Beginning with Data in R

Best practices for Importing, Managing and Saving Data Sets

Bruce Dudek

2020-07-30

Contents

1 Introduction

This document is target to researchers who are beginning to learn R usage and students who are learning R in introductory statistics classes. The document has general utility but is specifically targeted for students in B. Dudek's APSY510/511 statistics classes. Parts of it compare methods in R with those used in other software such as SPSS. It is expected that most users of this document are in the early stages of learning R. Therefore, some time is taken to provide additional basic instruction in the R language where it is needed to facilitate the data import process.

For researchers, the most important first steps are learning how to import data, of varying formats (*e.g.*, .txt, .csv, .xlsx, .sav, etc.), into R and saving it. Within R, data can be found in many different kinds of objects such as vectors, matrices, lists, etc. The most common type of data structure used in statistical analysis is a so-called "flat-file" or "rectangular" structure. These are two dimensional arrays where rows are "cases" and columns are "variables". This is the most common structure used for statistical analysis and by commercial software such as SPSS, Stata, Systat and SAS. In R, the parallel is called a data frame. The primary emphasis in this document is import of flat file data sets into R as data frames. Additional needs such as import of relational databases, textual data for text mining, web scraping, etc are mentioned superficially but they are not the primary goal of this document.

Two other flat file data types in R can be seen as enhanced versions of data frames: data tables and tibbles. Some attention to them is also provided in brief sections.

Two major classes of file types containing data are emphasized in this document. They are ascii text files and proprietary binary files created by commercial software. This recognizes that the most common method of bringing data into R is from one of these other file types, such as a .csv file, an Excel spreadsheet or an SPSS/Stata/Systat/SAS system file.

The flow of the document is to outline base R methods first and then outline use of newer methods employing tidyverse functions from **readr**.

In my view the most important parts of this document are those which treat importing data in .csv files. Although much additional background and illustration is included, skipping ahead to those sections is permitted!

One possibility for a learning path is to look at the last section in this document: "Using RStudio to directly import data". This menu-driven approach is a convenient way to quickly import data and learn code at the same time.

1.1 Flat Text Files such as .txt or .csv

Data often exist as flat files in plain ascii text file formats. These may be in a fixed column format or a free field structure as covered in an earlier document. Each row is a case and variables are delimited by spaces, tabs, commas, semi-colons or other indicators. A .txt file most typically has white space as a delimiter (a simple space or a tab). A text file that uses commas as a delimiter is called a .csv file (**C**omma **S**eparated **V**alues). These .csv files are probably the most important file type for moving data around from application to application. So, for example, saving data in SPSS as a .csv file is a useful skill with the "Export" menu operation. Saving an Excel spreadsheet as a .csv file may be the most common way of producing a portable data file.

1.2 Proprietary binary file types such as .xlsx or .sav

Commercial software will save data files with their own special binary format. An Excel .xlsx file will not only contain the data value in the cells but also the Excel Formulas and formatting. In the major commercial statistics packages their formats may include variable attributes such as value and variable labels as well. Moving data from these software types into R can involve exporting the relevant information into a .csv file or we will see that it is possible to import these files directly into R as well. The author has a preference for exporting to the .csv format which is typically the most easily imported into R.

2 The R Environment for this Document

Several packages are required for the work in this document. They are loaded here, but comments/reminders are placed in some code chunks so that it is clear which package some of the functions come from. This next code chunk tests whether the user has the relevant packages already installed and then installs those that aren't. And then succeeding code chunk loads them.

```
# define an objecct that is a list of names of all the packages used in this document
pkg <- c("knitr", "rmarkdown", "foreign", "data.table", "readr", "xlsx", "readxl", "writexl", "haven",

# Check if packages are not installed and create a new object that
# is a list of the names of the packages not installed: new.pkg
new.pkg <- pkg[!(pkg %in% installed.packages())]

# install those that are not already installed
if (length(new.pkg)) {
  install.packages(new.pkg, repos = "http://cran.rstudio.com")
}
```

```
library(knitr)
library(foreign)
library(readr)
library(xlsx)
library(readxl)
library(writexl)
library(haven)
library(dplyr)
```

3 Folders, directories, projects and file locations

For any of the import functions, one has to pass the data file name as an argument so knowing where that file resides is important. At the outset, importing data from a file requires a method for creating directories/folders with the computer system OS that enables efficient organization. Importing data sets requires knowing where such files are saved and orienting R to that location.

3.1 Setting The Working Directory

The recommended way of working in R is to make extensive use of "projects" in RStudio. This automatically sets the working directory to a folder specified when the project was created in RStudio. For some "projects" it is satisfactory to have the data file (*e.g.*, a .csv file) in that folder. Perhaps a subfolder called "data" might also be desireable.

If one is not working in an RStudio "project" then setting the working directory can be done with a menu option: Session - Set Working Directory - Choose directory. Then the files pane will show the files in that folder.

If one is not working in RStudio, but instead in RGui/RConsole, then setting the working directory can also be done. Choose File - Change Dir and then navigagate to the proper directory in the dialog box.

Setting the working directory can always be done with code:

```
setwd("full folder name")
```

Identifying the current working directory can also be done with code:

```
getwd()
```

3.2 Using the `file.choose()` function

All of the import functions have a first argument that is the name of the file to be imported. Often, rather than type in the exact name (which might be forgotten or in another folder than the working folder) it is possible to have R generate a dialog box permitting mouse-based choice of the file name and folder. Examples below make regular use of this function.

4 Import text files such as .txt or .csv using Base R methods

The illustrations here provide the basics for reading ascii text files. The `read.csv` and `read.delim` functions are variants of the more general `read.table` function. With `read.table`, we can read any ascii text file and specify the exact characteristics. There are several core characteristics that need to be defined and they comprise a list of arguments to be passed to `read.table`:

- file: name of the file to be imported
- header: does the first line of the data file contain variable names? default is FALSE
- sep: what is the separator between values? default is whitespace (includes spaces and tabs)
- dec: defining character for decimal place. default is a period
- encoding: defines encoding method for ascii text. default is Latin-1 or UTF-8

With these defaults, `read.table` expects a file without a header line of variable names, spaces or tabs are used as the delimiter, a period for a decimal place, and usually UTF-8 encoding (see later section on encoding)

4.1 Using `read.table`

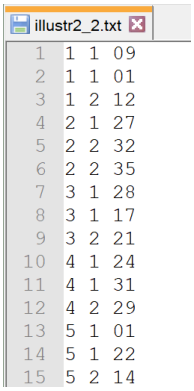
The `read.table()` family (including `read.csv` and `read.delim`) of functions is found in the **utils** package installed with base R.

These first two templates in this initial code chunk use the defaults defined above and assume that the data file is available in the default directory (first commented line of code) or somewhere on the local machine (for the `file.choose` usage of the second line of code).

For each instance, the results of the `read.table` function (and later functions in other sections) are a data frame that is created as an object with the name of "df1". The assignment operator (`<-`) is used to define the new data frame object as the chosen name (df1 here) containing the results of the `read.table` operation.

```
#df1 <- read.table("datafile1.txt")
df1 <- read.table(file.choose())
```

Here is an example of a .txt file containing 14 cases and 3 variables, in the three columns. There is no header row containing variable names. The file is a fixed format file where the three variables reside in exactly the same columns for each case, although a freee field format is also readable this way as long as the delimiter is present. White spaces between variables are simply spaces and delimit the variables. This figure shows what the plain text file looks like in a text editor. The first variable is a numeric code for religion, the second is a numeric code for gender and the third is a measured variable hypothetically defined as church attendances in the past year; each row is a separate individual or case.



