A Short Introduction to Working With Data in R
EXTRAS

Jonathan Whiteley

2023-09-19

1. Reading a csv file with base `R`
2. File Encoding, Windows, & Microsoft Excel™
3. Downloading Data From the Internet
4. Reading Data in Other Formats
5. Exporting to other formats
6. Programming
7. Some Advanced `dplyr` Examples

Section 1

Reading a csv file with base R

Load a csv file using read.csv()

```
?read.csv
```

- If read.csv() encounters problems reading a file, it is more likely to trigger an **error** than read_csv(), which gives a warning more often.

```
DF_path <- file.path("..", "data", "data_example.csv")
try( read.csv(DF_path) )
```

```
# Error in read.table(file = file, header = header, sep = sep, quote
#   more columns than column names
```

Check the file contents

- Let's take a peek at the first few lines and see if we can identify the problem (this is more often necessary with `read.csv()`):

```
readLines(DF_path, n = 4)
```

```
# [1] "Data from an experiment on the cold tolerance of the grass sp
# [2] "Modified from `data(CO2)`.  See `?CO2`."
# [3] "Type,Treatment,PlantNum,95,175,250,350,500,675,1000"
# [4] "Quebec,nonchilled,1,16,30.4,34.8,37.2,35.3,39.2,39.7"
```

- The first **2** lines don't look like comma-separated values!
- They look like extra information that is not part of the data table *structure*.

Load a csv file into R

- We can tell R to skip the lines with no data:

```
DF <- read.csv(DF_path, skip = 2)
DF_readr <- readr::read_csv(DF_path, skip = 2)
```

- Just because there were no Errors from R, doesn't mean there's nothing wrong with the data!

Loading data: readr vs base R functions

readr		base R	
read_csv()	comma separated values	read.csv()	
read_csv2()	';' as delimiter (allows ',' for decimals)	read.csv2()	',' for decimals, ';' as separator
read_tsv()	tab separated values	read.delim()	delimited files (tab is default)
read_delim()	(generic) files with any delimiter	read.table()	
read_fwf()	fixed width files	read.fwf()	

readr descriptions based on #dsbox

Comparison of `read.csv()` and `read_csv()`

- In keeping with Tidyverse conventions, functions are names with words separated by "_"
  - instead of "`.`" or `camelCase`, as in many base `R` functions
- The column names are different
  - `read.csv()` automatically applies `make.names()` to the column names to make 'syntactically valid' names to use in `R`.
  - convenient, but not always what we want.
  - there are other 'cleaning' functions available (e.g., `clean_names()` in the `janitor` package)
- `read_csv()` automatically replaced empty strings in the `Treatment` column with `NA`s.
- `read_csv()` left the '675' column as numeric, but ignored the commas, resulting in larger numbers.
- `read_csv()` produces a `"tbl_df"` (*tibble*) object, not a simple `data.frame`

Tibble examples

- Tibbles have an enhanced `print()` method
- and they will *not* do partial matching on variable names, triggering a *warning* instead for columns that do not exist.

```
print(DF_readr, n=2)
```

```
# # A tibble: 13 x 10
#   Type   Treatment  PlantNum  `95` `175` `250` `350` `500`
#   <chr>  <chr>         <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
# 1 Quebec nonchilled        1  16    30.4  34.8  37.2 35.3
# 2 Quebec <NA>              2  13.6  27.3  37.1  41.8 40.6
# # i 11 more rows
# # i 2 more variables: `675` <dbl>, `1000` <dbl>
```

```
is.null(DF$Treat)
```

```
# [1] FALSE
```

```
is.null(DF_readr$Treat)
```

```
# Warning: Unknown or uninitialised column: `Treat`.
```

```
# [1] TRUE
```

Writing data: `readr` vs base `R` functions

readr		**base R**
`write_csv()`	← comma separated values	`write.csv()`
`write_csv2()`	← allows ';' as delimiter and ',' for decimals (depending on locale)	`write.csv2()` ',' for decimals, ';' as separator
`write_tsv()`	← tab separated values	
`write_delim()`	← (generic) files with an arbitrary delimiter	`write.table()`
`write_excel_csv()`, `write_excel_csv2()`	← include a UTF-8 Byte order mark, which indicates to Excel the csv is UTF-8 encoded	

Section 2

File Encoding, Windows, & Microsoft Excel^{TM}

Encoding non-English characters

- If you are running `R` in Windows, you may notice that some text values (character) look strange when read with `read.csv()`:

  "**QuÃ©bec**" instead of "Québec"

- There is nothing wrong with the file — this indicates a *mismatch* between the *encoding* used to write the file, and what `R` used to read it.

