Lesson 8: Beyond the csv – parsing xml and json files

Randall Julian, Adam Zabell

1/2/2018

Flat and Structured Files

The commands *read_csv*, *read_tsv*, and *read_excel* (discussed in Lesson 5) open files that are often called 'flat' because there is no inherent structure to the delimited rows and columns. Using the first row as a column header is a common convention, but is not required for the function to work properly. In contrast, most databases have a required structure to each element, and these elements sometimes have a nestest structure within structure. The details of database construction are well beyond the scope of this class, but if the source of your data is a database query, the returned results are typically in XML or JSON format. Being able to work with these files in **R** requires some basic parsing techniques.

The extensible markup language (xml)

When read as a text file, xml looks a lot like html, because html is (by and large) a subset of xml. That's not to say they're interchangable, since html permits some shortcuts which xml does not, and xml isn't restricted to web browser tags. The xml format declares and closes a *key*, with the *value* enclosed between them: `<key>value</key>`. That value can itself be one or more nested key-value statements.

Reading an xml file requires the **xml2** package (or one like it). Consider a short list of album information, where each entry contains the same number of keys, and those keys are always in the same order.

```r
cd <- read_xml("https://www.w3schools.com/xml/cd_catalog.xml")
cd.list <- as_list(cd)
cd.rows <- length(cd.list)
cd.names <- names(cd.list[[1]])
cd.df <- data.frame(
  matrix(
    unlist(cd.list),
    nrow = cd.rows,
    byrow = TRUE,
    dimnames = list(NULL, cd.names)
  ),
  stringsAsFactors = FALSE
)
```

The javascript object notation (json)

A web browser can only send one thing, or retrieve one thing, at a time. The most complex online form has to be reduced to a single string, and then unpacked on the other side. Although that string could be sent as xml, virtually every website operates with json because this format is more lightweight than xml and thus takes less time to transmit. The format looks very different, but operates on the same principle of first naming a thing, and then giving it a value. With json, the key-value statements are separated by a colon, and each statement is separated by a comma: `{"key":value}`. Nested terms are explicitly either an array (ordered, and enclosed in square brackets) or an object (unordered, and enclosed in curly brackets).

Reading a json file requires the **jsonlite** package (or one like it). Consider an enumeartion of named colors and their RGB value.

```r
colr <- fromJSON("https://raw.githubusercontent.com/corysimmons/colors.json/master/colors.json")
colr.rows <- length(colr)
colr.df <- data.frame(
  matrix(
    unlist(colr),
    nrow = colr.rows,
    byrow = TRUE,
    dimnames = list(NULL, c("R", "G", "B", "T"))
  ),
  stringsAsFactors = FALSE
)
colr.df$name <- names(colr)
```

Solving problems

The most common problem is that numbers, even if they were stored as numbers, can become text strings during conversion to a data.frame. Jumping ahead to the next lesson, the simplest fix will loop on the column of data that should be numeric.

```r
cd.df$PRICE %<>% map_dbl(as.numeric) # also invokes a from-and-back-into pipe notation
```

> Exercise 1: Renumbering Redux
> Will reading the xml into a tibble (instead of a data.frame) solve this number recognition? First, try it. Then, even if it did work, construct a 'standard pipe' mechanism using `%>%` to change both *PRICE* and *YEAR* to double precision numbers.

```r
# cd.tibble <- as_tibble( )
```

There are many more problems that could happen, ultimately because each entry in the xml or json file is an independent object. There may be an element not present in every entry (e.g. not including a price for a CD), or multiple elements within an element (e.g. two artists for a CD), or the elements may not be retrieved in the same order each time (e.g. all Bob Dylan albums return the artist before the title). All are disasterous; the conversion examples shown here using `matrix` will now be wildly offset.

Most problems should be solved on the query-side of the equation, to confirm the database query was built correctly and that the database is populated as expected. But, since "two artists for one CD" is at least possible and might even be common, the retrieved data could simply be less data.frame friendly than you would like. Unless you're very sure of what the returned data must look like, some basic checks now will avoid exhausting effort later.

The following code snippet, for example, will go through each list element of `cd.list` and count how many variables are present, and create a key showing their order. Then, at the end, we look for all the *unique* components of our `numberOfElements` and `namesForElements`. If we have more than one, it's a sign we need to correct our query or adjust our expectations.

```r
numberOfElements <- c()
namesForElements <- c()
for (l in cd.list) {
  numberOfElements <- c(numberOfElements, length(l))
  namesForElements <- c(namesForElements, paste(names(l), collapse = ":"))
  for (e in l) {
    valuesInElement <- length(e)
    if (valuesInElement != 1) {
      break
    }
  }
  if (valuesInElement != 1) {
    break
  }
}
paste(unique(numberOfElements), unique(namesForElements), sep = " | ")
```

Exercise 2: Fixing a known misordered data.frame
Add this next line to `cd.list` and rerun the troubleshooting code section. What changed in the output from the *paste* command? How would you fix the list so the columns are correct while still using the *matrix* method described in Exercise 1?

```
cd.list[[27]] <- list(
  ARTIST = list("Benny Andersson", "Tim Rice", "Bjorn Ulvaeus"),
  TITLE = list("Chess"),
  COUNTRY = list("Sweden"),
  COMPANY = list("RCA Victor"),
  PRICE = list("19.95"),
  YEAR = list("1984")
)
```

Exercise 3: Preventing a misordered data.frame
Propose another way to convert the xml file so it automatically adjusts for inconsistent ordering of the keyword pairs.

```
cd <- read_xml("https://www.w3schools.com/xml/cd_catalog.xml")
```

Summary

- **xml2** for read/writing xml files
- **jsonlite** for read/writing json files
- troubleshoot the returned query result to catch
  - multiple values where a single was expected
  - the same key names are in every entry
  - the key names are always in the same order (or the conversion code robustly adjusts to handle this)
  - numbers stored as strings
  - strings stored as factors