Lesson 3: Reading files - beyond the basics

Patrick Mathias

This is a much shorter and less philosophical lesson than the previous lessons but hopefully is very useful when considering how to pull data into R.

Base functions for reading and writing files

Reading files

R has solid built-in functions for importing data from files with the `read.table()` family of functions. `read.table()` is the generic form that expects a filename (in quotes) at a minimum and, importantly, an indication of the separator character used - it defaults to "" which indicates white space (one or more spaces, tabs, newlines, or carriage returns). The default header parameter for `read.table()` is FALSE, meaning that the function will **not** use the first row to determine column names. Because non-Excel tabular files are generally comma-delimited or tab-delimited with a first row header, `read.csv()` and `read.delim()` are the go-to base file reading functions that include a `header = TRUE` parameter and use comma and tab delimting, respectively, by default.

There are a variety of other useful parameters to consider, including explicitly supplying the column names via the `col.names` parameter (if not defined in header, for example). One related group of parameters to be conscious of with these functions are `stringsAsFactors` and `colClasses`. When R is reading a file, it will convert each column to a specific data type based on the content within that column. The default behavior of R is to convert columns with non-numeric data into a factor, which are a representation of categorical variables. For example, you may want to separate out data by sex (M/F) or between three instruments A, B, and C, and it makes perfect sense to represent these as a factor, so that you can easily stratify the groups during analyses in R, particularly for modeling questions. So, by default, with these base functions `stringsAsFactors = TRUE`, which means that any columns with characters may not have the expected behavior when you analyze the data. In general this may not be a big deal but can cause problems in a couple scenarios: 1. You are expecting a column to be a string to parse the data (using the stringr package for example). Not a huge deal - you can convert to a character 2. There are typos or other data irregularities that cause R to interpret the column as a character and then automatically convert to a factor. If you are not careful and attempt to convert this column back to a numeric type (using `as.numeric()` for example), you can end up coverting the column to a completely different set of numbers! That is because factors are represented as integers within R, and using a function like `as.numeric()` will convert the value to its backend factor integer representation. So `c(20, 4, 32, 5)` could become `c(1, 2, 3, 4)` and you may not realize it.

Problem #2 will come back to haunt you if you are not careful. The brute force defense mechanism is to escape the default behavior: `read.csv("file_name.csv", stringsAsFactors = FALSE)`. This will prevent R from converting any columns with characters into factors. However, you may want some of your columns to be represented as factors. You can modify behavior on a column by column basis. `read.csv("file_name.csv", colClasses = c("character", "factor", "integer")` will set a 3 column csv file to character, factor, and integer data types in that column order.

To be safe, the best practice is arguably to explicitly define column types when you read in a file. It is a little extra work up front but can save you some pain later on.

For the curious, additional information about the history of of stringsAsFactors can be found here.

> Exercise 1
>
> Let's run through the base reading function with a csv. 1. Use the base `read.csv()` function to read the "2017-01-06.csv" file into a data frame. Recall how to use the `here()` function - it is using the "coursework" directory as the point of reference 1. What is the internal structure of the

object? (Recall the `str()` command to quickly view the structure.) 1. What does the data look like? (Recall the `summary()` function to view column types and characteristics about the data.)

```
base_load <- read.csv(here::here("class_data", "2017-01-06.csv"))
# str(base_load)
summary(base_load)
```

Repeat the previous steps starting with #2, but include the argument `stringsAsFactors = FALSE` when you read in the data.

```
base_load_nofactors <- read.csv(here("class_data", "2017-01-06.csv"),
                                stringsAsFactors = FALSE)
# str(base_load_nofactors)
summary(base_load_nofactors)
```

For this data set, which fields should be strings and which should be factors?

Writing files

The functions for reading files in base R have equivalents for writing files as well: `write.table()` and `write.csv()`. The first argument in these functions is the data frame or matrix to be written and the second argument is the file name (in quotes).

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

There are a few other important parameters: - `sep` indicates the field separator ("\t" for tab) - `row.names` is set to TRUE by default - in general this makes for an ugly output file becuase the first column shows the row number (I almost always set this to FALSE) - `na` indicates the string to use for missing data and is set to R's standard of "NA" by default - `append` can be set to TRUE if you would like to append your data frame/matrix to an existing file

Speeding things up with the *readr* package

Base R functions get the job done, but they have some weaknesses: - they are slow for reading large files (slow compared to?) - the automatic conversion of strings to factors by default can be annoying to turn off - output with row names by default can be annoying to turn off

One package in the tidyverse family meant to address these issues is readr. This package provides functions similar to the base R file reading functions, with very similar function names: `read_csv()` (instead of `read.csv()`) or `read_delim()` for example. Tab-delimited files can be read in with `read_tsv()`. These functions are ~10x faster at reading in files than the base R functions and do not automatically convert strings to factors. Readr functions also provide a helpful syntax for explicitly defining column types:

```
# purely a dummy example, not executable!
imaginary_data_frame <- read_csv(
  "imaginary_file.csv",
  col_types = cols(
    x = col_integer(),
    y = col_character(),
    z = col_datetime()
  )
)
```

Another advantage of these functions is that they actually explicitly tell you how the columns were parsed when you import (as we'll see in the exercise).

