Data Science with R

Part VI: Data Input/Output

Raphael Schleutker

March 27, 2020

If it's not written, it never happened. If it is written, it doesn't matter what happened.
— Sercan Leylek

Table of contents

A primer on files and connections

A primer on files and connections

So far we only worked with artificially created data or sample data sets from within R. In reality, we work with data that is not yet available in R. The data sources are diverse:

- CSV files.
- Other text files.
- Special file formats like .xlsx (or from other software).
- Databases.
- etc.

A primer on files and connections

Let us first focus on simple files that are stored on our local computer. We want some general interface that we can always use to read in data. This is challenging. Some files are so big that they do not fit entirely in the main memory. So what could a general way of working with files look like?

In the following we will see, how R (and other programming languages) handles files internally. It is very, very useful for a good understanding. However, there are a lot of convenience functions that we can use in most situations so we do not have to worry about this stuff.

R has the concept of connections. Connections in this sense does not necessarily mean an internet connections. We also talk about connections when R opens a file (connects to it).

A primer on files and connections

We can create a connection to a file by using the `file` function.

```
connection <- file("./data/03-18-2020.csv")
connection


# A connection with
# description "./data/03-18-2020.csv"
# class       "file"
# mode        "r"
# text        "text"
# opened      "closed"
# can read    "yes"
# can write   "yes"
```

Initially, the file is closed. We could still go to the file explorer and move or delete the file.

A primer on files and connections

We can open this connection using open. open takes an argument called open that specifies how to open the file (the mode).

```
open(connection, open = "r")     # r=reading, w=writing, a=appending
connection


# A connection with
# description "./data/03-18-2020.csv"
# class       "file"
# mode        "r"
# text        "text"
# opened      "opened"
# can read    "yes"
# can write   "no"
```

The file is now blocked on the computer. When we try to move or delete it, our operating system will tell us that it is currently opened in R.

A primer on files and connections

Now we can start to read from the file. Initially, the pointer is positioned at the beginning of the file. Text files are usually read line-wise. When a line is read the pointer is moved to the next line.

```
readLines(connection, 1) # Reads a single line from the file.

# [1] "Province/State,Country/Region,Last Update,Confirmed,Deaths,Recovered,Lat

readLines(connection, 1) # Pointer was moved. Reads next line.

# [1] "Hubei,China,2020-03-18T12:13:09,67800,3122,56927,30.9756,112.2707"
```

We can store these strings in a variable and process it further.

Every time we read a line or several lines, the pointer in the file is moved to the next line. Principally, we could reset the pointer. This can cause some problems, so I will not show you how ;)

A primer on files and connections

When we are done with a connection, we can close it. Not closing a file connection is problematic, cause the file remains blocked for any other action.

```
close(connection)
connection

# A connection, specifically, 'file', but invalid.
```

Closing a connection does not only close it but also destroys it. It can not be used again. Instead we have to create a new connection to the file (that we can store in the same variable, of course).

```
open(connection)

# Error in open.connection(connection):  invalid connection
```

A primer on files and connections

Connections can also be used to write strings to files.

```
connection <- file("./data/output.txt", "w")
writeLines(c("Line 1", "Line 2"), connection)
close(connection)

connection <- file("./data/output.txt", "r")
readLines(connection)

# [1] "Line 1" "Line 2"

close(connection)
```

Note, that we can also set the mode for opening directly in the `file` function. This opens the file directly.

A primer on files and connections

Such file connections can also be used for binary files (like .xlsx) but the processing is more difficult because such files contain raw bytes and not human readable text (the specification for how excel files are structured comprises more than 1100 pages).

Connections can also be used to connect to compressed files or files on the internet via the URL or to databases using the IP address of the database and the login credentials.

Depending on the resource to which we connect, the connection object is different and only some functions can work with them. For instance, it is not possible to use `readLines` for connections to databases.

A primer on files and connections

R has many convenience functions that handle all this stuff: Creating and opening a connection, reading the contents, processing it and creating e.g. a data frame. So usually, we only have to provide the file name and R does the rest. We will see some of these functions in the following.

Why is it anyways important to understand how this works? One very important reason: Some files are several gigabytes big. They do not fit completely into memory. But even for those files we can establish a file connection, read each line separately and process it. At every moment only the current line is read into memory (this is very much like FIJIs virtual stacks, where only the current frame is read into memory).

A primer on files and connections

Imagine the following: You have a really, really big file (several million lines) with some genomic data about drosophila but you are only interested in a few thousand lines from the file. You can establish two connections: One for the huge file and one for the extracted data. You read in the huge file line by line. If the current line is interesting for you, you write it to the target file. Depending on your computer it will still take several minutes to process the whole data but you end up with a new file that only contains the interesting data and that is small enough to be easily handled.

A primer on files and connections

Other reasons: Some data are crappy and you have to check for each line separately if it is properly formatted.

