# Reading Delimited Data

As one of our goals is to read in and wrangle data, we need to learn how to effectively take raw data (data not in R) and bring it into R. For most of our data sources, we'll store the data as a data frame (usually a tibble). For some types of data we'll need to read it in as a character string or as a list and then parse it with R.

## Data Formats

Data comes in many formats such as

- 'Delimited' data: Character (such as ',' , '>', or [' ']) separated data
- Fixed field data
- Excel data
- From other statistical software, Ex: SPSS formatted data or SAS data sets
- From a database
- From an Application Programming Interface (API)

As with many tasks in R, there are many ways to read in data from these sources.

- We could stick with `BaseR` (use functions like `read.csv()`)
- Use functions from a particular ecosystem (`tidyverse` or `data.table`)

We'll use the `tidyverse` due to its popularity and ease of functionality.

- Make sure `tidyverse` package is installed (this can take a while)

```
install.packages("tidyverse")
```

- Load the library into your current session

```
library(tidyverse)
```

```
Warning: package 'tidyverse' was built under R version 4.1.3

-- Attaching packages --------------------------------------- tidyverse
1.3.1 --

v ggplot2 3.5.1     v purrr   1.0.1
v tibble  3.2.1     v dplyr   1.1.2
v tidyr   1.3.0     v stringr 1.5.0
v readr   2.1.2     v forcats 0.5.1

Warning: package 'tibble' was built under R version 4.1.3

Warning: package 'tidyr' was built under R version 4.1.3

Warning: package 'readr' was built under R version 4.1.3

Warning: package 'purrr' was built under R version 4.1.3
```

```
Warning: package 'dplyr' was built under R version 4.1.3

Warning: package 'stringr' was built under R version 4.1.3

Warning: package 'forcats' was built under R version 4.1.3

-- Conflicts ---------------------------------------
tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

- You can see this loads in the eight core packages mentioned previously. The warnings can easily be ignored but we should take care with the conflicts. We've overwritten some functions from `BaseR`. Recall, we can call those functions explicitly if we'd like (`stats::filter()`).

## Locating Files

- Once our library is loaded, check `help(read_csv)` in your console. This brings up help for a suite of functions useful for reading in **delimited** data.
- Focus on *file* argument as everything else has defaults. Notice a *path to a file, a connection, or literal data* must be given.

Before we start reading in data, let's recap how R finds files.

- We can give a *full path name* to the file
  - ex: `C:/Users/jbpost2/Documents/Repos/ST-558/datasets/`
  - ex: `C:\\\\Users\\\\jbpost2\\\\Documents\\\\Repos\\\\ST-558\\\\datasets`
- Full path names are not good to use generally!
  - If you share your code with someone else, they don't have the same folder structure, username, etc.
  - Instead, use a *relative* path. That is, a path from `R`'s current working directory (the place it looks by default)
- It is recommend to do everything in an R project!
  - When you create an R project, you might note that it gets associated with a directory (or folder or repo). That folder is what the project uses as the working directory.
  - **You should try to always use relative paths from your project's working directory.**
  - **Note: When you render a `.qmd` (or `.Rmd`) file, the working directory for that rendering is the folder in which the `.qmd` (or `.Rmd`) file lives in.**

# Plan

- Go though examples of reading different types of data raw data

| Type of file | Package | Function |
|---|---|---|
| Delimited | `readr` | `read_csv()`, `read_tsv()`,`read_table()`, `read_delim(...,delim = ,...)` |
| Excel (.xls,.xlsx) | `readxl` | `read_excel()` |
| SPSS (.sav) | `haven` | `read_spss()` |
| SAS (.sas7bdat) | `haven` | `read_sas()` |

## Reading CSV Files

Let's start by considering a comma separated value (or CSV) file. This is a common basic format for raw data in which the delimiter is a comma ( `,` ).

Suppose we want to read in the file called `bikeDetails.csv` available at: https://www4.stat.ncsu.edu/~online/datasets/bikeDetails.csv

- We can download the file and store it locally, reading it in from there
- Or, for this type of file, we can also read it directly from the web!

We'll use the `read_csv()` function from the `readr` package. The inputs are:

```
read_csv(
  file,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  name_repair = "unique",
  num_threads = readr_threads(),
  progress = show_progress(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = should_read_lazy()
)
```

We really only need to specify the `file` argument but we see there are a few others that might be useful. We'll cover some important arguments shortly. Let's start with a basic call and see how it works:

```
bike_details <- read_csv("https://www4.stat.ncsu.edu/~online/datase
```

```
Rows: 1061 Columns: 7
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (3): name, seller_type, owner
dbl (4): selling_price, year, km_driven, ex_showroom_price

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

```
bike_details
```

```
# A tibble: 1,061 x 7
   name        selling_price  year seller_type owner km_driven
ex_showroom_price
   <chr>               <dbl> <dbl> <chr>       <chr>     <dbl>
<dbl>
 1 Royal Enfi~        175000  2019 Individual  1st ~       350
NA
 2 Honda Dio           45000  2017 Individual  1st ~      5650
NA
 3 Royal Enfi~        150000  2018 Individual  1st ~     12000
148114
 4 Yamaha Faz~         65000  2015 Individual  1st ~     23000
89643
 5 Yamaha SZ ~         20000  2011 Individual  2nd ~     21000
NA
 6 Honda CB T~         18000  2010 Individual  1st ~     60000
53857
 7 Honda CB H~         78500  2018 Individual  1st ~     17000
87719
 8 Royal Enfi~        180000  2008 Individual  2nd ~     39000
NA
 9 Hero Honda~         30000  2010 Individual  1st ~     32000
NA
10 Bajaj Disc~         50000  2016 Individual  1st ~     42000
60122
# i 1,051 more rows
```

Notice the fancy printing! As we read the data in with a function from the `tidyverse`, we have our data in the form of a `tibble` (special data frame).

Aside from the special printing, `tibble`s have one other important difference from data frames: **they do not coerce down to a vector when you subset to only one column using `[`**

```
        bike_details[1:10,1]
```

```
# A tibble: 10 x 1
   name
   <chr>
 1 Royal Enfield Classic 350
 2 Honda Dio
 3 Royal Enfield Classic Gunmetal Grey
 4 Yamaha Fazer FI V 2.0 [2016-2018]
 5 Yamaha SZ [2013-2014]
 6 Honda CB Twister
 7 Honda CB Hornet 160R
 8 Royal Enfield Bullet 350 [2007-2011]
 9 Hero Honda CBZ extreme
10 Bajaj Discover 125
```

```
        as.data.frame(bike_details)[1:10 ,1]
```

```
 [1] "Royal Enfield Classic 350"
 [2] "Honda Dio"
 [3] "Royal Enfield Classic Gunmetal Grey"
 [4] "Yamaha Fazer FI V 2.0 [2016-2018]"
 [5] "Yamaha SZ [2013-2014]"
 [6] "Honda CB Twister"
 [7] "Honda CB Hornet 160R"
 [8] "Royal Enfield Bullet 350 [2007-2011]"
 [9] "Hero Honda CBZ extreme"
[10] "Bajaj Discover 125"
```

If we use our usual `$` operator we do coerce to a vector though

```
        bike_details$name[1:10]
```

```
 [1] "Royal Enfield Classic 350"
 [2] "Honda Dio"
 [3] "Royal Enfield Classic Gunmetal Grey"
 [4] "Yamaha Fazer FI V 2.0 [2016-2018]"
 [5] "Yamaha SZ [2013-2014]"
 [6] "Honda CB Twister"
 [7] "Honda CB Hornet 160R"
 [8] "Royal Enfield Bullet 350 [2007-2011]"
 [9] "Hero Honda CBZ extreme"
[10] "Bajaj Discover 125"
```