The family of `read.table` functions can also open a data file specified by a URL as the "filename" argument:

```
df1 <- read.table("https://bcdudek.net/datasets/illustr2_2.txt")
df1
```

```
##    V1 V2 V3
## 1   1  1  9
## 2   1  1  1
## 3   1  2 12
## 4   2  1 27
## 5   2  2 32
## 6   2  2 35
## 7   3  1 28
## 8   3  1 17
## 9   3  2 21
## 10  4  1 24
## 11  4  1 31
## 12  4  2 29
## 13  5  1  1
## 14  5  1 22
## 15  5  2 14
```

Since no column heading names existed in that data file, we would create them in R to name the variables.

```
colnames(df1) <- c("religion", "gender", "attend")
df1
```

```
##    religion gender attend
## 1         1      1      9
## 2         1      1      1
## 3         1      2     12
## 4         2      1     27
## 5         2      2     32
## 6         2      2     35
## 7         3      1     28
## 8         3      1     17
## 9         3      2     21
## 10        4      1     24
## 11        4      1     31
## 12        4      2     29
## 13        5      1      1
## 14        5      1     22
## 15        5      2     14
```

`read.table` should be thought of as the core/parent function in a family of ascii text reading functions. All arguments/capabilities are available in the "children" versions of `read.table` outlined next.

4.2 Using `read.csv`

The `read.csv` function can be thought of as `read.table` with `sep=","` and `header=TRUE`. It is probably the most commonly used data import technique in R. Once again, a file name or `file.choose` can be specified and it also accommodates URL's as shown above.

```
#df2 <- read.csv("datfile1.csv")
df2 <- read.csv(file.choose())
```

An example is the same data set as used above, but contained in a .csv file. One change is that the gender variable is written as a string variable instead of as a numeric code. The .csv file looks like this:

illustr2_5b.csv

```
religion,gender,attend
1,female,9
1,female,1
1,male,12
2,female,27
2,male,32
2,male,35
3,female,28
3,female,17
3,male,21
4,female,24
4,female,31
4,male,29
5,female,1
5,female,22
5,male,14
```

Here is code reading that file at an internet location:

```
df3 <- read.csv("https://bcdudek.net/datasets/illustr2_5b.csv")
str(df3)
```

```
## 'data.frame':    15 obs. of  3 variables:
##  $ religion: int  1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : chr  "female" "female" "male" "female" ...
##  $ attend  : int  9 1 12 27 32 35 28 17 21 24 ...
```

```
df3
```

```
##    religion gender attend
```

```
## 1         1 female      9
## 2         1 female      1
## 3         1   male     12
## 4         2 female     27
## 5         2   male     32
## 6         2   male     35
## 7         3 female     28
## 8         3 female     17
## 9         3   male     21
## 10        4 female     24
## 11        4 female     31
## 12        4   male     29
## 13        5 female      1
## 14        5 female     22
## 15        5   male     14
```

Notice that the structure (from the `str` function) of the data frame indicates the "class" of gender is "chr", meaning character. Attend is "int", meaning integer both of which make sense. Historically, prior to R version 4.0, `read.csv` produced a class for string variables, such as gender, that is called a factor. For many analyses that users of this document this is desirable when those variables are used in analyses such as ANOVA or t-tests, or regression. There is an argument for `read.csv` called `stringsAsFactors` that controls this. Previously it was set to TRUE by default. Since R version 4.0, the default was changed to FALSE. If we re-read the data file with that argument set to TRUE, then gender will be a factor.

```
df3a <- read.csv("https://bcdudek.net/datasets/illustr2_5b.csv", stringsAsFactors=TRUE)
str(df3a)
```

```
## 'data.frame':    15 obs. of  3 variables:
##  $ religion: int  1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : Factor w/ 2 levels "female","male": 1 1 2 1 2 2 1 1 2 1 ...
##  $ attend  : int  9 1 12 27 32 35 28 17 21 24 ...
```

4.3 using `read.delim`

We can think of `read.delim` as a `read.table` function with `header=TRUE` and the separator is a tab (`sep="\t"`). These would be the defaults forread.delimand offer a quicker way to write the code than to useread.table‘ specifying the arguments.

```
#df4 <- read.delim("datafile1.txt")
df4 <- read.delim(file.choose())
```

4.4 Using `read.csv2` and `read.delim2`

Two other variations are also available for special situations and can speed coding.

`read.csv2` changes the delimiter to a semi-colon, and changes the decimal to a comma and is thus like using `read.table` this way:

```
df5 <- read.table("filename", header=TRUE, sep=";", dec=",")
```

`read.delim2`retains the tab as delimiter but also expects a header and specifies the decimal as a comma. It is the same as using `read.table` this way:

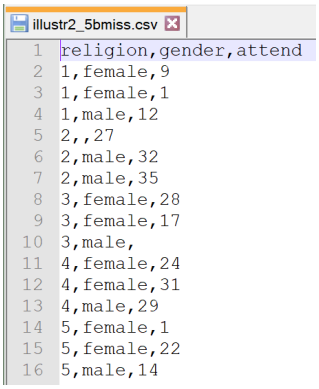
```
df6 <- read.table("filename", header=TRUE, sep="\t", dec=",")
```

These two functions quickly enable standard practices for delimited text files in some countries or specialized fields.

4.5 Handling missing data in reading .csv files

The `read.table`, `read.csv`, and `read.delim` functions have an argument called `na.strings`. Best practice would be to have missing data values in the .csv files converted to a value of "NA". The "NA" is interpreted as missing. So for example if a .csv file has a missing value, then that should result in two successive commas (or a blank field after the last comma if the last varaible's data is missing) for that record in the location of the missing value. Once imported into a data frame by `read.csv`, we would like the missing value to become a NA.

Here is a screen capture of an example where one value for gender was deleted, in the fourth case, and one value for attend was missing in the 9th case. It is the same data file used above to produce df3 and df3a.



Now `read.csv` is used to import that file with the addition of an argument that indicates that empty fields, indicated by the two successive quotation marks, will be read as missing and an NA inserted.

```
df3miss <- read.csv("illustr2_5bmiss.csv",
                    stringsAsFactors = TRUE,
                    na.strings = c(""))
df3miss
```

```
##    religion gender attend
## 1         1 female      9
## 2         1 female      1
## 3         1   male     12
## 4         2   <NA>     27
## 5         2   male     32
## 6         2   male     35
## 7         3 female     28
## 8         3 female     17
## 9         3   male     NA
## 10        4 female     24
## 11        4 female     31
## 12        4   male     29
## 13        5 female      1
## 14        5 female     22
## 15        5   male     14
```

One final clarification is needed here. In the rmarkdown-generated display of the table in this document, the missing values for both gender and attend are shown slightly differently. For gender the missing value is "" and for attend it is "NA". This is normal. The "" value is used for string/character/factor variables and the "NA" is used for numeric variables such as attend. Both values are treated the same in analyses.