Or you are working with data for which no function exists.

Tabular text files

Tabular text files

Text files often contain tabular data. The precise structure might differ from case to case, e.g. the delimiter often varies between different sources.

Text files

Text files on any operating system only contain text data. It is completely irrelevant what the extension of such a file is, you can even invent one. File extensions for text files are only helpful for humans to guess what kind of data the file contains and for the operating system to determine a standard program to open the file. .csv files are often automatically opened in excel. But one can rename the file to .txt and it is still valid.

Tabular text files

The basic function to read in text files is `read.table`. It takes a connection or the file name plus some additional arguments and creates a data frame.

```
our_data <- read.table(
        "./data/03-18-2020.csv",
        header = TRUE, # Use first line in file as column names.
        sep = ",", # Delimiter between columns.
        quote = "\"", # Quoting character.
        stringsAsFactors = FALSE # Do not convert strings to factors by default.
)
```

`read.table` has many arguments. Study the help page to see which and if you need them.

Tabular text files

`stringsAsFactors`

One argument that you will often see when reading text files is `stringsAsFactors = FALSE`. Factors are the way R represents categorical variables. A factor is a special form of an integer vector where each integer has a name. By default, R converts all strings from text files into factors. This is often not what we want. So we will usually set this argument to `FALSE`.

With the release of R 4.0.0 the default behavior will change and strings will not longer be converted to factors automatically.

Tabular text files

```
str(our_data)


# 'data.frame': 284 obs. of  8 variables:
#  $ Province.State: chr  "Hubei" "" "" "" ...
#  $ Country.Region: chr  "China" "Italy" "Iran" "Spain" ...
#  $ Last.Update   : chr  "2020-03-18T12:13:09" "2020-03-18T17:33:05" "2020-
03-18T12:33:02" "2020-03-18T13:13:13" ...
#  $ Confirmed     : int  67800 35713 17361 13910 12327 9043 8413 3028 2626 249
#  $ Deaths        : int  3122 2978 1135 623 28 148 84 28 71 16 ...
#  $ Recovered     : int  56927 4025 5389 1081 105 12 1540 15 65 0 ...
#  $ Latitude      : num  31 41.9 32.4 40.5 51.2 ...
#  $ Longitude     : num  112.27 12.57 53.69 -3.75 10.45 ...
```

Tabular text files

`read.table` allows to precisely specify the format of the file. Often the format follows a known convention like CSV (comma separated values). For this, functions are available that are basically wrappers for `read.table` but set some arguments like the delimiter automatically.

```
our_data2 <- read.csv(
        "./data/03-18-2020.csv",
        stringsAsFactors = FALSE
)
str(our_data2)


# 'data.frame': 284 obs. of  8 variables:
#  $ Province.State: chr  "Hubei" "" "" "" ...
#  $ Country.Region: chr  "China" "Italy" "Iran" "Spain" ...
#  $ Last.Update   : chr  "2020-03-18T12:13:09" "2020-03-18T17:33:05" "2020-
03-18T12:33:02" "2020-03-18T13:13:13" ...
#  $ Confirmed     : int  67800 35713 17361 13910 12327 9043 8413 3028 2626 249
#  $ Deaths        : int  3122 2978 1135 623 28 148 84 28 71 16 ...
#  $ Recovered     : int  56927 4025 5389 1081 105 12 1540 15 65 0 ...
#  $ Latitude      : num  31 41.9 32.4 40.5 51.2 ...
#  $ Longitude     : num  112.27 12.57 53.69 -3.75 10.45 ...
```

Tabular text files

```
read.csv


# function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
#     fill = TRUE, comment.char = "", ...)
# read.table(file = file, header = header, sep = sep, quote = quote,
#     dec = dec, fill = fill, comment.char = comment.char, ...)
# <bytecode: 0x0000000014faec50>
# <environment: namespace:utils>
```

Tabular text files

We can use `write.table` to save data to a text file. It takes a data frame and a file name and saves the data frame to the file. Like for `read.table`, there are many arguments and often we would rather use `write.csv`.

```
write.csv(iris, "./data/iris.csv")
```

If there are no special reasons to do otherwise I'd like to encourage you to use `write.csv`.

