05 Importing data

Soyoung Park

Pusan National University
Department of Statistics

Loding library

```
library(tidyverse)
```

Paths and the working directory

The first step when importing data from a spreadsheet is to locate the file containing the data. Here we use the `read_csv` function from the **readr** package, which is part of the tidyverse.

```
library(tidyverse)
dat <- read_csv(filename)
```

The data is imported and stored in `dat`.

The working directory

You can get the full path of your working directory without writing out explicitly by using the `getwd` function.

```
wd <- getwd()
```

If you need to change your working directory, you can use the function `setwd`

Generating path names

Another example of obtaining a full path without writing out explicitly was given above when we created the object `fullpath` like this:

```
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
```

The function `system.file` provides the full path of the folder containing all the files and directories relevant to the package specified by the `package` argument.

Generating path names

By exploring the directories in `dir` we find that the `extdata` contains the file we want:

```
dir <- system.file(package = "dslabs")
filename %in% list.files(file.path(dir, "extdata"))
```

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

Generating path names

The `system.file` function permits us to provide a subdirectory as a first argument, so we can obtain the fullpath of the `extdata` directory like this:

```
dir <- system.file("extdata", package = "dslabs")
```

The function `file.path` is used to combine directory names to produce the full path of the file we want to import.

```
fullpath <- file.path(dir, filename)
```

Copying files using paths

The final line of code we used to copy the file into our home directory used the function `file.copy`.

```
file.copy(fullpath, "murders.csv")
```

```
## [1] FALSE
```

If a file is copied successfully, the `file.copy` function returns `TRUE`.

The readr and readxl packages

In this section we introduce the main tidyverse data importing functions. We will use the `murders.csv` file provided by the **dslabs** package as an example. To simplify the illustration we will copy the file to our working directory using the following code:

```
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
file.copy(fullpath, "murders.csv")
```

```
## [1] FALSE
```

readr

The **readr** library includes functions for reading data stored in text file spreadsheets into R. **readr** is part of the **tidyverse** package, or you can load it directly:

```
library(readr)
```

readr

The following functions are available to read-in spreadsheets:

Function	Format	Typical suffix
read_table	white space separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv
read_delim	general text file format, must define delimiter	txt

readr

We can open the file to take a look or use the function `read_lines` to look at a few lines:

```
read_lines("murders.csv", n_max = 3)
```

```
## [1] "state,abb,region,population,total" "Alabama,AL,South,4
## [3] "Alaska,AK,West,710231,19"
```

readr

Now we are ready to read-in the data into R. From the .csv suffix and the peek at the file, we know to use `read_csv`:

```
dat <- read_csv(filename)
```

```
##
## -- Column specification --------------------------------
## cols(
##   state = col_character(),
##   abb = col_character(),
##   region = col_character(),
##   population = col_double(),
##   total = col_double()
## )
```

readr

We can confirm that the data has in fact been read-in with:

```
View(dat)
```

Finally, note that we can also use the full path for the file:

```
dat <- read_csv(fullpath)
```

```
##
## -- Column specification --------------------------------
## cols(
##   state = col_character(),
##   abb = col_character(),
##   region = col_character(),
##   population = col_double(),
##   total = col_double()
## )
```

readxl

You can load the readxl package using

```
library(readxl)
```

readxl

The package provides functions to read-in Microsoft Excel formats:

Function	Format	Typical suffix
read_excel	auto detect the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new format	xlsx

Exercises

1. Use the `read_csv` function to read each of the files that the following code saves in the `files` object:

```
path <- system.file("extdata", package = "dslabs")
files <- list.files(path)
files
```

Exercises

2. Note that the last one, the `olive` file, gives us a warning. This is because the first line of the file is missing the header for the first column.

Read the help file for `read_csv` to figure out how to read in the file without reading this header. If you skip the header, you should not get this warning. Save the result to an object called `dat`.