4.6 Additional Detailed information on the `read.` family of functions

The rudimentary introduction to `read.table`, `read.csv` and `read.delim` found here provides a good start that will enable most users to import most of the types of data files they will encounter. But this family of functions is capable of handling many more types of situations. Many sources on the web can be found, but one very good starting place is the document on the R Project.org site:

https://cran.r-project.org/doc/manuals/r-release/R-data.pdf

5 Import text files such as .txt or .csv using Tidyverse methods

Developers at the RStudio group have created a suite of R packages that work well together and with the whole R ecosystem. It is called the tidyverse. Included in this suite of packages are many functions that serve as replacements for the base system approach to things. Data import and data management (called data wrangling in the data science world) is one of the emphases. Here, we will look at functions that do the work that we have just covered with the `read.table` family of data import functions.

These functions come from the **readr** package. One distinction from the `read.table` family of functions is that the **readr** package functions produce a variation on the data frame called a tibble. Tibbles approach some challenging issues in use of `read.table` with improved efficiency, and provide some additional capability, provide a better foundation for use of tibbles by other tidyverse packages, and are a bit easier to use. One major advantage of tibbles is purported to be its relative speed at importing large files. There are some downsides however (some older packages don't work with tibbles), so one of the illustrations below shows how to convert a tibble to a data frame if that need is present.

Several blogs/websites can provide more detail on the question of tibbles vs. data frames and can give more detailed expositon on use of the **readr** functions than is provided here:

- **readr** package description
- CRAN page on tibbles
- R for Data Science chapter on tibbles
- The trouble with tibbles

5.1 using `read_table` and `read_csv` for ascii text files

The **readr** package has at least seven functions for varying file formats:

- `read_csv()`: comma separated (CSV) files (also `read_csv2`)
- `read_tsv()`: tab separated files
- `read_delim()`: general delimited files
- `read_fwf()`: fixed width files
- `read_table()`: tabular files white-space separated (also `read_table2`)
- `read_log()`: web log files
- `read_file()`: complete files

We will use `read_table` and `read_csv` here. Note how the function names mirror the base system `read.table` and `read.csv` names, using the underscore rather than the dot/period. Their capabilities and defaults are similar.

The first illustration mirrors the first illustration done above with `read.table` and uses the same data file. `read_table` is used when whitespaces are the delimiter. The `col_names` argument is an expanded version of the `header` argument found in `read.table`. Since our data file did not have a header line containing variable names, the argument is set to FALSE here and variable names are automatically generated. By default, `read_table` has `col_names` set to true, and would expect a header line, which we don't have

```
df7 <- read_table("https://bcdudek.net/datasets/illustr2_2.txt",
                  col_names=FALSE)
```

```
## Parsed with column specification:
## cols(
##   X1 = col_double(),
##   X2 = col_double(),
##   X3 = col_character()
## )
```

```
df7
```

```
## # A tibble: 15 x 3
##       X1    X2 X3
##    <dbl> <dbl> <chr>
##  1     1     1 09
##  2     1     1 01
##  3     1     2 12
##  4     2     1 27
##  5     2     2 32
##  6     2     2 35
##  7     3     1 28
##  8     3     1 17
##  9     3     2 21
## 10     4     1 24
## 11     4     1 31
## 12     4     2 29
## 13     5     1 01
## 14     5     1 22
## 15     5     2 14
```

One nice feature of `read_table` is that an additional usage of `col_names` can be employed. If a string of variable names is passed, those variable names will be applied to the variables within the `read_table` operation and a second line of code using `colnames` as was done above is not necessary.

```
df7b <- read_table("https://bcdudek.net/datasets/illustr2_2.txt",
                  col_names= c("religion", "gender", "attend"))
```

```
## Parsed with column specification:
## cols(
##   religion = col_double(),
##   gender = col_double(),
##   attend = col_character()
## )
```

```
df7b
```

```
## # A tibble: 15 x 3
##    religion gender attend
##       <dbl>  <dbl> <chr>
##  1        1      1 09
##  2        1      1 01
##  3        1      2 12
##  4        2      1 27
##  5        2      2 32
##  6        2      2 35
##  7        3      1 28
```

```
##  8        3      1 17
##  9        3      2 21
## 10        4      1 24
## 11        4      1 31
## 12        4      2 29
## 13        5      1 01
## 14        5      1 22
## 15        5      2 14
```

Similary, `read_csv` handles .csv files with headers and comma delimitation quite directly:

```
df8 <- read_csv("https://bcdudek.net/datasets/illustr2_5b.csv")
```

```
## Parsed with column specification:
## cols(
##   religion = col_double(),
##   gender = col_character(),
##   attend = col_double()
## )
```

```
str(df8)
```

```
## tibble [15 x 3] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ religion: num [1:15] 1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : chr [1:15] "female" "female" "male" "female" ...
##  $ attend  : num [1:15] 9 1 12 27 32 35 28 17 21 24 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   religion = col_double(),
##   ..   gender = col_character(),
##   ..   attend = col_double()
##   .. )
```

```
df8
```

```
## # A tibble: 15 x 3
##    religion gender attend
##       <dbl> <chr>   <dbl>
##  1        1 female      9
##  2        1 female      1
##  3        1 male       12
##  4        2 female     27
##  5        2 male       32
##  6        2 male       35
##  7        3 female     28
##  8        3 female     17
##  9        3 male       21
## 10        4 female     24
## 11        4 female     31
## 12        4 male       29
## 13        5 female      1
## 14        5 female     22
## 15        5 male       14
```

Notice how the printed version of these two tibbles contain definitional information about the type of variable that each was determined to be. Note that gender is a character vector, not a factor. The following code chunk shows a way to convert those character variables to factors with the `mutate_if` function from **dplyr**.

```
df8b <- read_csv("https://bcdudek.net/datasets/illustr2_5b.csv")
```

```
## Parsed with column specification:
## cols(
##   religion = col_double(),
##   gender = col_character(),
##   attend = col_double()
## )
```

```
df8b <- mutate_if(df8b, is.character, factor)
df8b
```

```
## # A tibble: 15 x 3
##    religion gender attend
##       <dbl> <fct>   <dbl>
##  1        1 female      9
##  2        1 female      1
##  3        1 male       12
##  4        2 female     27
##  5        2 male       32
##  6        2 male       35
##  7        3 female     28
##  8        3 female     17
##  9        3 male       21
## 10        4 female     24
## 11        4 female     31
## 12        4 male       29
## 13        5 female      1
## 14        5 female     22
## 15        5 male       14
```

6 Import Commercial Statistical Software Data Files using foreign

Data from the major commercial statistical analysis packages (*e.g.*, SPSS, Stata, and SAS, and Systat can all be imported into SPSS and there are only a few ways of doing this. One indirect way to import data from these packages is to first save it from the application as a .csv file (see a section below on doing this in SPSS). Alternatively it is possible to read the proprietary binary files (*e.g.*, .sav, .sas7bdat, .dta, .sysdat .The **foreign** package has been providing this capability for quite a few years for SPSS, Stata, Minitab, Epi Info, and Octave. The **sas7bdat** package can be used to import sas7 files. Exported XPORT files from SAS can be read with a function from the **HMISC** package An example with SPSS is provided here, and templates for a few others. Users are urged to read more detailed documents.

- DataCamp
- **foreign** package
- **sas7bdat** package
- `sasxport.get` from **HMISC**

6.1 Importing data from SPSS

One can export data from SPSS into .csv files and then use some of the previously described methods to read those .csv files. This is a recommended method and a section below provides more guidance. However, it is possible to read the SPSS .sav files directly. In this section, the `read.spss` function from **foreign** is employed. One caution is that in the `read.spss` documentation it indicates that **foreign** was written for

earlier versions of SPSS and has not looked to incorporate all changes in later versions of .sav files. It does say that few changes have occured in .sav files in recent versions. I have found `read.spss` to be very capable, but if issues arise, the tidyverse package called **haven** is a good alternative (outlined in a later section).