The function commonly used from the tidyverse to grab a single column and return it as a vector is the `pull()` function from `dplyr`.

```
        bike_details[1:10, ] |>
          pull(name)
```

```
 [1] "Royal Enfield Classic 350"
 [2] "Honda Dio"
 [3] "Royal Enfield Classic Gunmetal Grey"
```

```
 [4] "Yamaha Fazer FI V 2.0 [2016-2018]"
 [5] "Yamaha SZ [2013-2014]"
 [6] "Honda CB Twister"
 [7] "Honda CB Hornet 160R"
 [8] "Royal Enfield Bullet 350 [2007-2011]"
 [9] "Hero Honda CBZ extreme"
[10] "Bajaj Discover 125"
```

Ok, back to the main task - reading the data in. We see in the fancy printing that R has each column stored in a particular format. How did R determine the column types?

From the help under `col_types` we see the following:

> One of NULL, a cols() specification, or a string. See vignette("readr") for more details.
> If NULL, all column types will be inferred from guess_max rows of the input, interspersed throughout the file. This is convenient (and fast), but not robust. If the guessed types are wrong, you'll need to increase guess_max or supply the correct types yourself.
> Column specifications created by list() or cols() must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only().
> Alternatively, you can use a compact string representation where each character represents one column:
> c = character
> i = integer
> n = number
> d = double
> l = logical
> f = factor
> D = date
> T = date time
> t = time
> ? = guess
> _ or - = skip
> By default, reading a file without a column specification will print a message showing what readr guessed they were. To remove this message, set show_col_types = FALSE or set 'options(readr.show_col_types = FALSE).

Ahh, so the `guess_max` argument tells our function to scan the first `x` number of rows and try to determine the column type. Note it says you may need to increase that argument to make sure data can be read in.

- **Checking column type is a basic data validation step!**
- You should check that each column was read in the way you would expect. If not, you may need to clean the data and convert the column to the appropriate data type.

## Reading in Any Delimited File

- Functions from *readr* and their purpose

| Delimiter | Function |
|---|---|
| comma ‘,’ | read_csv() |
| tab | read_tsv() |
| space ’ ’ | read_table() |
| semi-colon ‘;’ | read_csv2() (This uses `;` instead of commas, which is common in many countries) |
| other | read_delim(...,delim = ,...) |

Consider the `umps.txt` file available at: https://www4.stat.ncsu.edu/~online/datasets/umps2012.txt

- Download the file or open it in your browser.
- Note that the delimiter is a `>` sign!
- Note that there are no column names provided:
  - `Year Month Day Home Away HPUmpire` are the appropriate column names

We can use `read_delim()` to read in a generic delimited raw data file! Let’s check the help:

```
read_delim(
  file,
  delim = NULL,
  quote = "\"",
  escape_backslash = FALSE,
  escape_double = TRUE,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  comment = "",
  trim_ws = FALSE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  name_repair = "unique",
  num_threads = readr_threads(),
  progress = show_progress(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = should_read_lazy()
)
```

We see two arguments we need to worry about right off:

- `file` (path to file)
- `delim` the delimiter used in the raw data file
    - Single character used to separate fields within a record.
    - We want to specify a character string with the delimiter for this.

As we don't have column names we should also consider the `col_names` argument. This is set to `TRUE` by default. The help says:

> Either TRUE, FALSE or a character vector of column names.
> If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.
> If col_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.
> Missing (NA) column names will generate a warning, and be filled in with dummy names ...1, ...2 etc. Duplicate column names will generate a warning and be made unique, see name_repair to control how this is done.