Exercises

3. A problem with the previous approach is that we don't know what the columns represent. Type:

```
names(dat)
```

to see that the names are not informative.

Use the `readLines` function to read in just the first line (we later learn how to extract values from the output).

Downloading files

Another common place for data to reside is on the internet. For example, we note that because our **dslabs** package is on GitHub, the file we downloaded with the package has a url:

```
url <- "https://raw.githubusercontent.com/rafalab/dslabs/
master/inst/extdata/murders.csv"
```

The `read_csv` file can read these files directly:

```
dat <- read_csv(url)
```

Downloading files

If you want to have a local copy of the file, you can use the `download.file` function:

```
download.file(url, "murders.csv")
```

This will download the file and save it on your system with the name murders.csv.

R-base importing functions

R-base also provides import functions. These have similar names to those in the **tidyverse**, for example `read.table`, `read.csv` and `read.delim`. You can obtain an data frame like `dat` using:

```
dat2 <- read.csv(filename)
```

R-base importing functions

An often useful R-base importing function is `scan`, as it provides much flexibility. With scan you can read-in each cell of a file. Here is an example:

```
path <- system.file("extdata", package = "dslabs")
filename <- "murders.csv"
x <- scan(file.path(path, filename), sep = ",", what = "c")
x[1:10]
```

```
##  [1] "state"      "abb"        "region"     "population" "t
##  [6] "Alabama"    "AL"         "South"      "4779736"    "1
```

Note that the tidyverse provides `read_lines`, a similarly useful function.

Text versus binary files

For data science purposes, files can generally be classified into two categories: text files (also known as ASCII files) and binary files. The csv tables you have read are also **text files**. One big advantage of these files is that we can easily "look" at them without having to purchase any kind of special software or follow complicated instructions.

Any text editor can be used to examine a text file, including freely available editors such as RStudio, Notepad, textEdit. To see this, try opening a csv file using the "Open file" RStudio tool.

However, if you try to open, say, an Excel xls file, jpg or png file, you will not be able to see anything immediately useful. These are **binary files**. Excel files are actually compressed folders with several text files inside. But the main distinction here is that **text files can be easily examined**.

Text versus binary files

Although R includes tools for reading widely used binary files, such as xls files, in general you will want to find data sets stored in text files. In general, plain-text formats make it easier to share data since commercial software is not required for working with the data.

Extracting data from a spreadsheet stored as a text file is perhaps the easiest way to bring data from a file to an R session. Unfortunately, spreadsheets are not always available and the fact that you can look at text files does not necessarily imply that extracting data from them will be straightforward.

Unicode versus ASCII

A pitfall in data science is assuming a file is an ASCII text file when, in fact, it is something else that can look a lot like an ASCII text file: a Unicode text file.

To understand the difference between these, remember that everything on a computer needs to eventually be converted to 0s and 1s. **ASCII** is an *encoding* that maps characters to numbers. ASCII uses 7 bits (0s and 1s) which results in $2^7 = 128$ unique items, enough to encode all the characters on an English language keyboard.

However, other languages use characters not included in this encoding. For example, the é in México is not encoded by ASCII.

Unicode versus ASCII

For this reason, a new encoding, using more than 7 bits, was defined: Unicode. When using Unicode, one can chose between 8, 16, and 32 bits abbreviated **UTF-8**, **UTF-16**, and **UTF-32** respectively. RStudio actually defaults to **UTF-8** encoding.

Although we do not go into the details of how to deal with the different encodings here, it is important that you know these different encodings exist so that you can better diagnose a problem if you encounter it.

This StackOverflow discussion is an example: https://stackoverflow.com/questions/18789330/r-on-windows-character-encoding-hell.

Exercises

1. Pick a measurement you can take on a regular basis. For example, your daily weight or how long it takes you to run 5 miles. Keep a spreadsheet that includes the date, the hour, the measurement, and any other informative variable you think is worth keeping. Do this for 2 weeks. Then make a plot.