** in the url, `read.xls` function will be able to import the file.

## Scraping HTML Table Data

Sometimes, web pages contain several HTML tables and we may want to read the data from that HTML table. The simplest approach to scraping HTML table data directly into R is by using the `rvest` package. Recall that. HTML tables are contained within `<table>` tags; therefore, to extract the tables, we need to use the `html_nodes()` function to select the `<table>` nodes.

To illustrate, I will use the example from the help page for `rvest`, which loads all tables from the U.S. Social Security webpage: https://www.ssa.gov/oact/babynames/numberUSbirths.html (https://www.ssa.gov/oact/babynames/numberUSbirths.html)

First, we will install and load the `rvest` package:

```
# first install and load the rvest package

install.packages("rvest")
library(rvest)
```

We will use `read_html` to locate the URL of the HTML table. When we use `read_html`, all table nodes that exist on the webpage will be captured.

```
births <- read_html("https://www.ssa.gov/oact/babynames/numberUSbirths.html")
```

In this example, using the `length` function we can see that the `html_nodes` captures 2 HTML tables.

```
length(html_nodes(births, "table"))
```

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

This includes data from a few additional tables used to format other parts of the page (i.e. table of contents, table of figures, advertisements, etc.). The second table on the webpage is the place where our data is located, thus, we will select the second element of the `html_nodes`.

```
# select the second element of the html_nodes
births_data<- html_table(html_nodes(births, "table")[[2]])

# view the header of the births_data

head(births_data)
```

```
##   Year ofbirth    Male  Female   Total
## 1         1880 118,400  97,604 216,004
## 2         1881 108,282  98,855 207,137
## 3         1882 122,031 115,695 237,726
## 4         1883 112,477 120,059 232,536
## 5         1884 122,739 137,586 260,325
## 6         1885 115,945 141,949 257,894
```

# Exporting Data

Exporting data out of R is equally important as importing data into R. In this section, we will cover how to export data to text files, Excel files and save to R data objects. In addition to the commonly used base R functions to export data, we will also cover functions from the popular `readr` and `xlsx` packages.

## Exporting Data to text files

Similar to the previous examples provided in the importing text files section, in this section I will introduce the base R and `readr` package functions to export data to text files.

### Base R functions

`write.table()` is the multi-purpose function in base R for exporting data. The function `write.csv()` is a special case of `write.table()` in which the defaults have been adjusted for efficiency. To illustrate, let's create a data frame and export it to a CSV file in our working directory.

```
# create a data frame and assign it to an object named df

df <- data.frame (cost = c(10, 25, 40),
                  color = c ("blue", "red", "green"),
                  suv = c (TRUE, TRUE, FALSE),
                  row.names = c ("car1", "car2", "car3"))

df
```

```
##      cost color   suv
## car1   10  blue  TRUE
## car2   25   red  TRUE
## car3   40 green FALSE
```

To export `df` to a CSV file we will use `write.csv()`.

```
# write to a csv file in our working directory

write.csv(df, file = "cars_csv")
```

If you want to save the data frame in a different directory we will use:

```
# write to a csv and save in a different directory (i.e., ~/Desktop)

write.csv(df, file = "~/Desktop/cars_csv")
```

This function have additional arguments which will allow you to exclude row/column names, specify what to use for missing values, add or remove quotations around character strings, etc.

```
# write to a csv file without row names

write.csv(df, file = "cars_csv", row.names = FALSE)
```

In addition to CSV files, we can also write to other text formats using `write.table()` by specifying the `sep` argument.

```
# write to a tab delimited text file

write.table(df, file = "cars_txt", sep="\t")
```

## The **readr** Package

The `readr` package functions, `write_csv` and `write_delim` are twice as fast as base R functions and they are very similar in usage.

Let's use the same example to illustrate the usage of ` `write_csv` and `write_delim`

```
# load the library

library(readr)

# write to a csv file in the working directory

write_csv(df, path = "cars_csv2")
```

```
# write to a csv and save in a different directory (i.e., ~/Desktop)

write_csv(df, path = "~/Desktop/export_csv2")
```

```
# write to a csv file without column names

write_csv(df, path = "export_csv2", col_names = FALSE)
```

```
# write to a txt file in the working directory

write_delim(df, path = "export_txt2")
```

Note that the base R write functions use the `file =` argument whereas, `readr` write functions use `path =` to specify the name of the file.