Excel files

Excel files

R does not know natively how to read excel files. We need an additional package for this. There are several, one of which is `openxlsx`. It can be used for both, reading and writing from and to excel files.

```
install.packages("openxlsx", dependencies = TRUE)
```

Excel files

The main function we need from this package is `read.xlsx`. However, in order to make the functions from the newly installed package available, we have to load it first.

```
library("openxlsx")

my_data <- read.xlsx("./data/03-18-2020.xlsx")
str(my_data)

# 'data.frame': 284 obs. of  8 variables:
#  $ Province/State: chr  "Hubei" NA NA NA ...
#  $ Country/Region: chr  "China" "Italy" "Iran" "Spain" ...
#  $ Last.Update   : chr  "2020-03-18T12:13:09" "2020-03-18T17:33:05" "
03-18T12:33:02" "2020-03-18T13:13:13" ...
#  $ Confirmed     : num  67800 35713 17361 13910 12327 ...
#  $ Deaths        : num  3122 2978 1135 623 28 ...
#  $ Recovered     : num  56927 4025 5389 1081 105 ...
#  $ Latitude      : num  31 41.9 32.4 40.5 51.2 ...
#  $ Longitude     : num  112.27 12.57 53.69 -3.75 10.45 ...
```

Excel files

Apparently, some journals want the data for plots as excel sheets... To write data to excel, we can use the `write.xlsx` function.

```
write.xlsx(iris, "./data/iris.xlsx")
```

JSON: Hierarchically structured data

JSON: Hierarchically structured data

Some data cannot be adequately expressed in tables and thus cannot be stored in a csv file. For instance, FlyBase provides a file with the most important Gal4 drivers. Each record has a varying number of stocks, publications, tissues, and stages. There is no way to store this in a single table. But we can use a JSON file instead.

JSON files are text files like CSV. The difference is the way the data is structured in the files and accordingly how they have to be interpreted.

JSON: Hierarchically structured data


```
{
  "metaData": {
    "release": "fb_2018_06",
    "dataSource": "fb_2018_06_reporting",
    "dataProvider": "FlyBase"
  },
  "data": [
    {
      "driver": {
        "expression_desc_text": "Expression throughout ectoderm starting at embryonic stage 9.",
        "stocks": {
          "FBst0001774": "1774",
          "FBst0305166": "106499"
        },
        "major_tissues": {
          "FBbt00000111": "ectoderm"
        },
        "fbid": "FBal0040470",
        "transposons": {},
        "name": "Scer\\GAL4<up>69B</up>",
        "major_stages": {
          "FBdv00005289": "embryonic stage"
        },
        "pubs": {
          "FBrf0183875": "Gorfinkiel et al., 2005, Dev. Cell 8(2): 241--253",
          "FBrf0095439": "Kockel et al., 1997, Genes Dev. 11(13): 1748--1758",
          ...
```


Using the `jsonlite` package we can read in this data as a nested named list.

JSON: Hierarchically structured data

```
library("jsonlite")

gal4_data <- read_json("./data/fu_gal4_table_fb_2018_06.json")

gal4_data[["metaData"]]

# $release
# [1] "fb_2018_06"
#
# $dataSource
# [1] "fb_2018_06_reporting"
#
# $dataProvider
# [1] "FlyBase"

length(gal4_data[["data"]])

# [1] 236
```

JSON: Hierarchically structured data

```
gal4_data[["data"]][[1]]


# $driver
# $driver$expression_desc_text
# [1] "Expression throughout ectoderm starting at embryonic stage 9."
#
# $driver$stocks
# $driver$stocks$FBst0001774
# [1] "1774"
#
# $driver$stocks$FBst0305166
# [1] "106499"
#
#
# $driver$major_tissues
# $driver$major_tissues$FBbt00000111
# [1] "ectoderm"
#
#
# $driver$fbid
# [1] "FBal0040470"
```

JSON: Hierarchically structured data

```
gal4_data[["data"]][[1]][[1]][["pubs"]]


# $FBrf0183875
# [1] "Gorfinkiel et al., 2005, Dev. Cell 8(2): 241--253"
#
# $FBrf0095439
# [1] "Kockel et al., 1997, Genes Dev. 11(13): 1748--1758"
#
# $FBrf0187389
# [1] "Laugier et al., 2005, Dev. Biol. 283(2): 446--458"
#
# $FBrf0192322
# [1] "Verdier et al., 2006, BMC Dev. Biol. 6(1): 38"
#
# $FBrf0238311
# [1] "Tsoumpekos et al., 2018, J. Cell Biol. 217(3): 1033--1045"
#
# $FBrf0190943
# [1] "Sano et al., 2005, J. Cell Biol. 171(4): 675--683"
#
# $FBrf0208858
```

JSON: Hierarchically structured data

The same package also provides functions for exporting JSON files: Read the help pages ;-)

Some last general advice

Some last general advice

There are dozens/hundreds of different file types not to mention databases. Even for text files there are many, many different ways to structure them. BUT: If the data follows some convention then there is a package out there that can read the data (use Google or a less dubious search engine!).

The most important thing: You have to understand your data! You have to know how it is organized/structured. Only then you can decide what the best way is to work with them.

Some last general advice

Even more important than the most important thing: Never trust data of other people! There is ALWAYS something wrong with it, always! Reading the data in can already be difficult if the data is formatted crappily. But even when you manage to get your data into R, you have to check that it really is what you think. Otherwise you will run into trouble.

As a rule of thumb: Data scientists use about 80% of their time for reading and cleaning data. As soon as it is properly formatted the actual analysis is fast.