Reading and Writing to/from R

R functions and packages

- `read.table()` and friends
- `read.fwf()`
- `scan()`
- `readLines()`
- `"connections"`
- Packages
  - `"readr"`
  - `"XML"`

Data storage and formats (outside R)

At this point we are going to turn to reading data into R. We'll focus on doing these manipulations in R, but the concepts and tools involved are common to other languages, so familiarity with these in R should allow you to pick up other tools more easily.

R has the capability to read in a wide variety of file formats. Here are some of the common ones:

1. Flat—or plain—text files (ASCII files): data are often provided as simple text files. Often one has one record or observation per row and each column or field is a different variable. Such files can either have a special character used as a delimiter that separates the fields in each row, or they can have a fixed number of characters in each field (fixed-width format). Common filed delimiters are tabs, one or more spaces, commas, semicolons, and bars (\). Common file extensions are `.txt`, `.csv`, `.dat`, .tsv'. Metadata (information and description about the data) are often stored in a separate file.

2. In some contexts, such as textual data and bioinformatics data, the data may be in a text file with one piece of information per row, but without meaningful columns/fields.

3. Data may also be in text files in formats designed for data interchange between various languages, in particular XML or JSON. These formats are self-describing; namely the metadata is part of the file. The `"XML"` and `"jsonlite"` packages are useful for reading and writing from these formats.

4. You may be scraping information on the web, so dealing with text files in various formats, including HTML. The `"XML"` package is also useful for reading HTML.

5. Data may already be in a database or in the data storage of another statistical package (i.e. Stata, SAS, SPSS, etc). The `"foreign"` package in R has excellent capabilities for importing these types of files.

6. For Excel, there are also a handful of packages such as `"readxl"`, or `"XLConnect"` (among others), but you can just go into Excel and export a spreadsheet as a CSV file or the like and then read that into R. In general, it is best not to pass around data files as Excel or other spreadsheet format files because (1) Excel is proprietary, so someone may not have Excel and the format is subject to change, (2) Excel imposes limits on the number of rows, (3) one can easily manipulate text files such as CSV using UNIX tools, but this is not possible with an Excel file, (4) Excel files often have more than one sheet, graphs, macros, etc., so they are not a data storage format per se.

Reading data from text files into R

The main downside to working with datasets in R is that the entire dataset resides in memory, so R is not so good for dealing with very "large" datasets. Another common frustration is controlling how the variables are interpreted (numeric, character, factor) when reading data tables as data frames. Despite these issues, R provides a wide variety of options to read in data and text files in general.

Function `read.table()` and friends

To import data tables, the main function is `read.table()`. Associated with this function there are wrappers like `read.delim()`, and `read.csv()`, among others. All of these functions read in a dataset and will return a `"data.frame"` object.

If you check the documentation of `read.table()`, there are more than 20 parameters to play with. I'll mention some of the important ones:

- `file`: the name of the file (path name)
- `header`: whether the first row contains names of columns
- `sep`: single character indicating the field separator
- `dec`: character used for decimal points
- `row.names`: vector of row names (can be the number giving the column position used for row names)
- `col.names`: vector column names (optional)
- `na.strings`: character vector of strings which are interpreted as `NA`
- `colClasses`: character vector with the data type of each column
- `nrows`: number of rows to read
- `skip`: number of lines to skip before reading
- `stringsAsFactors`: whether characters are converted to factors

The most difficult part of reading in files with `read.table()` (and friends) has to do with how R determines the classes of the fields that are read in. One issue with the read-table functions is that character and numeric fields are sometimes read in as factors. The reading-table functions try to read fields in as numeric, and if it finds non-numeric and non-NA values, it reads in as a factor (Trivia: What is the reason of this behavior?).

There are a number of arguments that allow the user to control the behavior of the assigned classes. One important argument is `stringsAsFactors`, which has a default value of `TRUE`; so by default, all string values will be converted as factors. Some of us believe this is the wrong default. When reading a data table, you should keep character vectors as characters. Then, if you want one or more columns to be treated as factors, you should explicitly convert them as factors.

Another interesting argument is `colClasses`, which allows you to specify the data type for each column. Whenever possible use `colClasses`, especially if you are working with a relatively large data set. Not only you save time importing the data set, but it will also let you manipulate data with more control. **Protip:** you can avoid reading in one or more columns by specifying `"NULL"` as the column class for those columns to be omitted.

