6. Getting Data In and Out of R

6.1 Reading and Writing Data

Watch a video of this section[1]

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (`inverse` of `dump`)
- `dget`, for reading in R code files (`inverse` of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

There are of course, many R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.

There are analogous functions for writing data to files

- `write.table`, for writing tabular data to text files (i.e. CSV) or connections
- `writeLines`, for writing character data line-by-line to a file or connection
- `dump`, for dumping a textual representation of multiple R objects
- `dput`, for outputting a textual representation of an R object
- `save`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- `serialize`, for converting an R object into a binary format for outputting to a connection (or file).

6.2 Reading Data Files with `read.table()`

The `read.table()` function is one of the most commonly used functions for reading data. The help file for `read.table()` is worth reading in its entirety if only because the function gets used a lot (run `?read.table` in R). I know, I know, everyone always says to read the help file, but this one is actually worth reading.

The `read.table()` function has a few important arguments:

[1] https://youtu.be/Z_dc_FADyi4

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are separated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset. By default `read.table()` reads an entire file.
- `comment.char`, a character string indicating the comment character. This defalts to `"#"`. If there are no commented lines in your file, it's worth setting this to be the empty string `""`.
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, should character variables be coded as factors? This defaults to `TRUE` because back in the old days, if you had data that were stored as strings, it was because those strings represented levels of a categorical variable. Now we have lots of data that is text data and they don't always represent categorical variables. So you may want to set this to be `FALSE` in those cases. If you *always* want this to be `FALSE`, you can set a global option via `options(stringsAsFactors = FALSE)`. I've never seen so much heat generated on discussion forums about an R function argument than the `stringsAsFactors` argument. Seriously.

For small to moderately sized datasets, you can usually call read.table without specifying any other arguments

```
> data <- read.table("foo.txt")
```

In this case, R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table.

Telling R all these things directly makes R run faster and more efficiently. The `read.csv()` function is identical to read.table except that some of the defaults are set differently (like the `sep` argument).

6.3 Reading in Larger Datasets with read.table

Watch a video of this section[2]

With much larger datasets, there are a few things that you can do that will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints
- Make a rough calculation of the memory required to store your dataset (see the next section for an example of how to do this). If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.

[2] https://youtu.be/BJYYIJO3UFI

- Set `comment.char = ""` if there are no commented lines in your file.
- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick an dirty way to figure out the classes of each column is the following:

```
> initial <- read.table("datatable.txt", nrows = 100)
> classes <- sapply(initial, class)
> tabAll <- read.table("datatable.txt", colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

In general, when using R with larger datasets, it's also useful to know a few things about your system.

- How much memory is available on your system?
- What other applications are in use? Can you close any of them?
- Are there other users logged into the same system?
- What operating system ar you using? Some operating systems can limit the amount of memory a single process can access

6.4 Calculating Memory Requirements for R Objects

Because R stores all of its objects physical memory, it is important to be cognizant of how much memory is being used up by all of the data objects residing in your workspace. One situation where it's particularly important to understand memory requirements is when you are reading in a new dataset into R. Fortunately, it's easy to make a back of the envelope calculation of how much memory will be required by a new dataset.

For example, suppose I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame? Well, on most modern computers double precision floating point numbers[3] are stored using 64 bits of memory, or 8 bytes. Given that information, you can do the following calculation

[3] http://en.wikipedia.org/wiki/Double-precision_floating-point_format

1,500,000 × 120 × 8 bytes/numeric	= 1,440,000,000 bytes
	= 1,440,000,000 / 2^{20} bytes/MB
	= 1,373.29 MB
	= 1.34 GB

So the dataset would require about 1.34 GB of RAM. Most computers these days have at least that much RAM. However, you need to be aware of

- what other programs might be running on your computer, using up RAM
- what other R objects might already be taking up RAM in your workspace

Reading in a large dataset for which you do not have enough RAM is one easy way to freeze up your computer (or at least your R session). This is usually an unpleasant experience that usually requires you to kill the R process, in the best case scenario, or reboot your computer, in the worst case. So make sure to do a rough calculation of memeory requirements before reading in a large dataset. You'll thank me later.

7. Using the `readr` Package

The `readr` package is recently developed by Hadley Wickham to deal with reading in large flat files quickly. The package provides replacements for functions like `read.table()` and `read.csv()`. The analogous functions in `readr` are `read_table()` and `read_csv()`. These functions are often *much* faster than their base R analogues and provide a few other nice features such as progress meters.

For the most part, you can read use `read_table()` and `read_csv()` pretty much anywhere you might use `read.table()` and `read.csv()`. In addition, if there are non-fatal problems that occur while reading in the data, you will get a warning and the returned data frame will have some information about which rows/observations triggered the warning. This can be very helpful for "debugging" problems with your data before you get neck deep in data analysis.

The importance of the `read_csv` function is perhaps better understood from an historical perspective. R's built in `read.csv` function similarly reads CSV files, but the `read_csv` function in `readr` builds on that by removing some of the quirks and "gotchas" of `read.csv` as well as dramatically optimizing the speed with which it can read data into R. The `read_csv` function also adds some nice user-oriented features like a progress meter and a compact method for specifying column types.

A typical call to `read_csv` will look as follows.

```
> library(readr)
> teams <- read_csv("data/team_standings.csv")
Parsed with column specification:
cols(
  Standing = col_integer(),
  Team = col_character()
)
> teams
# A tibble: 32 × 2
   Standing        Team
      <int>       <chr>
1         1       Spain
2         2 Netherlands
3         3     Germany
4         4     Uruguay
5         5   Argentina
6         6      Brazil
7         7       Ghana
8         8    Paraguay
9         9       Japan
10       10       Chile
# ... with 22 more rows
```