Readr also offers equivalent write functions such as `write_csv()` and `write_tsv()`. There is a variant of `write_csv()` specifically for csv files intended to be read with Excel: `write_excel_csv()`. These functions do not write row names by default.

> Exercise 2
>
> Now let's run through using the readr function for a csv: 1. Use the `read_csv()` function to read the "2017-01-06.csv" file into a data frame. 1. What is the internal structure of the object? (Recall the `str()` command to quickly view the structure.) 1. What does the data look like? (Recall the `summary()` function to view column types and characteristics about the data.)

```
readr_load <- read_csv()
```

Now compare the time required to run the base `read.csv()` function with the readr `read_csv()` function using `system.time()`.

Time to read with base:

```
system.time(base_load <- read.csv(here("class_data", "2017-01-06.csv")))
```

Time to read with readr:

```
system.time(readr_load <- read_csv(here("class_data", "2017-01-06.csv")))
```

Finally, let's follow some best practices and explicitly define columns with the `col_types` argument. We want to explicitly define compoundName and sampleType as factors. Note that the `col_factor()` expects a definition of the factor levels but you can get around this by supplying a `NULL`. Then run a summary to review the data

```
readr_load_factors <- read_csv(
  here("class_data", "2017-01-06.csv"),
  col_types = cols(
    compoundName = col_factor(NULL),
    sampleType = col_factor(NULL)
    )
  )
summary(readr_load_factors)
```

Dealing with Excel files (gracefully)

You may have broken up with Excel, but unfortunately many of your colleagues have not. You may be using a little Excel on the side. (Don't worry, we don't judge!) So Excel files will continue to be a part of your life. The readxl package makes it easy to read in data from these files and also offers additional useful functionality. As with the other file reading functions, the syntax is pretty straightforward: `read_excel("file_name.xlsx")`. Excel files have an added layer of complexity in that one file may have multiple worksheets, so the `sheet = "worksheet_name"` argument can be added to specify the desired worksheet. Different portions of the spreadsheet can be read using the `range` arugment. For example a subset of rows and columns can be selected via cell coordinates: `read_excel("file_name.xlsx", range = "B1:D6")` or `read_excel("file_name.xlsx, range = cell_cols("A:F")`.

If you are dealing with Excel data that is not a traditional tabular format, the tidyxl package is useful to be aware of. We will not cover it in this course but it is worth reading up on if you ever have to analyze a pivot table or some other product of an Excel analysis.

> Exercise 3

You might be able to guess what comes next: we'll read in an Excel file. 1. Use the `read_excel()` function to read the "2017-01-06.xlsx" file into a data frame 1. View a summary of the imported data 1. Now read in only the first 6 columns using the `range` parameter 1. Review the first 6 lines of the imported data

```
readxl_load <- read_excel( )
summary()
readxl_load_subset <- read_excel( , range = )
head(readxl_load_subset)
```

Importing dirty data

To close out the discussion on reading files, there is one more useful package to introduce that helps with a variety of data cleaning functions. Since this is R, the package is cleverly and appropriately named janitor. The quick take home in terms of useful functions from this package: - `clean_names()` will reformat column names to conform to the tidyverse style guide: spaces are replaced with underscores & uppercase letters are converted to lowercase - empty rows and columns are removed with `remove_empty_rows()` or `remove_empty_columns()` - `tabyl(variable)` will tabulate into a data frame based on 1-3 variables supplied to it

Let's take these functions for a spin using our data set. We are going to use the development version of the package because there is new, additional functionality. I will chain the commands together with pipes (which we'll discuss in more detail in the next lesson).

First let's review the first few lines of data after cleaning the columns names:

```
devtools::install_github("sfirke/janitor")
# the development version of janitor handles cleaning names better than the current CRAN version
library(janitor)
readr_load_cleaned <- readr_load %>%
  clean_names()
head(readr_load_cleaned)
```

Now we'll do a quick tabulation to count the number of rows:

```
readr_load_cleaned %>% tabyl(compound_name)
```

Summary

- The base R functions for reading files `read.delim()`, `read.csv()`, etc. are useful tools but it is important to recognize how they handle strings (and the dangers in automatic conversion to factors)
- readr functions such as `read_delim()` or `read_csv()` are faster than base R functions and do not automatically convert strings to factors
- The readxl function `read_excel()` reads Excel files and offers functionality in specifying worksheets or subsets of the spreadsheet
- The janitor package can help with cleaning up irregularly structured input files