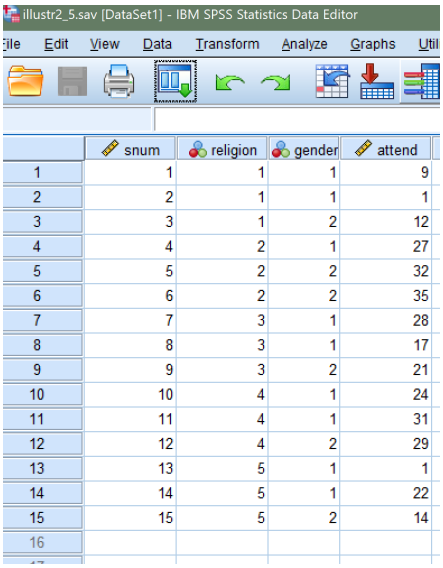
Several considerations come in to play in reading .sav files. The two most important are handling of numerically coded categorical variables and their value labels and the handling of missing data. The `read.spss` function provides facility for choices in both of these topics.

A large number of `read.spss` arguments give the user detailed control over various aspects of the import. The reader is encouranged to examine the documentation for the `read.spss` function:

`read.spss` documentation

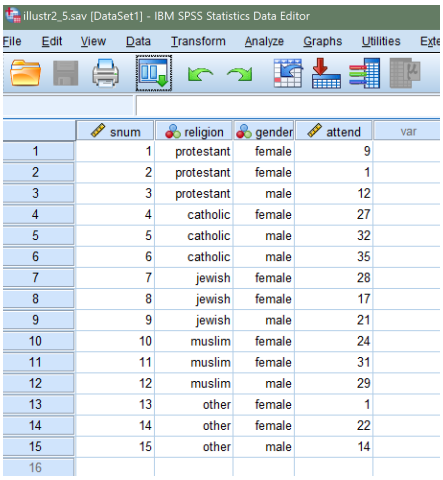
Most of the defaults for `read.spss` arguments are usable, but some attention should be given to a few. The first question to be addressed is how to handle variables where categorical variables are numerically coded. If those variables also have value label definitions in SPSS then the value labels can be created when the import is done and either serve as the data values in the R data frame or as attributes associated with the numerical values of that variable. Note that an issue arises if not all numerical codes for a variable have a value label (please read the documentation on this).

For example here is an SPSS screen shot of the same data set used above, with religion and gender coded numerically:



	snum	religion	gender	attend
1	1	1	1	9
2	2	1	1	1
3	3	1	2	12
4	4	2	1	27
5	5	2	2	32
6	6	2	2	35
7	7	3	1	28
8	8	3	1	17
9	9	3	2	21
10	10	4	1	24
11	11	4	1	31
12	12	4	2	29
13	13	5	1	1
14	14	5	1	22
15	15	5	2	14
16				

Here is a screen shot of the same data set showing the value labels for those variables:



	snum	religion	gender	attend	var
1	1	protestant	female	9	
2	2	protestant	female	1	
3	3	protestant	male	12	
4	4	catholic	female	27	
5	5	catholic	male	32	
6	6	catholic	male	35	
7	7	jewish	female	28	
8	8	jewish	female	17	
9	9	jewish	male	21	
10	10	muslim	female	24	
11	11	muslim	female	31	
12	12	muslim	male	29	
13	13	other	female	1	
14	14	other	female	22	
15	15	other	male	14	
16					

A second issue to address is that With its default settings, `read.spss` does not create a data frame. We would typically want it to do so, so an argument must be set to accomplish this. Note that in this default setup, the religion and gender variables are read as factors rather than numerics. This is because another argument, use.value.labels is set to TRUE by default. Most typically, this approach to use the string values of the value labels is what would be preferred. In most analytical applications in traditional statistics, treating the categorical variables as factors would be preferred. If not, then they can be converted to character types after the import (with the `as.character` function), or imported as numerics in the manner shown in the succeeding code chunk.

```
df9 <- read.spss("illustr2_5.sav", to.data.frame=TRUE)
```

```
## re-encoding from UTF-8
```

```
str(df9)
```

```
## 'data.frame':    15 obs. of  4 variables:
##  $ snum    : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ religion: Factor w/ 5 levels "protestant","catholic",..: 1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : Factor w/ 2 levels "female","male": 1 1 2 1 2 2 1 1 2 1 ...
##  $ attend  : num  9 1 12 27 32 35 28 17 21 24 ...
##  - attr(*, "variable.labels")= Named chr(0)
##   ..- attr(*, "names")= chr(0)
##  - attr(*, "codepage")= int 65001
```

```
df9
```

```
##    snum   religion gender attend
## 1     1 protestant female      9
## 2     2 protestant female      1
## 3     3 protestant   male     12
## 4     4   catholic female     27
## 5     5   catholic   male     32
## 6     6   catholic   male     35
## 7     7     jewish female     28
## 8     8     jewish female     17
## 9     9     jewish   male     21
## 10   10     muslim female     24
## 11   11     muslim female     31
## 12   12     muslim   male     29
## 13   13      other female      1
## 14   14      other female     22
```

```
## 15   15      other   male     14
```

If we want to read the religion and gender variables as their numeric values then we change the `to.data.frame` argument:

```
df10 <- read.spss("illustr2_5.sav", to.data.frame=TRUE, use.value.labels=FALSE)
```

```
## re-encoding from UTF-8
```

```
str(df10)
```

```
## 'data.frame':    15 obs. of  4 variables:
##  $ snum    : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ religion: num  1 1 1 2 2 2 3 3 3 4 ...
##   ..- attr(*, "value.labels")= Named chr [1:5] "5" "4" "3" "2" ...
##   .. ..- attr(*, "names")= chr [1:5] "other" "muslim" "jewish" "catholic" ...
##  $ gender  : num  1 1 2 1 2 2 1 1 2 1 ...
##   ..- attr(*, "value.labels")= Named chr [1:2] "2" "1"
##   .. ..- attr(*, "names")= chr [1:2] "male" "female"
##  $ attend  : num  9 1 12 27 32 35 28 17 21 24 ...
##  - attr(*, "variable.labels")= Named chr(0)
##   ..- attr(*, "names")= chr(0)
##  - attr(*, "codepage")= int 65001
```

```
df10
```

```
##    snum religion gender attend
## 1     1        1      1      9
## 2     2        1      1      1
## 3     3        1      2     12
## 4     4        2      1     27
## 5     5        2      2     32
## 6     6        2      2     35
## 7     7        3      1     28
## 8     8        3      1     17
## 9     9        3      2     21
## 10   10        4      1     24
## 11   11        4      1     31
## 12   12        4      2     29
## 13   13        5      1      1
## 14   14        5      1     22
## 15   15        5      2     14
```

Another issue to be considered is the handling of missing data. In R, missing values should be written as "NA". By default, `read.spss` converts any SPSS-defined missing values to these NA quantities. The user should be careful to doublecheck that what is expected is what was produced when there are missing values.

6.2 Templates for importing from SAS, Stata, Systat, Minitab

Stata .dta files are also readable by a **foreign** function. Reading the helpfile for information on arguments passed to the function is important. For example, by default convert.factors is set to TRUE.

```
df11 <- read.dta("filename")
```

Two other packages also have functions for handling Stata files, but are not illustrated here. See the **haven** package discussed below and the `Stata.file` function in the **memisc** package.