- This means we want to set the value to `FALSE` or supply a character vector with the corresponding names!

```
        ump_data <- read_delim("https://www4.stat.ncsu.edu/~online/datasets
                               delim = ">",
                               col_names = c("Year", "Month", "Day", "Home
        )
```

```
Rows: 2359 Columns: 6
-- Column specification --------------------------------------------------------
Delimiter: ">"
chr (3): Home, Away, HPUmpire
dbl (3): Year, Month, Day

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
        ump_data
```

```
# A tibble: 2,359 x 6
    Year Month   Day Home  Away  HPUmpire
   <dbl> <dbl> <dbl> <chr> <chr> <chr>
 1  2012     4    12 MIN   LAA   D.J. Reyburn
 2  2012     4    12 SD    ARI   Marty Foster
 3  2012     4    12 WSH   CIN   Mike Everitt
 4  2012     4    12 PHI   MIA   Jeff Nelson
 5  2012     4    12 CHC   MIL   Fieldin Culbreth
```

```
 6  2012     4    12 LAD   PIT   Wally Bell
 7  2012     4    12 TEX   SEA   Doug Eddings
 8  2012     4    12 COL   SF    Ron Kulpa
 9  2012     4    12 DET   TB    Mark Carlson
10  2012     4    13 NYY   LAA   Mike DiMuro
# i 2,349 more rows
```

## Quick Aside: Date data

We see that the first three columns represent a `Year`, `Month`, and `Day`. These are currently stored as `dbl` (numeric data). Obviously, that's not great. We can't easily subtract two dates to get the difference in time or anything like that.

Insert the `lubridate` package. This is the `tidyverse` package for dealing with dates!

```
install.packages("lubridate") #only do this once!
```

```
library(lubridate) #do this each session
```

```
Attaching package: 'lubridate'

The following objects are masked from 'package:base':

    date, intersect, setdiff, union
```

If we look at `help(lubridate)` you can see under the section for parsing dates:

> Lubridate's parsing functions read strings into R as POSIXct date-time objects. Users should choose the function whose name models the order in which the year ('y'), month ('m') and day ('d') elements appear the string to be parsed: dmy(), myd(), ymd(), ydm(), dym(), mdy(), ymd_hms()). A very flexible and user friendly parser is provided by parse_date_time().

Ok, so we want to use `ymd()` or a variant and pass it a character string of the date to parse! No problem, we know how to do that :)

Under the help for the `ymd()` function, examples are given at the bottom of how to use the function. One example is

```
x <- c("09-01-01", "09-01-02", "09-01-03")
ymd(x)
```

```
[1] "2009-01-01" "2009-01-02" "2009-01-03"
```

Let's write a quick loop to loop through our observations, create this type of character string, and output a date variable!

We'll see a better way to do this once we get into `dplyr` but for now, let's initialize column to store date in and give it a date value.

```
        ump_data$date <- ymd("2012-01-01")
        ump_data
```

```
# A tibble: 2,359 x 7
    Year Month   Day Home  Away  HPUmpire         date
   <dbl> <dbl> <dbl> <chr> <chr> <chr>            <date>
 1  2012     4    12 MIN   LAA   D.J. Reyburn     2012-01-01
 2  2012     4    12 SD    ARI   Marty Foster     2012-01-01
 3  2012     4    12 WSH   CIN   Mike Everitt     2012-01-01
 4  2012     4    12 PHI   MIA   Jeff Nelson      2012-01-01
 5  2012     4    12 CHC   MIL   Fieldin Culbreth 2012-01-01
 6  2012     4    12 LAD   PIT   Wally Bell       2012-01-01
 7  2012     4    12 TEX   SEA   Doug Eddings     2012-01-01
 8  2012     4    12 COL   SF    Ron Kulpa        2012-01-01
 9  2012     4    12 DET   TB    Mark Carlson     2012-01-01
10  2012     4    13 NYY   LAA   Mike DiMuro      2012-01-01
# i 2,349 more rows
```