- Even though '`.csv`' files are plain text, letters (especially non-english characters) can be *encoded* in different ways to represent them in the computer.

- "`UTF-8`" is a character encoding standard designed to handle many non-english characters.
  - The example data file was written in "`UTF-8`"
  - Most OSes and many programs use "`UTF-8`" encoding by default.
  - But *Windows* uses "`latin1`" by default, and so does `R` (< 4.2.0) when running in Windows.
  - Starting with v4.2.0, `R` uses "`UTF-8`" as the default encoding on Windows.

Read a csv file with a different encoding

- You can specify the encoding used in the file with the ‘encoding’ argument of read.csv()

```
DF <- read.csv(DF_path, skip = 2, encoding = "UTF-8")
```

- If reading a file that was created on a Windows computer and encoded in “latin1”, on a different system (mac, Unix, linux, etc.) — or a recent version of R (>=4.2) on Windows — you can specify that, too:

```
read.csv(DF_path, skip = 2, encoding = "latin1")
```

Encoding & Microsoft Excel™

- Excel can save a `.csv` file using `UTF-8` encoding, but in doing so, it adds "**b**yte **o**rder **m**ark" ("BOM") to the file.
  - This is a special character that Excel also uses to recognize that the file is encoded using `UTF-8`.
  - Thus a BOM can make the file "easier" to use with Excel, by allowing it to automatically recognize the `UTF-8` encoding, but it can also **cause problems for other programs** (like `R`) that do not expect such a non-Unicode character.
- Without the BOM, Excel will assume the file is encoded in "`latin1`" if you double-click on the csv file to open it in Excel, even if it was actually encoded with `UTF-8`.
  - This can cause special characters to appear incorrectly.
  - You can still import a `.csv` file encoded in `UTF-8` into Excel correctly, but it requires opening the file within Excel, or importing it using commands in the "Data" ribbon / menu

Read a file with a BOM using `read.csv()` in R

- Reading a `.csv` file with a BOM using the usual method may cause the BOM to be included in the name of the first column (on Windows).

```
bom_bad <- read.csv("../data/data_example_bom.csv",
                    encoding = "UTF-8")
names(bom_bad)
```

- The solution with `read.csv()` is to use the argument ‘`fileEncoding = "UTF-8-BOM"`’ (instead of the ‘`encoding`’ argument)

```
bom <- read.csv("../data/data_example_bom.csv",
                fileEncoding = "UTF-8-BOM")
bom[4, 1:4]
```

```
#     Type Treatment PlantNum  X95
# 4 Québec   chilled        1 14.2
```

Read a file with a BOM using `read_csv()`

- The `readr` package uses "`UTF-8`" encoding by default, and **automatically ignores a BOM**, if present.

```
bom_readr <- readr::read_csv("../data/data_example_bom.csv")
bom_readr[4, 1:4] |> knitr::kable()
```

Type	Treatment	PlantNum	95
Québec	chilled	1	14.2

- `write_csv()` (in the `readr` package) automatically encodes output files using "UTF-8", for greater portability across systems.
  - *except* for older versions of base `R` (`read.csv()`) on Windows :(

!

Hopefully, these examples have demonstrated that the `readr` package makes it easy to work with "`UTF-8`" files by default, on any platform.

Add a BOM to an output file

- It is possible to add a BOM to a csv file, but it must be done *manually* with base R:
    - code adapted from this StackOverflow answer

```
writeChar(
  iconv("\ufeff", to = "UTF-8"),
  "output.csv",
  eos = NULL
)
write.csv(Data, "output.csv", append = TRUE, ... )
```

- The readr package can do this directly with a special write_excel_csv function:

```
write_excel_csv(Data, "output.csv", ... )
```

!

R **does not recommend doing this** (see ?file), so use with caution.

Using other encodings with `readr`

- You can control the encoding used by `readr` functions with the `locale` **argument**.

```
write_csv(Data, "data.csv",
          locale = locale(encoding = "latin1")
)

read_csv(Data, "data.csv",
         locale = locale(encoding = "latin1")
)
```

- See `?readr::read_csv` and `?readr::locale` for details.