Let's see some examples. You can find the datasets in the `data/` directory from the github repository.

```r
# check what directory R is looking at
getwd()

# heimport data
dat <- read.table(
  "../data/RTADataSub.csv",
  sep = ",",
  header = TRUE)

# check its structure
str(dat, vec.len = 1)

# some inspection (what's happening?)
summary(dat[,1:5])
```

Let's convert those `x`'s into missing values, and tell R to stop converting strings as factors:

```r
# another option
dat2 <- read.table(
  "../data/RTADataSub.csv",
  sep = ",",
  header = TRUE,
  na.strings = c("NA", "x"),
  stringsAsFactors = FALSE)
```

```r
unique(dat2[ ,2])

which(dat[ ,2] == '')
dat2[which(dat[ ,2] == '')[1], ]
```

Another recommendation is to take a look at the input file in the shell or in an editor before reading into R to catch issues in advance like:

- codification of missing values other than `NA`
- character delimiter (e.g. comma, tab, semicolon)
- character of decimal point (e.g. dot, comma, other?)
- type of line endings

Likewise, you could also use some of the Unix utilities (e.g. `head`, `tail`, `wc`, `cut`, `sort`, `unique`, `grep`) to inspect the data files before importing them in R.

Command and Tab-delimited files

Fro the common cases of reading in data whose fields are separated by commas or tabs,R provides three wrappers of `read.table()`:

- `read.csv()` (comma separated)
- `read.csv2()` (semicolon separated)
- `read.delim()` (tab-separated)

All these functions accept any of the optional arguments to `read.table()`, and they are often more convenient than using `read.table()` and setting the appropriate arguments manually.

Fixed-Width files

Sometimes input files are stored with no delimiters between the values, but with each variable occupying the same columns on each line of input. In these cases, you can use the function `read.fwf()`. In addition to the specified data `file`, the other main argument of `read.fwf()` is `widths`. This argument can be a vector containing the widths of the fields to be read, using negative numbers to indicate "columns" to be skipped. If the data for each observation occupies more than one line, `widths` can be a list of as many vectors as there are lines per observation.

Function scan()

Another useful function to import data in R is scan(). This function can read data from the console, or from a file. As a matter of fact, scan() is the function called by read.table() and related functions to read in the values.

By default, scan() expects all of its input to be numeric data; but this can be overridden with the argument waht, specifying the type of data that scan() will read.

One of the most common uses of scan() is to read in data matrices. Since scan() returns a vector, you can embed it inside a call to the matrix() function. Assuming that you have a file numbers.txt with 12 numbers (from 1 to 12, one number per row), you can import it in a matrix form as:

```r
# read by columns
M1 <- matrix(scan('numbers.txt'), nrow = 3, ncol = 4)

# read by rows
M2 <- matrix(scan('numbers.txt'), nrow = 3, ncol = 4, byrow = TRUE)
```

Refer to the data in file cpds.csv. The first 52 rows (except the header) has to do with data from Australia. Say you want to read columns 3, 5 and 6, and generate a numeric matrix. To achieve this task with scan(), there are various arguments you need to specify: what lines to read, the character delimiter, and the argument what. In order to skip columns while reading in data with scan(), a type of NULL should be used in the list passed to the what argument. To specify the numeric values, you can use a value of 0.

```r
# data for Australia
aus <- scan(
  file = '../data/cpds.csv',
  skip = 1,
  nlines = 51,
  sep = ",",
  what = c(list(NULL), list(NULL), f3=0, list(NULL), f5=0, f6=0)
)

australia <- cbind(aus$f3, aus$f5, aus$f6)
```

scan() is also useful if your file is free format: it's not one line per observation, but just all the data one value after another.

Function readLines()

If the file is not nicely arranged by field (e.g. if it has ragged lines), we'll need to do some more work. The function readLines() will read the file contents into a character vector, with as many elements as read lines.

R package "readr"

An interesting alternative to the base functions in R for reading tabular data can be found in the R package "readr":

- https://github.com/hadley/readr
- https://blog.rstudio.org/2015/04/09/readr-0-1-0/
- https://blog.rstudio.org/2015/10/28/readr-0-2-0/
- `read_table()`
- `read_csv()` reads a csv file (similar to `read.csv()`)
- `read_tsv()` reads a tab separated value file
- `read_delim()` (similar to `read.tsv()`)
- `read_fwf()` reads a fixed-width file (similar to `read.fwf()`)
- `read_file()` reads whole file into a single string
- `read_lines()` reads lines into a vector (similar to `readLines()`)
- `read_log()` reads a web log file

Connections

Now that we've seen some of the most common functions to read in different types of data files, it's time to discuss a less known topic that is fundamental for importing and exporting data.

R allows you to read in not just from a file but from a more general construct called a **connection**. A connection is a special type of object in R that is used (behind the scenes) when we are reading and writing data to/from R. Connection objects allow R functions—such as `read.table()`, `scan()`, `readLines()`, `writelines()`, `cat()`—to read and interpret data from outside of R, when the data can come from a variety of sources. Data sources can be a local file, a location on the web, or an R character vector.

So what exactly is a connection? A connection is a reference to a data stream.

