Chapter 1

Importing the data

Welcome! This course will give you some additional practice with importing and cleaning data through a series of four case studies. We're assuming you've already completed these four courses:

- Intro to R
- Intermediate R
- Importing Data Into R
- Cleaning Data in R

If not, you'll want to check those out first.

You'll be importing and cleaning four real datasets that are a little messier than before. Don't worry -- you're up for the challenge!

Your first dataset describes online ticket sales for various events across the country. It's stored as a Comma-Separated Value (CSV) file called sales.csv. Let's jump right in!

Import sales.csv to the variable sales using the read.csv() function. Set the **stringsAsFactors** argument to FALSE so that character strings are preserved.

 # Import sales.csv: sales

```
sales <- read.csv("sales.csv", stringsAsFactors = FALSE)
```

_

Examining the data

As you know from the Cleaning Data in R course, the first step when preparing to clean data is to inspect it. Let's refresh your memory on some useful functions that can do that:

- dim() returns the dimensions of an object
- head() displays the first part of an object
- names() returns the names associated with an object

The sales data frame you created in the last exercise is pre-loaded in your workspace.

- View the dimensions of your new sales data frame.
- Inspect the first 6 rows of sales.
- View the column names of sales.

 # View dimensions of sales

```
dim(sales)
```

 # Inspect first 6 rows of sales

```
head(sales)
```

 # View column names of sales

```
names(sales)
```

_

Summarizing the data

Luckily, the rows and columns appear to be arranged in a meaningful way: each row represents an observation and each column a variable, or piece of information about that observation.

In R, there are a great many tools at your disposal to help get a feel for your data. Besides the three you used in the previous exercise, the functions str() and summary() can be very helpful.

The dplyr package, introduced in Cleaning Data in R, offers the glimpse() function, which can also be used for this purpose. The package is already installed on DataCamp; you just need to load it.

- Look at the structure of sales.
- View a summary of your data.
- Load the dplyr package.
- Use glimpse() to look at your data.

```
# Look at structure of sales
str(sales)
# View a summary of sales
summary(sales)
# Load dplyr
library(dplyr)
# Get a glimpse of sales
glimpse(sales)
```

_

Removing redundant info

You may have noticed that the first column of data is just a duplication of the row numbers. Not very useful. Go ahead and delete that column.

Remember that nrow() and ncol() return the number of rows and columns in a data frame, respectively.

Also, recall that you can use square brackets to subset a data frame as follows:

```
my_df[1:5, ] # First 5 rows of my_df
my_df[, 4]   # Fourth column of my_df
```

Alternatively, you can remove rows and columns using negative indices. For example:

```
my_df[-(1:5), ] # Omit first 5 rows of my_df
my_df[, -4]     # Omit fourth column of my_df
```

Take a subset of sales to omit the first column. Assign the result to sales2.

```
## sales is available in your workspace
# Remove the first column of sales: sales2
sales2 <- sales[, -(1)]
```

_

Information not worth keeping

Many of the columns have information that's of no use to us. For example, the first four columns contain internal codes representing particular events. The last fifteen columns also aren't worth keeping; there are too many missing values to make them worthwhile.

An easy way to get rid of unnecessary columns is to create a vector containing the column indices

- Create a vector called keep that contains the **indices** of the columns you want to save. Remember: you want to keep everything besides the first 4 and last 15 columns of sales2.
- Subset the columns of sales2 using your vector and assign the result to sales3.

sales2 is available in your workspace

 # Define a vector of column indices: keep

keep <- c(5 : (ncol(sales2) - 15))

 # Subset sales2 using keep: sales3

sales3 <- sales2[keep]

_

Separating columns

Some of the columns in your dataframe include multiple pieces of information that should be in separate columns. In this exercise, you will separate such a column into two: one for date and one for time. You will use the separate()function from the tidyr package (already installed for you).

Take a look at the event_date_time column by typing head(sales3$event_date_time) in the console. You'll notice that the date and time are separated by a space. Therefore, you'll use sep = " " as an argument to separate().