Systat .syd files can also be read with a **foreign* function. Once again, reading the help file is important.

```
df12 <- read.systat("filename")
```

SAS data files of the .sas7bdat format can be read with a package of that name **sas7bdat**.

```
df13 <- sas7bdat("filename.sas7bdat")
```

Two additional functions in the **foreign** package will read SAS Permanent Datasets or SAS XPORT Formatted libraries, respectively.

```
df14 <- read.ssd("filename.ssd")
df15 <- read.xport("filename.xport")
```

7 Commercial Statistical Software Data Files (SPSS, STATA, SAS) using Tidyverse methods

The recently developed **haven** package from the tidyverse is a robust suite of tools for importing data sets from commercial packages. The main data import functions in **haven** are:

- 'read_sas()': reads .sas7bdat and .sas7bcat files from SAS
- 'read_xpt()': reads from SAS Transport files
- 'read_sav()': reads .sav files from SPSS
- 'read_por()': reads .por Portable file exports from SPSS
- 'read_dta()': reads .dta files from Stata

As was the case with the **readr** and **readxl** package functions described above, these functions produce tibbles and the user may want to convert to a data as I've shown in this first SPSS example.

7.1 Read SPSS .sav file with `read_sav` or `read_spss`

Using the same SPSS data file illustration as shown above with `read.spss`:

```
df16 <- read_sav("illustr2_5.sav")
str(df16)
```

```
## tibble [15 x 4] (S3: tbl_df/tbl/data.frame)
##  $ snum    : num [1:15] 1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "format.spss")= chr "F8.0"
##  $ religion: dbl+lbl [1:15] 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5
##    ..@ format.spss: chr "F8.0"
##    ..@ labels     : Named num [1:5] 1 2 3 4 5
##    .. ..- attr(*, "names")= chr [1:5] "protestant" "catholic" "jewish" "muslim" ...
##  $ gender  : dbl+lbl [1:15] 1, 1, 2, 1, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 2
##    ..@ format.spss  : chr "F6.0"
##    ..@ display_width: int 6
##    ..@ labels       : Named num [1:2] 1 2
##    .. ..- attr(*, "names")= chr [1:2] "female" "male"
##  $ attend  : num [1:15] 9 1 12 27 32 35 28 17 21 24 ...
##   ..- attr(*, "format.spss")= chr "F8.0"
```

```
df16
```

```
## # A tibble: 15 x 4
##     snum       religion     gender attend
##    <dbl>      <dbl+lbl>  <dbl+lbl>  <dbl>
##  1     1 1 [protestant] 1 [female]      9
##  2     2 1 [protestant] 1 [female]      1
##  3     3 1 [protestant] 2 [male]       12
```

```
##  4     4 2 [catholic]   1 [female]     27
##  5     5 2 [catholic]   2 [male]       32
##  6     6 2 [catholic]   2 [male]       35
##  7     7 3 [jewish]     1 [female]     28
##  8     8 3 [jewish]     1 [female]     17
##  9     9 3 [jewish]     2 [male]       21
## 10    10 4 [muslim]     1 [female]     24
## 11    11 4 [muslim]     1 [female]     31
## 12    12 4 [muslim]     2 [male]       29
## 13    13 5 [other]      1 [female]      1
## 14    14 5 [other]      1 [female]     22
## 15    15 5 [other]      2 [male]       14
```

Also note that in reading this same SPSS .sav as we did above with `read.spss`, that here the religion and gender variables are numeric values, but they are indicated to have labels associated with them. This means that the tibble is structured much like the SPSS system file in that the numeric codes are the core value but value labels are retained (called attribute names).

If we convert the tibble to a data frame by nesting the `as.data.frame` function around the `read_sav` function you will see that the attribute names are retained - they are not a tibble property but a general data frame property. The nice feature is that `read_sav` implements them directly, unlike `read.spss` as we saw above.

```
df17 <- as.data.frame(read_sav("illustr2_5.sav"))
str(df17)
```

```
## 'data.frame':    15 obs. of  4 variables:
##  $ snum    : num  1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "format.spss")= chr "F8.0"
##  $ religion: dbl+lbl [1:15] 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5
##    ..@ format.spss: chr "F8.0"
##    ..@ labels     : Named num  1 2 3 4 5
##    .. ..- attr(*, "names")= chr [1:5] "protestant" "catholic" "jewish" "muslim" ...
##  $ gender  : dbl+lbl [1:15] 1, 1, 2, 1, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 2
##    ..@ format.spss  : chr "F6.0"
##    ..@ display_width: int 6
##    ..@ labels       : Named num  1 2
##    .. ..- attr(*, "names")= chr [1:2] "female" "male"
##  $ attend  : num  9 1 12 27 32 35 28 17 21 24 ...
##   ..- attr(*, "format.spss")= chr "F8.0"
```

```
df17
```

```
##    snum religion gender attend
## 1     1        1      1      9
## 2     2        1      1      1
## 3     3        1      2     12
## 4     4        2      1     27
## 5     5        2      2     32
## 6     6        2      2     35
## 7     7        3      1     28
## 8     8        3      1     17
## 9     9        3      2     21
## 10   10        4      1     24
## 11   11        4      1     31
## 12   12        4      2     29
## 13   13        5      1      1
## 14   14        5      1     22
```

```
## 15   15        5      2     14
```

I have not found a way to read the value labels as the core values for categorical variables with `read_spss` (e.g., religion and gender here). If one needs to do this, then `read.spss` may be a better choice. This is an important issue because using a categorical variable would typically be used as a factor in analyses such as IVs in regression or in ANOVAs.

Reading .sav files with `read_sav` will apparently only produce a tibble with those variables as numerics and they would have to be converted to factors afterwards, even if the conversion to a data frame is done as with df17 here. So, this code chunk does the conversions and then uses the variables in a regression. We see in the results (fit2) that the five level religion factor has four indicator variables listed, and this is the correct analysis. Fit1 (commented out before the conversion) would be an improper analysis that used religion as a single numeric variable.

```
#fit1 <- lm(attend ~ religion + gender, data=df15)
df17$religion <- as.factor(df17$religion)
df17$gender <- as.factor(df17$gender)
fit2 <- lm(attend ~ religion + gender, data=df17)
summary(fit2)
```

```
##
## Call:
## lm(formula = attend ~ religion + gender, data = df17)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -10.2667  -3.0333  -0.4667   2.6667  10.7333
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    6.267      3.913   1.601  0.14375
## religion2     22.933      5.407   4.242  0.00217 **
## religion3     14.667      5.277   2.780  0.02141 *
## religion4     20.667      5.277   3.917  0.00353 **
## religion5      5.000      5.277   0.948  0.36809
## gender2        3.200      3.540   0.904  0.38953
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.462 on 9 degrees of freedom
## Multiple R-squared:  0.7731, Adjusted R-squared:  0.647
## F-statistic: 6.132 on 5 and 9 DF,  p-value: 0.00961
```

7.2 haven templates for SAS and Stata

The help pages for these functions are important to read.

First, read SAS .sas7bdat or .sas7cdat files:

```
df18 <- read_sas("filename.sas7bdat")
```

Next, read Stata .dta files:

```
df18 <- read_stata("filename")
```

7.2.1 Comments on the haven functions

Please note that there is much more to the capabilities of these **haven** functions than is illustrated here. And there is additional flexibility provided by the production of tibbles. However, the limitation noted about creation of factors when categorical variables are read is a major drawback for the author, not offset by the ease of use, faster performance and added flexibility of tibble creation. There are ways of converting the value labels to actual values after the fact, but that is beyond the scope of this document. I would still prefer the **foreign** functions.