## Exporting Data to Excel files

Since we covered importing data with the `xlsx` package, we will use the same package for exporting data to Excel. However, the `readxl` package which I demonstrated in the importing data section does not have a function to export to Excel, therefore I will skip it here.

We will use the `write.xlsx()` function in the `xlsx` package to export the previous example to a xlsx file.

```
# load the library

library(xlsx)
```

```
## Loading required package: rJava
```

```
## Loading required package: xlsxjars
```

```
# write to a .xlsx file in the working directory

write.xlsx(df, file = "cars.xlsx")
```

```
# write to a .xlsx file without row names in the working directory

write.xlsx(df, file = "cars.xlsx", row.names = FALSE)
```

In some cases we may wish to create a .xlsx file that contains multiple data frames. In this case you can just create an empty workbook and save the data frames on separate worksheets within the same workbook. Let's try it using the built in mtcars and iris data sets. First, we will create an empty workbook using createWorkbook() function.

```
# create empty workbook using createWorkbook() function

multiple_df <- createWorkbook()
```

```
# create worksheets within workbook

car_df <- createSheet(wb = multiple_df, sheetName = "Cars")
iris_df <- createSheet(wb = multiple_df, sheetName = "Iris")
```

We will use `addDataFrame()` to add the data frames into the worksheets as follows:

```
# add data frames to worksheets

addDataFrame(x = mtcars, sheet = car_df)

addDataFrame(x = iris, sheet = iris_df)
```

Lastly, we will save it as a .xlsx file using `saveWorkbook()`.

```
# save as a .xlsx file in the working directory

saveWorkbook(multiple_df, file = "combined.xlsx")
```

# Saving Data as an R object File

Sometimes we may need to save data or other R objects outside of the workspace or may want to store, share, or transfer between computers. Basically, we can use the .rda or .RData file types when we want to save several, or all, objects and functions that exist in the global environment. On the other hand, if we only want to save a single R object such as a data frame, function, or statistical model results, it is best to use the .rds file type. Still, we can use .rda or .RData to save a single object but the benefit of .rds is it only saves a representation of the object and not the name whereas .rda and .RData save both the object and its name. As a result, with .rds the saved object can be loaded into a named object within R that is different from the name it had when originally saved.

To illustrate let's create two objects named x and y and save them to a .RData file using `save()` function.

```
# generate random numbers from uniform and normal distribution and assign the
m to objects named x and y, respectively.

x <- runif(10)
y <- rnorm(10, 0, 1)

# Save both objects in .RData format in the working directory

save(x, y, file = "xy.RData")
```

Also, the `save.image()` function will save your all current workspace as .RData.

```
# save all objects in the global environment

save.image()
```

The following examples will illustrate how a single object will be saved using `saveRDS()`

```
# save a single object to file

saveRDS(x, "x.rds")

# restore it under a different name

x2 <- readRDS("x.rds")

# check if x and x2 are identical

identical(x, x2)
```

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

# Additional Resources and Further Reading

R data import/export manual https://cran.r-project.org/doc/manuals/R-data.html (https://cran.r-project.org/doc/manuals/R-data.html) (R Team (2000)) is a comprehensive source for all types of data importing and exporting tasks in R. Also, RStudio's "Data Import Cheatsheet (https://github.com/rstudio/cheatsheets/raw/master/data-import.pdf)" is a compact resource for all importing functions available in the `readr` package.

# References

Adler, Joseph. 2010. *R in a Nutshell: A Desktop Quick Reference*. " O'Reilly Media, Inc."

Boehmke, Bradley C. 2016. *Data Wrangling with R*. Springer.

Deborah, Nolan, and TL Ducan. 2014. "XML and Web Technologies for Data Sciences with R." *N. Deborah, & TL Ducan, XML and Web Technologies for Data Sciences with R*, 581–618.

Munzert, Simon, Christian Rubba, Peter Meißner, and Dominic Nyhuis. 2014. *Automated Data Collection with R: A Practical Guide to Web Scraping and Text Mining*. John Wiley & Sons.

Team, R Core. 2000. "R Data Import/Export."