Section 3

Downloading Data From the Internet

Downloading Data From the Internet

- You can use read.csv & read_csv with urls as file paths:

```
csv_web <-
  read_csv( paste0(
    "https://raw.githubusercontent.com/jawhiteley/",
    "R_training_jaw/main/R2_data_scripts/",
    "data/data_example.csv"
    ),
    skip = 2
  )
```

```
# Rows: 13 Columns: 10
# -- Column specification ------------------------------------
# Delimiter: ","
# chr (3): Type, Treatment, 500
# dbl (6): PlantNum, 95, 175, 250, 350, 1000
# num (1): 675
#
# i Use `spec()` to retrieve the full column specification for this
```

Section 4

Reading Data in Other Formats

Other `tidyverse` packages

- The Data Import Chapter of R for Data Science (2e) describes these tidyverse packages for other types of data:
  - **`haven`** reads SPSS, Stata, and SAS files.
  - **`DBI`**, along with a database specific backend (e.g. `RMySQL`, `RSQLite`, `RPostgreSQL`, etc.) allows you to run SQL queries on a **database** and return a data frame.

Other options

- The `foreign` package can read data in a variety of formats, including: 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', and others
  - *May require access to external software to read their formats*
- Parquet files: an efficient columnar format, popular with Big Data and cloud computing
  - Apache Arrow (i.e., the '`arrow`' package)
- See the Import Section of R for Data Science (2nd edition) For more details on getting data into R from these and other sources.
- Other options are also described in the R Data Import/Export manual.

Section 5

Exporting to other formats

Writing to Microsoft Excel^{TM} files

Packages that can write to Excel files:

- `xlsx`: read, write, format Excel 2007 (`.xlsx`) and Excel 97/2000/XP/2003 (`.xls`) files.
  - Requires Java and the `rJava` package
- `XLConnect`: comprehensive and cross-platform R package for manipulating Microsoft Excel files (`.xlsx` & `.xls`) from within R.
  - Requires a Java Runtime Environment (JRE)
- `openxlsx`: simplified creation of Excel `.xlsx` files (**not** `.xls`).
  - *No dependency* on Java
- `writexl`: portable, light-weight data frame to **xlsx** exporter.
  - No Java or Excel required

!

I recommend *avoiding* exporting data to Excel files if possible. `csv` files are easier to read to & write from, and can be read by a wider variety of software (they are more portable).
Automated reports can be produced with R Markdown and output to a variety of more portable formats (pdf, HTML, etc.) instead.

Section 6

Programming

Control flow: `if` conditions

```
?Control
```

- Control what code in a script actually runs with conditional expressions and `if` statements

```
if (condition) {
  message("The condition is TRUE")
}
```

- **IF** the *expression* (condition) in the parentheses `()` evaluates to `TRUE`, **then** the code inside the braces `{}` will run.
  - Otherwise, it will not.

Control flow: `if` & `NA`

- The conditional expression *must* result in *either* `TRUE` **or** `FALSE`; anything else causes an **error**.
  - `NA` is a common source of problems here.
  - `isTRUE()` and `isFALSE()` are useful to ensure a `TRUE`/`FALSE` result.

```
if (NA) {
  message("The condition is NA")
}
```

```
# Error in if (NA) { : missing value where TRUE/FALSE needed
```

```
if (isTRUE(NA)) {
  message("The condition is NA")
}
if (is.na(NA)) {
  message("The condition is TRUE")
}
```

```
# The condition is TRUE
```

Control flow: `if`, `else` conditions

- An `else` condition is optional:
  IF the *expression* (condition) in the `if` statement evaluates to `FALSE`, **then** the code inside the braces {} *after* `else` will run.

```
if (isTRUE(condition)) {
  message("The condition is TRUE")
} else {
  warning("The condition is FALSE")
}
```

```
if (isFALSE(0 == 1)) {
  message("The condition is TRUE")
} else {
  warning("The condition is FALSE")
}
```

```
# The condition is TRUE
```

Control flow: multiple `if`, `else` conditions

- You can have multiple "`if-else`" statements, as long as each `if` after the first follows immediately after the `else` keyword.

```
if (isTRUE(NA)) {
  stop("NA is TRUE")
} else if (isFALSE(NA)) {
  warning("NA is FALSE")
} else {
  message("NA is neither TRUE nor FALSE")
}
```

```
# NA is neither TRUE nor FALSE
```

Control flow: `for` loops

```
?Control
```

- Tell R to run a block of code multiple times in a row with a *loop*

```r
for (i in 1:3) {
  print( paste("i =", i) )
}
```

```
# [1] "i = 1"
# [1] "i = 2"
# [1] "i = 3"
```

`for` loops

- Put inside the parentheses `()`, in order:
  - The **name of a variable** to update each time through the loop (*iteration*)
  - The keyword '`in`'
  - A **vector** of values
- The code inside the the braces {} is run with the specified *variable* assigned each value in the *vector*, in sequence.