More information on the **haven** functions can be found here:

- Scheidel Tutorials

8 Import Excel Spreadsheet Data

There are many methods of importing data from Excel spreadsheets. We will illustrate two in particular, outline two others, and suggest additional packages to examine.

8.1 Copy/Paste from the Excel spreadsheet

One quick and direct method, albeit less reproducible, is to highlight an array of values in the Excel spreadsheet and copy. Then the following use of `read.table` can create a data frame. The reader should also note that the copy could be done from other applications such as an ascii text editor (e.g., notepad++). The primary downside of this method is that it requires another app to be installed and open.

```
df19 = read.table("clipboard")
```

8.2 Save Excel files to .csv format

An indirect method is to save the relevant Excel worsheet/array to a .csv file and then use one of the previous methods for reading .csv files. This creates an extra step and extra file, but .csv files are easy to work with.

8.3 Using the xlsx package

When reading data from a spreadsheet such as Excel is done, the user may have to specify which worksheet contains the data to be read - unless the spreadsheet only has one worksheet. Worksheets can be identified by number or name.

The `read.xlsx` function from **xlsx** handles Excel spreadsheets well, and fairly accurately determines the class of each variable. The example shown here, uses the same data set as above, but in Excel format. The sheet is defined as the first worksheet in the file.

```
df20 <- xlsx::read.xlsx("illustr2_5b.xlsx",
                sheetName=1)
df20
```

```
##    religion gender attend
## 1         1 female      9
## 2         1 female      1
## 3         1   male     12
## 4         2 female     27
## 5         2   male     32
## 6         2   male     35
## 7         3 female     28
## 8         3 female     17
## 9         3   male     21
## 10        4 female     24
```

```
## 11        4 female     31
## 12        4   male     29
## 13        5 female      1
## 14        5 female     22
## 15        5   male     14
```

Here, using tidyverse methods from the **readxl** package, the worksheet is defined by its name rather than its numeric position as we illustrated in the prior example.

```
df21 <- readxl::read_excel("illustr2_5b.xlsx",
               sheet="illustr2_5b")
df21
```

```
## # A tibble: 15 x 3
##    religion gender attend
##       <dbl> <chr>   <dbl>
##  1        1 female      9
##  2        1 female      1
##  3        1 male       12
##  4        2 female     27
##  5        2 male       32
##  6        2 male       35
##  7        3 female     28
##  8        3 female     17
##  9        3 male       21
## 10        4 female     24
## 11        4 female     31
## 12        4 male       29
## 13        5 female      1
## 14        5 female     22
## 15        5 male       14
```

Both of these latter methods work well in most circumstances, but remember that the tidyverse functions produce a tibble (df21), but `read.xlsx` produces a data frame (df20). Verify that asssertion with this code:

```
str(df20)
```

```
## 'data.frame':    15 obs. of  3 variables:
##  $ religion: num  1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : chr  "female" "female" "male" "female" ...
##  $ attend  : num  9 1 12 27 32 35 28 17 21 24 ...
```

```
str(df21)
```

```
## tibble [15 x 3] (S3: tbl_df/tbl/data.frame)
##  $ religion: num [1:15] 1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : chr [1:15] "female" "female" "male" "female" ...
##  $ attend  : num [1:15] 9 1 12 27 32 35 28 17 21 24 ...
```

Two other packages are also commonly used for Excel data file import, but will only be listed here: **XLConnect** and **openxlsx**.

Additional Details are available in many resources:

- STHDA tutorial
- Tidyverse **readxl** package documentation
- Stats and R Blog

- Milano R
- Datacamp

9 Converting tibbles to data frames

It is worthwhile to be redundant on this topic with what was shown in earlier code to reinforce the simplicity of the conversion.

A simple method exists for conversion of tibbles to data frames if the analysis requires a data frame. Recall that df21 above was created by `read_excel` which produces a tibble. We can look at the "structure" of the tibble (df21) and verify that it is a tibble. Then we can convert it to a data frame (new object called df21b) and verify by looking at the "structure":

```
str(df21)
```

```
## tibble [15 x 3] (S3: tbl_df/tbl/data.frame)
##  $ religion: num [1:15] 1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : chr [1:15] "female" "female" "male" "female" ...
##  $ attend  : num [1:15] 9 1 12 27 32 35 28 17 21 24 ...
```

```
df21b <- as.data.frame(df21)
str(df21b)
```

```
## 'data.frame':    15 obs. of  3 variables:
##  $ religion: num  1 1 1 2 2 2 3 3 3 4 ...
##  $ gender  : chr  "female" "female" "male" "female" ...
##  $ attend  : num  9 1 12 27 32 35 28 17 21 24 ...
```

10 Data Tables

In addition to data frames and tibbles, another data structure used in data science is managed by the **data.table** package. There read and write capabilities to work with data tables in this package and it provides a robust suite of alternatives to base and tidyverse functions for datamanagement. The biggest advantage of using data tables is the speed of import and procesing during data wrangling procedures. I don't see it used much in psychological research circles, so no examples or exposition are included here. However, a few links to online resources are an entry point into learning about it. Its users are enthusiastic about its capabilities.

- An Introdution to **data.table**
- Beginner's guide to **data.table**
- **data.table** and **magrittr** pipes, the best of both worlds
- Data Table Tutorial
- DataCamp: The **data.table** R Package Cheat Sheet

11 Importing data from Databases

Skills for importing data from databases have become critical in the broader Data Science world. In the scientific research world it is a less pressing skill and so will not be illustrated in detail here. Instead, just methods are described.

11.1 Importing Relational Databases

A few tutorials and other resources are recommended for importing from MonetDB, SQL and other relational databases:

- CRAN Task Views on Databases with R
- ProjectPro Tutorial
- RStudio tutorial on MySQL connections from R
- Tutorial on using MySQL databases with R

11.2 Importing from non-relational data bases

A few worthwhile sites:

- CRAN Task Views on Databases with R
- ProjectPro Tutorial
- Reading MonetDB databases with **dplyr**
- R and MongoDB

12 Importing data from tables in web pages (HTML tables)

R can import data that are found in HTML tables on web pages. The **XML** package has a function called `readHTMLTable` that appears to be simple to use. But extracting exactly what you want after obtaining the table may require some work. Here is a detailed explanation in a question posed on StatOverflow:

- StackOverflow Example with NFL Fantasy Football

A second method is argued to be an "improved" approach, using the **htmltab** package:

- Read HTML Tables with `htmltab()`

13 Web scraping

Data in unstructured formats on web pages (HTML tags) are common sources of information. Web scraping is a general approach to obtaining that infomration and structuring it. The most common method for doing this in R is the use of the **rvest** package. Here are some tutorials:

- Beginner's guide to web scaping in R using **rvest**
- Tidy web scraping in R - Tutorial and resources
- Practical Introduction to Web Scraping in R

14 Text Mining

Textual analysis is a major field that is beyond the scope/goals of this document, but a few web sites can provide a very good start:

- A light introduction to text analysis in R
- The five packages you should know for text analysis in R
- Text Mining and the **tidytext** package in R

