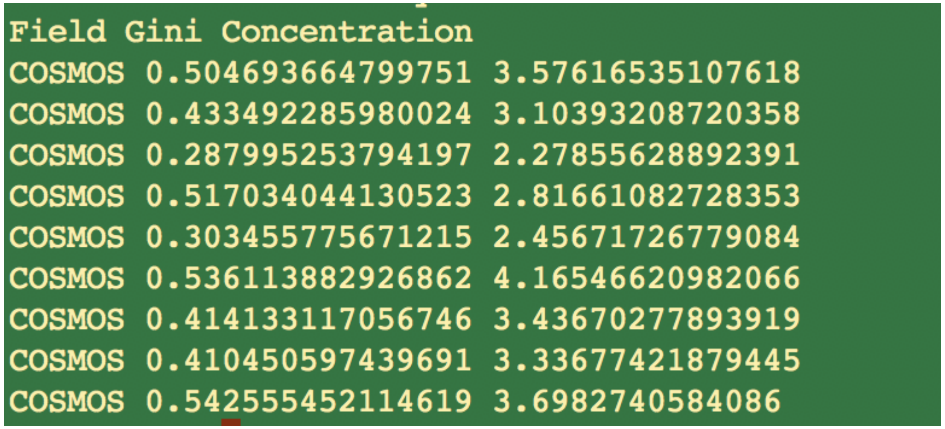
Basic Data I/O

36-600

Week 3 Tuesday – Fall 2022

Formatted Text Files: Base R

If your data file is an ASCII (i.e., human-readable) text file . . .



```
Field Gini Concentration
COSMOS 0.504693664799751 3.57616535107618
COSMOS 0.433492285980024 3.10393208720358
COSMOS 0.287995253794197 2.27855628892391
COSMOS 0.517034044130523 2.81661082728353
COSMOS 0.303455775671215 2.45671726779084
COSMOS 0.536113882926862 4.16546620982066
COSMOS 0.414133117056746 3.43670277893919
COSMOS 0.410450597439691 3.33677421879445
COSMOS 0.542555452114619 3.6982740584086
```

then you will generally use `read.table()` or `read.csv()`.

- `read.table()`: used with data whose values are separated by spaces
- `read.csv()`: used when you have "comma-separated values"

Other alternatives include `read.delim()` and `scan()` (for more general formats), along with `readLines()` (which simply reads each line of a file into an element of a character vector).

We won't go into binary (i.e., non-human-readable) files, as how to read in their data can be highly domain-specific. To get a handle on how you might proceed if a binary reader has not already been coded for you, see the help page for `connections`.

Formatted Text Files: Base R

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
           row.names, col.names, as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)

read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
```

Beware the arguments!

- `header` provides column names, and is `FALSE` by default for `read.table()`
- `stringsAsFactors` is default `TRUE` and will lead to your character string vector being treated as a factor variable
- `na.strings` is `NA` by default, but beware: domain scientists use many symbols and/or numbers to indicate that data are missing (like -99)

Useful arguments:

- `skip`: if there are lines of metadata preceding the first line of data, use `skip` to skip over them

Side effects:

- if your column name has spaces, `read.table()` and `read.csv()` will fill them in with periods

Formatted Text Files: Base R

What if my text file contains some columns that should be treated as representing factor variables, and some that should be treated as representing character strings, etc.?

```
file.url <- url("http://www.stat.cmu.edu/~pfreeman/mixed_data.csv")
readLines(file.url)
```

```
## [1] "name,height,favorite ice cream flavor" "Fred,68,Strawberry"
## [3] "Wilma,64,Chocolate"
```

Here, the columns represent character, numeric, and factor variables, respectively.

The key argument to implement is `colClasses`:

```
df <- read.csv(file.url,colClasses=c("character","numeric","factor"))
df$name  # remember the dollar sign?
```

```
## [1] "Fred"  "Wilma"
```

```
df$favorite.ice.cream.flavor
```

```
## [1] Strawberry Chocolate
## Levels: Chocolate Strawberry
```

Formatted Text Files: Base R

But...what about this?

```
file.url <- url("http://www.stat.cmu.edu/~pfreeman/weird_data.csv")
readLines(file.url)
```

```
## [1] "|1|2|3|4|-99|"   "|-99|5|6|7|-99|"
```

- first, `read.csv()` doesn't require commas...you can specify the separator with the `sep` argument
- second, let's presume that `-99` is the data preparer's way of saying "these data are missing"...`R` doesn't know that, so we use the `na.strings` argument
- third, we don't need no stinkin' header

```
(df <- read.csv(file.url,header=FALSE,sep="|",na.strings="-99"))
```

```
##   V1 V2 V3 V4 V5 V6 V7
## 1 NA  1  2  3  4 NA NA
## 2 NA NA  5  6  7 NA NA
```

But we're not quite done yet...the number of columns here is wrong.

Formatted Text Files: Base R

`R` still thinks there were data to the left of the first "`|`" and to the right of the last "`|`". We can fix this:

```
(df <- df[,-c(1,ncol(df))])  # you don't know how to do this yet, mind you
```

```
##   V2 V3 V4 V5 V6
## 1  1  2  3  4 NA
## 2 NA  5  6  7 NA
```

Note:

- what you've input is a *data frame* (hence the `df`)
- if there is no header, `R` defaults to `V1`, `V2`, etc.
- when you print out a data frame, row names are provided (here: numbers)…the pseudo-first column with the `1` and `2` are row names, not actual data
- you can create names on the fly using `names()` and `rownames()`

```
names(df) = c("first","second","third","fourth","fifth")
rownames(df) = c("one","two")
df
```

```
##     first second third fourth fifth
## one     1      2     3      4    NA
## two    NA      5     6      7    NA
```

Formatted Text Files: readr

Faster alternatives to base `R` functions are provided by the `readr` package, which is part of the tidyverse.