User-defined function

- Define your own functions with the `function` function!
  - function code goes between braces: {}
  - specify what value is returned with the `return()` function

```
'%==%' <- function (v1, v2) {
  same <- (v1 == v2) | (is.na(v1) & is.na(v2))
  same[is.na(same)] <- FALSE
  return(same)                    # return the result
}

# test it:
c(1, NA, 3, 4 , NaN) %==% c(1, NA, 1, NA, NaN)
```

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

```
c(1, NA, 3, 4 , NaN)  ==  c(1, NA, 1, NA, NaN)
```

```
# [1]  TRUE    NA FALSE    NA    NA
```

- This code defines a function that compares two vectors, accounting for missing values (`NA`)

An infix operator

```
c(1, NA, 3, 4 , NaN) %==% c(1, NA, 1, NA, NaN)
```

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

- This function is also a special type called an "*infix operator*", which goes between two objects (it's arguments) like an operator, instead of a 'typical' function call
  - it has exactly **2** arguments (lhs, rhs)
  - the name begins *and* ends with a percent symbol (%)

Debugging

These are some useful functions for troubleshooting or *debugging* code that is not doing what you want:

- `?traceback`: prints a list ("stack") showing the order of expressions that triggered the last error
  - useful in cases where functions call other functions iteratively, and you want to know which function in the "stack" caused the error.
- `?debug`: flag a function for debugging
  - The next time the function is called, it will open an interactive 'browser' session that lets you run the code in the function 1 line at a time, and even explore how objects change between lines.
- `?browser`: open the same interactive session as `debug()`, but at a specific location in the code (*anywhere*)

For more information, check out these (and other) sources: Advanced R, R Programming for Data Science, Debugging with the RStudio IDE

Section 7

Some Advanced `dplyr` Examples

Find values systematically

- Find columns that are character, but we expect to be numeric:

```
cols_charn <- DF %>%
  select(starts_with("X") & where(is.character)) %>%
  names()
```

- Loop through the identified column names (using `for`) and print values that would be `NA` if converted to numeric
  - `suppressWarnings()` suppresses the warnings we expect from `as.numeric()` in this case.

```
for (col in cols_charn) {
  DF %>% select(1:3, all_of(col)) %>%
    filter( get(col) %>% as.numeric() %>% is.na() %>%
              suppressWarnings() ) %>%
    print()
}
```

```
#     Type Treatment PlantNum                     X500
# 1 Québec   chilled        1 32.5 (umol/m^2 sec)
#     Type Treatment PlantNum X675
# 1 Québec   chilled        1 35,4
# 2 Québec                  2 37,5
# 3 Québec                  3 39,6
```

summarize() multiple columns with across()

- summarize() uses *mutate semantics*, but across() applies a **function** to multiple columns, specified using *select semantics*
  - i.e., across() lets you use *select semantics* in a context where you would normally use *mutate semantics*
- A function can be specified by name

```
DF %>%
  summarize( across(where(is.numeric), max) )
```

```
#   PlantNum  X95 X175 X250 X350 X1000
# 1        3 16.2 32.4   NA 42.1    NA
```

```
DF %>% group_by(Treatment) %>%
  summarize( across(starts_with("X"), max) )
```

```
# # A tibble: 3 x 8
#   Treatment      X95  X175  X250  X350 X500       X675  X1000
#   <chr>        <dbl> <dbl> <dbl> <dbl> <chr>      <chr> <dbl>
# 1 ""            16.2  32.4  NA    42.1 42.9       43.9   NA
# 2 "chilled"     14.2  24.1  30.3  34.6 32.5 (um~ 35,4   38.7
# 3 "nonchilled"  16    30.4  34.8  37.2 35.3       39.2   39.7
```

summarize() across() with ad hoc function

- For more complex operations, you may have to define a *custom function* or define an *ad hoc function* to include additional arguments

```
DF %>%
  summarize(
    across(where(is.numeric),
           function(x) max(x, na.rm = TRUE)
    )
  )
```

```
#   PlantNum  X95 X175 X250 X350 X1000
# 1        3 16.2 32.4 40.3 42.1  45.5
```

`summarize()` `across()` with "lambda notation"

- You can also define ad hoc functions using a special "lambda" notation
  - refer to the value in the column with '`.x`'

```
DF %>%
  summarize( across(everything(), ~ sum(is.na(.x))) )
```

```
#   Type Treatment PlantNum X95 X175 X250 X350 X500 X675
# 1    0         0        0   0    0    1    0    0    0
#   X1000
# 1     1
```

Tip

`sum(is.na())` *is a great way to count the number of missing values in a column.*

References (Extras)

CANSIM / CODR data:

- An ecosystem of R packages to access and process Canadian data
- Analyzing Canadian Demographic and Housing Data