15 Handling date and time values

One topic that requires additional background beyond the scope of this document is how R handles date and time data. Coding and conversions among types of date/time values is a major topic of it's own. Several good starting points on the internet are listed here, but first I show simple code for determining the date/time at the present moment in R.

```
Sys.time()
```

```
## [1] "2020-07-30 10:58:49 EDT"
```

```
Sys.Date()
```

```
## [1] "2020-07-30"
```

- Using Dates and Times in R
- Time data types in R
- Ch 16 from Grolemund/Wickham R for Data Science text
- Date and Time Values in R
- Package **datetime**

16 Write data frames to .txt, .csv, or Excel files

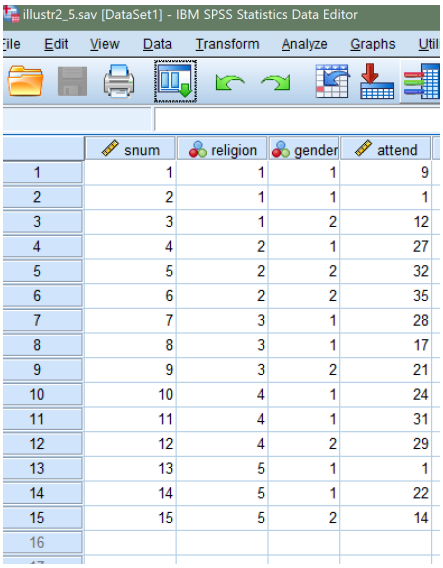
Data objects in R can be written to external files. Here are a few functions that permit that:

- `write.csv`: base system function to write a .csv with comma separation and header
- `write.table`: base system function to write an ascii text file
- `write_csv`: **readr** package function (tidyverse)
- `write_tsv`: **readr** package function for tab-delimited text (tidyverse)
- `write_excel_csv`: **readr** package function (tidyverse)
- `write_xlsx`: **write_xl** package function (tidyverse)
- `write_sav`: **haven** package function for SPSS (tidyverse)
- `write_sas`: **haven** package function SAS (tidyverse)

17 Saving or exporting .csv files from SPSS

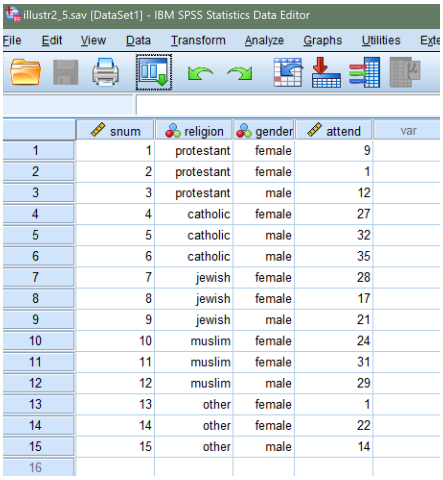
Since many of the users of this document will be frequently working with SPSS, it is helpful to include a section on how to export SPSS data files to .csv files. THIS IS A HIGHLY RECOMMENDED METHOD. CSV FILES ARE A STABLE METHOD OF MOVING DATA BETWEEN OTHER APPLICATIONS AND R. There are a few options in the export process that are helpful to be aware of and are illustrated with screen captures here.

First is a repeat of the earlier screen shot of the religion/gender/attend data file showing the numerically coded religion and gender variables.



	snum	religion	gender	attend
1	1	1	1	9
2	2	1	1	1
3	3	1	2	12
4	4	2	1	27
5	5	2	2	32
6	6	2	2	35
7	7	3	1	28
8	8	3	1	17
9	9	3	2	21
10	10	4	1	24
11	11	4	1	31
12	12	4	2	29
13	13	5	1	1
14	14	5	1	22
15	15	5	2	14
16				

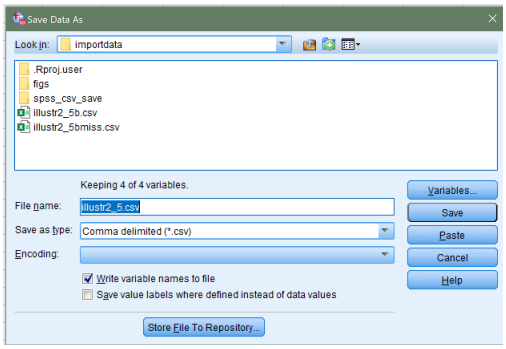
Then, the screen shot of the same data set showing the value labels for those variables:



	snum	religion	gender	attend	var
1	1	protestant	female	9	
2	2	protestant	female	1	
3	3	protestant	male	12	
4	4	catholic	female	27	
5	5	catholic	male	32	
6	6	catholic	male	35	
7	7	jewish	female	28	
8	8	jewish	female	17	
9	9	jewish	male	21	
10	10	muslim	female	24	
11	11	muslim	female	31	
12	12	muslim	male	29	
13	13	other	female	1	
14	14	other	female	22	
15	15	other	male	14	
16					

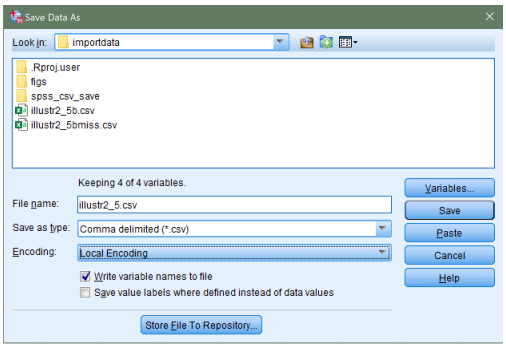
When exporting to a .csv file, one has the option of saving the values as numerics for religion and gender or as the value label strings. The latter will typically be preferred for use in R.

From the File-Export menu choice in SPSS choose the "CSV DATA" option. A dialog box should appear looking something like the following. In addition to choosing the file name and perhaps location to save the file, we need to change two items.

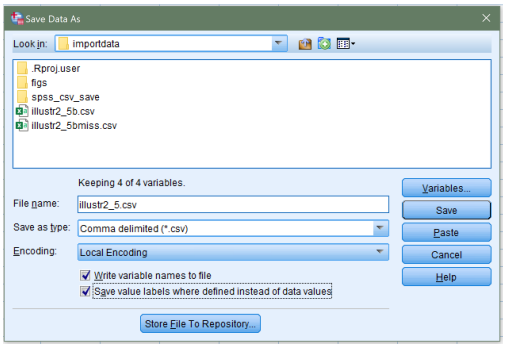


First, change the Encoding to "Local Encoding". Failure to do this can result in a problem shown at the end

of this section.



Also, use the checkbox to save Value Labels instead of numeric values for those categorical variables for which one has included value labels.



The Variables tab gives control over which variables are to be included/dropped in the .csv file. No more effort is required. Just click the save button.