Now we'll loop through, paste together the three columns, and parse the date (storing it appropriately).

```
        for (i in 1:nrow(ump_data)){
          ump_data$date[i] <- ymd(paste(ump_data$Year[i],
                                        ump_data$Month[i],
                                        ump_data$Day[i],
                                        sep = "-"))
        }
        ump_data
```

```
# A tibble: 2,359 x 7
    Year Month   Day Home  Away  HPUmpire         date
   <dbl> <dbl> <dbl> <chr> <chr> <chr>            <date>
 1  2012     4    12 MIN   LAA   D.J. Reyburn     2012-04-12
 2  2012     4    12 SD    ARI   Marty Foster     2012-04-12
 3  2012     4    12 WSH   CIN   Mike Everitt     2012-04-12
 4  2012     4    12 PHI   MIA   Jeff Nelson      2012-04-12
 5  2012     4    12 CHC   MIL   Fieldin Culbreth 2012-04-12
 6  2012     4    12 LAD   PIT   Wally Bell       2012-04-12
 7  2012     4    12 TEX   SEA   Doug Eddings     2012-04-12
 8  2012     4    12 COL   SF    Ron Kulpa        2012-04-12
 9  2012     4    12 DET   TB    Mark Carlson     2012-04-12
10  2012     4    13 NYY   LAA   Mike DiMuro      2012-04-13
# i 2,349 more rows
```

Great! Now we can subtract dates and do other useful things with date data. We'll cover this kind of code shortly but we might want to know the days between being home plate umpire:

```
        ump_data |>
          filter(HPUmpire == "Marty Foster") |>
          mutate(days_off = date - lag(date))
```

```
# A tibble: 34 x 8
```

```
# A tibble: 34 x 8
    Year Month   Day Home  Away  HPUmpire     date       days_off
   <dbl> <dbl> <dbl> <chr> <chr> <chr>        <date>     <drtn>
 1  2012     4    12 SD    ARI   Marty Foster 2012-04-12 NA days
 2  2012     4    16 SF    PHI   Marty Foster 2012-04-16  4 days
 3  2012     4    21 KC    TOR   Marty Foster 2012-04-21  5 days
 4  2012     4    25 TB    LAA   Marty Foster 2012-04-25  4 days
 5  2012     4    29 BAL   OAK   Marty Foster 2012-04-29  4 days
 6  2012     5     4 CHC   LAD   Marty Foster 2012-05-04  5 days
 7  2012     5     8 HOU   MIA   Marty Foster 2012-05-08  4 days
 8  2012     5    13 CIN   WSH   Marty Foster 2012-05-13  5 days
 9  2012     5    17 DET   MIN   Marty Foster 2012-05-17  4 days
10  2012     5    21 MIL   SF    Marty Foster 2012-05-21  4 days
# i 24 more rows
```

This is easily done as we can take a date and subtract another date (via `lag(data)`, which grabs the date from the previous row).

## Reading in Tricky Raw Data Files

Sometimes our raw data will be in a `.txt` type file but not in a super nice format. In that case, we have a few functions that can help us out:

- `read_file()`
    - reads an entire file into a single string
- `read_lines()`
    - reads a file into a character vector with one element per line

Once the data is read into an R object, we can then usually parse it with **regular expressions**. Hopefully, that's not something you need to do very often!

## Quick R Video

Please pop this video out and watch it in the full panopto player!

17 - Reading Delimited Data

[Auto-generated transcript. Edits may have been applied for clarity.]

NC STATE

Powered by Panopto

## Recap!

The `tidyverse` has a package called `readr` that has many functions for reading in delimited data (raw data separated by a character string)

| Delimiter | Function |
|---|---|
| comma ‘,’ | read_csv() |
| tab | read_tsv() |
| space ‘ ’ | read_table() |
| semi-colon ‘;’ | read_csv2() (This uses `;` instead of commas, which is common in many countries) |
| other | read_delim(...,delim = ,...) |

`lubridate` is another package in the `tidyverse` that is really useful for dealing with date-type data!