```
suppressMessages(library(tidyverse))
```

- `read.table()` -> `read_table()` and `read.csv()` -> `read_csv()`
- **NOTE:** `read_table()` can do a bad job of parsing data; consider `read_delim()` instead
- keeps variable names as is (no introduced periods)
- `stringsAsFactors` is `FALSE` by default, but you can explicitly define the data type for each column using the `col_types` argument
- reads data into a tibble, which you may (or may not) want to cast to a data frame

Note that in my own life, I use the base `R` functions, because this last point — the reading of data into a tibble object instead of a data frame object — tends to make the use of `readr` more of a pain than it is worth. (Your mileage may vary.)

Formatted Text Files: readr

```
library(readr)
file.url <- url("http://www.stat.cmu.edu/~pfreeman/GalaxyStatistics.txt")
read_delim(file.url,delim=" +") %>% head(3)
```

```
##

Rows: 8358 Columns: 1
##  [36mℹ [39m Rendering  [34m [34mNotes_03T.Rmd [34m [39m into  [32m [32mNotes_03T.html [32m [39m

##

##  [36mℹ [39m Rendering  [34m [34mNotes_03T.Rmd [34m [39m into  [32m [32mNotes_03T.html [32m [39m

── Column specification ────────────────────────────────────────────────────────
##  [36mℹ [39m Rendering  [34m [34mNotes_03T.Rmd [34m [39m into  [32m [32mNotes_03T.html [32m [39m


Delimiter: " +"
## chr (1): Field Gini Concentration
##
##  [36mℹ [39m Rendering  [34m [34mNotes_03T.Rmd [34m [39m into  [32m [32mNotes_03T.html [32m [39m

##

##
##  [36mℹ [39m Rendering  [34m [34mNotes_03T.Rmd [34m [39m into  [32m [32mNotes_03T.html [32m [39m

ℹ Use `spec()` to retrieve the full column specification for this data.
##  [36mℹ [39m Rendering  [34m [34mNotes_03T.Rmd [34m [39m into  [32m [32mNotes_03T.html [32m [39m

ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
##  [36mℹ [39m Rendering  [34m [34mNotes_03T.Rmd [34m [39m into  [32m [32mNotes_03T.html [32m [39m
```

Reading Excel Files: readxl

`readxl` is another tidyverse-related package for reading data directly from `Excel` spreadsheets with either `xls` or `xlsx` extensions. (Note: it does not yet, by all appearances, cleanly load non-local files!)

```r
library(readxl)
readxl_example()
```

```
##  [1] "clippy.xls"    "clippy.xlsx"   "datasets.xls"  "datasets.xlsx" "deaths.xls"    "deaths.xlsx"   "geometry.xls
##  [8] "geometry.xlsx" "type-me.xls"   "type-me.xlsx"
```

```r
(tbl <- read_excel(readxl_example("clippy.xls")))
```

```
## # A tibble: 4 × 2
##   name                 value
##   <chr>                <chr>
## 1 Name                 Clippy
## 2 Species              paperclip
## 3 Approx date of death 39083
## 4 Weight in grams      0.90000000000000002
```

You can specify four column types: `skip`, `numeric`, `date`, and `text`.

Reading Excel Files: readxl

For instance, `skip` allows you to ignore a column:

```
# ignore second column
(tbl <- read_excel(readxl_example("clippy.xls"),col_types=c("text","skip")))
```

```
## # A tibble: 4 × 1
##   name
##   <chr>
## 1 Name
## 2 Species
## 3 Approx date of death
## 4 Weight in grams
```

Note that in my own life, I find that it is easier to save spreadsheet contents to a `.csv` file and then use `read.csv()`. I'd advise that you do this. Again, your mileage may vary.

Writing Files: Base R

As you might expect, `read.table()` and `read.csv()` have analogous write functions: `write.table()` and `write.csv()`. The two main arguments to look out for are

- `quote`: default `TRUE`…it puts double quotes around your column (and row) names. Set this to `FALSE`.
- `row.names`: default `TRUE`…which means, if you don't have row names in your data frame, you'll have `"1"`, `"2"`, etc. as the row names in your output. Also set this to `FALSE` if you don't have row names already.

Writing Files: readr

Surprisingly, there is no `write_table()` function in `readr`; one can use `write_delim()` instead:

```
write_delim(df,"./df.txt") # write the data frame df to the local file df.txt
```

By default, there are no quotes around the column names and no row names generated from the aether!

There is a `write_csv()` function for comma-separated values.

For `Excel` spreadsheets: use `write_excel_csv()`.

Storing R Objects: Save and Load

One can save R objects (vectors, data frames, etc.) in a binary format, so as to be loaded later:

```
x <- 5
y <- list(a=1:2,b=TRUE)
save(x,y,file="tmp.Rdata")
rm(x,y)
gc()
```

```
##            used  (Mb) gc trigger  (Mb) limit (Mb) max used  (Mb)
## Ncells  2607985 139.3    4428582 236.6         NA  4428582 236.6
## Vcells 17958050 137.1   31189230 238.0      16384 31189230 238.0
```

```
load("tmp.Rdata")
x
```

```
## [1] 5
```

```
y
```

```
## $a
## [1] 1 2
##
## $b
## [1] TRUE
```

Storing R Objects: Save and Load

Note the following:

- the suffixes `Rdata` and `Rda` are interchangable
- there is a `saveRDS()` function which saves *one* `R` object in a file with suffix `Rds`…this object may be given a new name when read in with `readRDS()`