Session 1b

The Data Carpentry (amended and delivered by Chris Oldnall)

October 25th 2023

Usage and Adaptation of Data Carpentry Materials

Objectives of the session:

- Load external data from a .csv file into a data frame.
- Summarise the contents of a data frame.
- Describe the difference between a factor and a string.
- Convert between strings and factors.
- Examine and change date formats.

Questions to be able to answer:

- What is a data.frame?
- How can I read a complete csv file into R?
- How can I get basic summary information about my dataset?
- How can I change the way R treats strings in my dataset?
- Why would I want strings to be treated differently?
- How are dates represented in R and how can I change the format?

What are data frames and tibbles?

Data frames are the *de facto* data structure for tabular data in R, and what we use for data processing, statistics, and plotting.

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Data frames are analogous to the more familiar spreadsheet in programs such as Excel, with one key difference. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors). For example, here is a figure depicting a data frame comprising a numeric, a character, and a logical vector.

Data Frame Reading

Data frames can be created by hand, but most commonly they are generated by the functions `read_csv()` or `read_table()`; in other words, when importing spreadsheets from your hard drive (or the web).

Presentation of the SAFI Data

SAFI (Studying African Farmer-Led Irrigation) is a study looking at farming and irrigation methods in Tanzania and Mozambique. The survey data was collected through interviews conducted between November 2016 and June 2017. For this lesson, we will be using a subset of the available data. For information about the full teaching dataset used in other lessons in this workshop, see the dataset description (https://www.datacarpentry.org/socialsci-workshop/data/).

Importing data

You are going to load the data in R's memory using the function `read_csv()` from the **`readr`** package, which is part of the **`tidyverse`**; learn more about the **`tidyverse`** collection of packages here. **`readr`** gets installed as part as the **`tidyverse`** installation. When you load the **`tidyverse`** (`library(tidyverse)`), the core packages (the packages used in most data analyses) get loaded, including **`readr`**.

An Import Example

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------
## v dplyr     1.1.3     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.3     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.
```

```
interviews <- read_csv("https://raw.githubusercontent.com/
```

```
## Rows: 131 Columns: 14
## -- Column specification --------------------------------
## Delimiter: ","
## chr  (7): village, respondent_wall_type, memb_assoc, af
```

Side-note on Conflicts

Before proceeding, however, this is a good opportunity to talk about conflicts. Certain packages we load can end up introducing function names that are already in use by pre-loaded R packages. For instance, when we load the tidyverse package below, we will introduce two conflicting functions: `filter()` and `lag()`. This happens because `filter` and `lag` are already functions used by the stats package (already pre-loaded in R). What will happen now is that if we, for example, call the `filter()` function, R will use the `dplyr::filter()` version and not the `stats::filter()` one. This happens because, if conflicted, by default R uses the function from the most recently loaded package. Conflicted functions may cause you some trouble in the future, so it is important that we are aware of them so that we can properly handle them, if we want.

Inspecting data frames

When calling a `tbl_df` object (like `interviews` here), there is already a lot of information about our data frame being displayed such as the number of rows, the number of columns, the names of the columns, and as we just saw the class of data stored in each column. However, there are functions to extract this information from data frames. Here is a non-exhaustive list of some of these functions. Let's try them out!

Inspecting functions

Size:

- `dim(interviews)` - returns a vector with the number of rows as the first element, and the number of columns as the second element (the **dim**ensions of the object)
- `nrow(interviews)` - returns the number of rows
- `ncol(interviews)` - returns the number of columns

Content:

- `head(interviews)` - shows the first 6 rows
- `tail(interviews)` - shows the last 6 rows

Inspecting functions 2

Names:

- `names(interviews)` - returns the column names (synonym of `colnames()` for `data.frame` objects)

Summary:

- `str(interviews)` - structure of the object and information about the class, length and content of each column
- `summary(interviews)` - summary statistics for each column
- `glimpse(interviews)` - returns the number of columns and rows of the tibble, the names and class of each column, and previews as many values will fit on the screen. Unlike the other inspecting functions listed above, `glimpse()` is not a "base R" function so you need to have the `dplyr` or `tibble` packages loaded to be able to execute it.

Note: most of these functions are "generic." They can be used on other types of objects besides data frames or tibbles.

Factors

R has a special data class, called factor, to deal with categorical data that you may encounter when creating plots or doing statistical analyses. Factors are very useful and actually contribute to making R particularly well suited to working with data. So we are going to spend a little time introducing them.

Factors represent categorical data. They are stored as integers associated with labels and they can be ordered (ordinal) or unordered (nominal). Factors create a structured relation between the different levels (values) of a categorical variable, such as days of the week or responses to a question in a survey. This can make it easier to see how one element relates to the other elements in a column. While factors look (and often behave) like character vectors, they are actually treated as integer vectors by `R`. So you need to be very careful when treating them as strings.

Factor Example

Once created, factors can only contain a pre-defined set of values, known as *levels*. By default, R always sorts levels in alphabetical order. For instance, if you have a factor with 2 levels:

```
respondent_floor_type <- factor(c("earth", "cement",
"cement", "earth"))
```

Factor Example Continued

R will assign `1` to the level `"cement"` and `2` to the level `"earth"` (because `c` comes before `e`, even though the first element in this vector is `"earth"`). You can see this by using the function `levels()` and you can find the number of levels using `nlevels()`:

```
levels(respondent_floor_type)
```

```
## [1] "cement" "earth"
```

```
nlevels(respondent_floor_type)
```

```
## [1] 2
```

Factors in a Data Set

In the case where our data has encoded a factor variable as a string, we can instead use the 'as.factor()' function to convert it. This is useful for further data wrangling and visualisation.

```
memb_assoc <- interviews$memb_assoc
memb_assoc
```

```
##   [1] "NULL" "yes"  "NULL" "NULL" "NULL" "NULL" "no"   "
##  [11] "NULL" "yes"  "no"   "NULL" "yes"  "NULL" "NULL" "
##  [21] "no"   "NULL" "NULL" "no"   "no"   "no"   "NULL" "
##  [31] "NULL" "yes"  "no"   "yes"  "yes"  "yes"  "NULL" "
##  [41] "NULL" "no"   "no"   "NULL" "no"   "no"   "yes"  "
##  [51] "NULL" "no"   "yes"  "no"   "NULL" "yes"  "no"   "
##  [61] "yes"  "NULL" "NULL" "NULL" "no"   "yes"  "no"   "
##  [71] "yes"  "NULL" "no"   "yes"  "NULL" "NULL" "yes"  "
##  [81] "no"   "no"   "yes"  "no"   "yes"  "no"   "no"   "
##  [91] "yes"  "yes"  "yes"  "no"   "no"   "no"   "no"   "
## [101] "yes"  "yes"  "no"   "NULL" "no"   "no"   "NULL" "
## [111] "no"   "NULL" "NULL" "no"   "no"   "no"   "no"   "
```

As a factor...

```
memb_assoc <- as.factor(memb_assoc)
memb_assoc
```

```
##   [1] NULL yes  NULL NULL NULL NULL no   yes  no   no
##  [16] NULL NULL NULL NULL NULL no   NULL NULL no   no
##  [31] NULL yes  no   yes  yes  yes  NULL yes  NULL yes
##  [46] no   yes  NULL NULL yes  NULL no   yes  no   NULL
##  [61] yes  NULL NULL NULL no   yes  no   no   no   no
##  [76] NULL yes  no   no   yes  no   no   yes  no   yes
##  [91] yes  yes  yes  no   no   no   no   yes  no   no
## [106] no   NULL no   no   NULL no   NULL NULL no   no
## [121] no   no   no   no   no   no   no   no   no   yes
## Levels: no NULL yes
```

Formatting Dates

One of the most common issues that new (and experienced!) R users have is converting date and time information into a variable that is appropriate and usable during analyses. A best practice for dealing with date data is to ensure that each component of your date is available as a separate variable. In our dataset, we have a column `interview_date` which contains information about the year, month, and day that the interview was conducted. Let's convert those dates into three separate columns.

Overview of Dates Data

Let's extract our `interview_date` column and inspect the structure:

```
dates <- interviews$interview_date
str(dates)
```

```
##  POSIXct[1:131], format: "2016-11-17" "2016-11-17" "2016
```

Splitting This Up

When we imported the data in R, `read_csv()` recognized that this column contained date information. We can now use the `day()`, `month()` and `year()` functions to extract this information from the date, and create new columns in our data frame to store it:

```
interviews$day <- day(dates)
interviews$month <- month(dates)
interviews$year <- year(dates)
```