To create a `"connection"` object, R uses various types of functions (the following is not a complete list of connections):

- `file()`: files on the local system
- `gzfile()`: local gzipped file
- `unz()`: local zip archive

- `bzfile()`: local bzipped file
- `url()`: remote file read via http

Files are the most common types of connections. `"file"` connections are either specify by their path in the filesystem or created as temporary files. There are three classes of connections that extend files to include compression on input or output. They differ in the kind of compression done:

- `gzfile`: corresponds to the shell command `gzip`
- `bzfile`: corresponds to the shell command `bzip2`
- `unz`: read a `.zip` file

About connection objects

When you use one of the connection functions to create a connection object, it simply defines the object; it does not automatically open the object. Connections have a state of being *open* or *closed*. While a connection is open, successive input operations start where the previous operation left off. Similarly, succssive output operations on an open connection append bytes just after the last byte resulting from the previous operation. There are two common problems when working directly with connections: not explicitly closing them (when reading data), and not explicitly opening them (when writing). Luckily, you will rarely be using connections directly (at a low-level).

The recommended rules for functions that read or write from connections are:

- If the connection is initally closed, open it and close it on exiting from the function.
- If the connection is initially open, leave it open after input/output operations.

Consider the following piece of code, which writes the elements of a character vector `some_text`, one element per line (via a file connection) to the file `mytext.txt` in the local working directory:

```
# create a connection to a file
txt <- file("./mytext.txt")

# write contents to the file
writeLines(text = some_text, con = txt)
```

Calling `file()` just creates the connection object but it does not open it. The function `writeLines()` is the one that opens the connection, writes the content to the file `mytext.txt`, and then closes the connection on exiting.

When to explicitly use a connection?

When you call `read.table()` and pass it the name of a file to the `file` argument, R internally creates a `connection` that will be used to open the file, read the values, and close the file at the end of the function invocation. In the usual case you don't really have to explicitly open the file and close it after reading (or writing) its contents.

Perhaps the most common case when you will explicitly define a connection is when reading a file in pieces, usually through the `readLines()` function. A typical example is when you need to read a very large file that either does not fit into memory, or even if it does, reading the entire file will be very slow or inefficient.

Another common example is when you need to read a file in which each line requires various preprocessing steps that will put the data in the right shape/format.

The standard recipe to read a file in pieces follows these steps:

- use a connection function to **open** the connection in *read* mode
- use a loop (e.g. `while`, `repeat` or `for`) to read each line
- do something with the read line
- usually check whether the end of file has been reached
- use `close()` to close the connection at the end of the process

```r
# create connection (in read mode)
infile <- file("some-file.txt", open = "r")

# loop over the lines of the file
while (TRUE) {
  # read one line at a time
  one_line <- readLines(infile, n = 1)
  # are there any more lines to read?
  if (length(one_line) == 0) {
    break
  } else {
    # do some processing (e.g. print)
    print(one_line)
  }
}
# close connection when done
close(infile)
```

Reading HTML files

Let's see a brief example of reading in HTML files. One lesson here is not to write a lot of your own code to do something that someone else has probably already written a package for.

Unfortunately, there are some issues with dealing with https that we need to work around, rather than directly using `readHTMLTable()` as can be done with http. So we need to use `curl()` to get the HTML via https and then use the XML package functionality for parsing the HTML.

```
library(XML)

URL <- "https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population"

# this won't work
tbls <- readHTMLTable(URL)
```

```
## Error: XML content does not seem to be XML: 'https://en.wikipedia.org/wiki/List_of_co
```

Trying to use `readHTMLTable()` directly with an https resource will fail. Instead, we can do two things. One option is to download a local copy, and then read it with `readHTMLTable()`:

```
# download html file in the working directory
download.file(url = URL, destfile = "population.html")

tbls <- readHTMLTable("population.html")
```

The other option is to read the contents as a character vector with `curl()` inside `readLines()`, and then call `readHTMLTable()`:

```
library(curl)

html <- readLines(curl(URL))
tbls <- readHTMLTable(html)
```

```
## Warning: closing unused connection 5 (https://en.wikipedia.org/wiki/
## List_of_countries_and_dependencies_by_population)
```

```
# number of rows in available tables
sapply(tbls, nrow)
```

```
## $`NULL`
## NULL
##
## $`NULL`
## NULL
##
```

```
## $`NULL`
## [1] 250
##
## $`NULL`
## [1] 24
```

Knowing the number of the desired table, we can specify the `which` arguments of `readHTMLTable()` like so:

```
pop <- readHTMLTable(html, which = 3)
head(pop)
```

```
##   Rank Country (or dependent territory)    Population               Date
## 1    1                    China[Note 2] 1,378,780,000 September 16, 2016
## 2    2                            India 1,330,780,000 September 16, 2016
## 3    3            United States[Note 3]   324,492,000 September 16, 2016
## 4    4                        Indonesia   260,581,000       July 1, 2016
## 5    5                           Brazil   206,663,000 September 16, 2016
## 6    6                         Pakistan   194,269,000 September 16, 2016
##   % of world\npopulation                    Source
## 1                  18.8% Official population clock
## 2                  18.1% Official population clock
## 3                  4.42% Official population clock
## 4                  3.55%       Official projection
## 5                  2.81% Official population clock
## 6                  2.64% Official population clock
```