Reading that newly created file with `read.csv` or `read_csv` will produce a data frame or tibble, respectively:

```
df22 <- read.csv("illustr2_5.csv", stringsAsFactors=TRUE, na.strings=c(""))
df22
```

```
##    snum   religion gender attend
## 1     1 protestant female      9
## 2     2 protestant female      1
## 3     3 protestant   male     12
## 4     4   catholic female     27
## 5     5   catholic   male     32
## 6     6   catholic   male     35
## 7     7     jewish female     28
## 8     8     jewish female     17
## 9     9     jewish   male     21
## 10   10     muslim female     24
## 11   11     muslim female     31
## 12   12     muslim   male     29
## 13   13      other female      1
## 14   14      other female     22
## 15   15      other   male     14
```

The following code would read the data with `read_csv`, the tidyverse option, and convert the character vectors to factors (df22b would be a tibble):

```
df22b <- read_csv("illustr2_5.csv")
```

```
## Parsed with column specification:
## cols(
##   snum = col_double(),
##   religion = col_character(),
##   gender = col_character(),
##   attend = col_double()
## )
```

```
df22b <- mutate_if(df22b, is.character, factor)
df22b
```

```
## # A tibble: 15 x 4
##     snum religion   gender attend
##    <dbl> <fct>      <fct>   <dbl>
##  1     1 protestant female      9
##  2     2 protestant female      1
##  3     3 protestant male       12
##  4     4 catholic   female     27
##  5     5 catholic   male       32
##  6     6 catholic   male       35
##  7     7 jewish     female     28
##  8     8 jewish     female     17
##  9     9 jewish     male       21
## 10    10 muslim     female     24
## 11    11 muslim     female     31
## 12    12 muslim     male       29
## 13    13 other      female      1
## 14    14 other      female     22
## 15    15 other      male       14
```

18 Issues with Encoding of ascii text files

One infuriating outcome of reading .txt or .csv files that can occur from time to time is the appearance of odd/extraneous characters in the variable name of the first variable in the data frame. See the appearance of the "snum" variable here. This occurs because of a way the .txt or .csv file was encoded, and reflects the presence of what is called a Byte Order Mark. A common way this happens is if export from SPSS fails to use the "Local" encoding specification as outlined above. Another solution follows here.

```
df23 <- read.csv("illustr2_5encodingissue.csv")
df23
```

```
##    ï..snum religion gender attend
## 1         1        1      1      9
## 2         2        1      1      1
## 3         3        1      2     12
## 4         4        2      1     27
## 5         5        2      2     32
## 6         6        2      2     35
## 7         7        3      1     28
## 8         8        3      1     17
## 9         9        3      2     21
## 10       10        4      1     24
## 11       11        4      1     31
## 12       12        4      2     29
## 13       13        5      1      1
```

```
## 14      14        5       1     22
## 15      15        5       2     14
```

If a BOM is present in the .txt or .csv file, it can be removed by `read.csv` with a `fileEncoding` argument. See the help for the `file` function (?file) and read the "Encoding section" for details. But this code works fine. Another document may be provided to B. Dudek's classes that expands on this problem.

```
df23b <- read.csv("illustr2_5encodingissue.csv", fileEncoding="UTF-8-BOM" )
df23b
```

```
##    snum religion gender attend
## 1     1        1      1      9
## 2     2        1      1      1
## 3     3        1      2     12
## 4     4        2      1     27
## 5     5        2      2     32
## 6     6        2      2     35
## 7     7        3      1     28
## 8     8        3      1     17
## 9     9        3      2     21
## 10   10        4      1     24
## 11   11        4      1     31
## 12   12        4      2     29
## 13   13        5      1      1
## 14   14        5      1     22
## 15   15        5      2     14
```

19 Saving/loading R data frames, objects, and workspaces

Once data are in R as a saved object, it is helpful to save the object to an R file type so that future use would not require re-running the import code, but would just require reading the R file. Data frames and tibbles can be saved to one of two standard data formats—Rdata (sometimes shortened to Rda) and Rds. These formats are used when R objects are saved for later use. Rdata is used to save multiple R objects (or a whole workspace), while Rds is used to save a single R object.

19.1 Rds files for single R objects

Both the `save` and `saveRDS` functions permits saving .Rds files, requiring only that the data object and chosen file name be passed as arguments. `saveRDS` is preferred because of its pairing with `readRDS` seen below. This code will save df21 into the .Rds file, in the working directory.

```
saveRDS(df21, file="df21_tibble.rds")
```

These saved .rda and .rds files can be read with the `load` function. However, it is preferred to use `readRDS` for .Rds files. This will place the original tibble (or data frame) in the working environment, with the original object name (df21 in this case), so make sure you don't overwrite an object that already exists in the working environment with the same name that you inadvertently used again. This is why it is helpful to include the object name in the file name.

```
readRDS("df21_tibble.rds")
# or to use a different name for the object
newdf <- readRDS("df21_tibble.rds")
```

19.2 Tips on file naming

Since these `save` and `saveRDS` functions will overwrite existing files with the same name, it is good practice to use date and/or time info in the file name, or version numbers. For example:

```
saveRDS(df21, file="df21_tibble_july2_2020.rds")
```

19.3 RData files for multiple R objects or whole workspaces

Saving multiple objects to .Rdata format files is done with the `save` function:

```
save(df1, df21, file="df1_df21.Rdata")
```

Loading .Rdata files is done with the `load` function.

```
load("df1_df21.Rdata")
```

19.4 Saving the whole set of Global Environment Objects - the workspace

It is possible to save the whole workspace at once - all objects in the global environment. This permits re-establishing the working environment without re-running all of what might be lengthy code sequences to reproduce those objects. This is best done with the `save.image` function:

```
save.image("expt_20_03_july2_2020.Rdata")
```

Upon exiting RStudio and R there is a dialog box that asks if you want to save the workspace. This `save.image` function is what is used. for that. And then, when opening R again (or the RStudio Project) that workspace is automatically recreated. Using the `save.image` function as shown here does not result in the automatic reload - and this can be a useful strategy. To load a saved workspace, use the `load` function.

```
load("expt_20_03_july2_2020.Rdata")
```

20 Using RStudio to directly import data

Recent updates of RStudio have incorporated a menu driven approach to importing data. Many of the functions outlined above have been incorporated into this capability. It is very slick and easy to use. In RStudio if the default pane configuration has been unaltered by personal preference changes in options for the interface, an "Import Dataset" tab is visible under the "Environment" tab in the upper right hand pane. There, one can choose the file class to be imported and the type of R method to use (base system funtions or tidyverse functions, etc).

In the ensuing dialog box, the name of the imported object can be specified and then the code that was used is found in the console and can be copied/pasted for reproducibility.

This method is a good way to begin to learn some optional arguments available in the various functions - choose the options in the dialog box and look at how the code was written.

This section, perhaps, should have been the first in the document.

21 Documentation for Reproducibility

R software products such as this markdown document should be simple to reproduce, if the code is available. But it is also important to document the exact versions of the R installation, the OS, and the R packages in place when the document is created.

```
sessionInfo()
```

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 18362)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] dplyr_1.0.0    haven_2.3.1    writexl_1.3    readxl_1.3.1   xlsx_0.6.3
## [6] readr_1.3.1    foreign_0.8-80 knitr_1.29
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.5       cellranger_1.1.0 pillar_1.4.6     compiler_4.0.2
##  [5] forcats_0.5.0    tools_4.0.2      digest_0.6.25    evaluate_0.14
##  [9] lifecycle_0.2.0  tibble_3.0.3     pkgconfig_2.0.3  rlang_0.4.7
## [13] cli_2.0.2        curl_4.3         yaml_2.2.1       xfun_0.16
## [17] rJava_0.9-13     stringr_1.4.0    generics_0.0.2   xlsxjars_0.6.1
## [21] vctrs_0.3.2      hms_0.5.3        tidyselect_1.1.0 glue_1.4.1
## [25] R6_2.4.1         fansi_0.4.1      rmarkdown_2.3    purrr_0.3.4
## [29] magrittr_1.5     ellipsis_0.3.1   htmltools_0.5.0  assertthat_0.2.1
## [33] utf8_1.1.4       stringi_1.4.6    crayon_1.3.4
```