- Load the tidyr package.
- Split the event_date_time column of sales3 into "event_dt" and "event_time". Assign the result to sales4.
- Split the sales_ord_create_dttm column of sales4into "ord_create_dt" and "ord_create_time". Assign the result to sales5.

sales3 is pre-loaded in your workspace

 # Load tidyr

library(tidyr)

 # Split event_date_time: sales4

sales4 <- separate(sales3, event_date_time, c("event_dt", "event_time"), sep = " ")

 # Split sales_ord_create_dttm: sales5

sales5 <- separate(sales4, sales_ord_create_dttm, c("ord_create_dt", "ord_create_time"), sep = " ")

_

Dealing with warnings

Looks like that second call to separate() threw a warning. Not to worry; warnings aren't as bad as error messages. It's not saying that the command didn't execute; it's just a heads-up that something unusual happened.

The warning says Too few values at 4 locations. You may be able to guess already what the issue is, but it's still good to take a look.

The locations (i.e. rows) given in the warning are 2516, 3863, 4082, and 4183. Have a look at the contents of the sales_ord_create_dttmcolumn in those rows.

- Assign a vector issues that contains the indices of the four troublesome rows: 2516, 3863, 4082, and 4183.
- Subset sales3$sales_ord_create_dttmto look at these observations. Remember to use sales3 (not sales4), since you want the data frame from before you separated columns!
- For comparison, print element 2517 of sales3$sales_ord_create_dttm, which did not cause a warning.

 # Define an issues vector

issues <- c(2516, 3863, 4082, 4183)

 # Print values of sales_ord_create_dttm at these indices

sales3$sales_ord_create_dttm[issues]

 # Print a well-behaved value of sales_ord_create_dttm

sales3$sales_ord_create_dttm[2517]

_

Identifying dates

Some of the columns in your dataset contain dates of different events. Right now, they are stored as character strings. That's fine if all you want to do is look up the date associated with an event, but if you want to do any comparisons or math with the dates, it's MUCH easier to store them as Date objects.

Luckily, all of the date columns in this dataset have the substring "dt" in their name, so you can use the str_detect() function of the stringr package to find the date columns. Then you can coerce them to Date objects using a function from the lubridate package.

You'll use lapply() to apply the appropriate lubridate function to all of the columns that contain dates. Recall the following syntax for lapply() applied to some data frame columns of interest:

lapply(my_data_frame[, cols], function_name)

Also recall that function names in lubridate combine the letters y, m, d, h, m, and s depending on the format of the date/time string being read in.

- Load the stringr package.

- Use str_detect() to find values in the names() of sales5 containing the string "dt". Assign the resulting logical vector to the variable date_cols.
- Load the lubridate package.
- Coerce the date_cols into Date objects using lapply() and the appropriate function from lubridate. Conveniently, all date columns in sales5 are in year-month-day format, so you can use the ymd() function from lubridate.

sales5 is pre-loaded

 # Load stringr

library(stringr)

 # Find columns of sales5 containing "dt": date_cols

date_cols <- str_detect(names(sales5), "dt")

 # Load lubridate

library(lubridate)

 # Coerce date columns into Date objects

sales5[, date_cols] <- lapply(sales5[, date_cols], ymd)

_

More warnings!

As you saw, some of the calls to ymd() caused a failure to parse warning. That's probably because of more missing data, but again, it's good to check to be sure.

The first two lines of code (provided for you here) create a list of logical vectors called missing. Each vector in the list indicates the presence (or absence) of missing values in the corresponding column of sales5. See if the number of missing values in each column is the same as the number of rows that failed to parse in the previous exercise.

As a reminder, here are the warning messages:

Warning message: 2892 failed to parse.

Warning message: 101 failed to parse.

Warning message: 4 failed to parse.

Warning message: 424 failed to parse.

- Run the first line as-is to regenerate the date_cols vector.
- Run the second line as-is to generate a list of logical vectors representing missing values in the date columns of sales5.
- Use sapply() to create a numerical vector containing the number of NA values in each date column. Call this vector num_missing.
- Print out the num_missing vector.

stringr is loaded

 # Find date columns (don't change)

date_cols <- str_detect(names(sales5), "dt")

 # Create logical vectors indicating missing values (don't change)

missing <- lapply(sales5[, date_cols], is.na)

 # Create a numerical vector that counts missing values: num_missing

num_missing <- sapply(missing, sum)

 # Print num_missing

num_missing

_

Combining columns

Sure enough, the number of NAs in each column match the numbers from the warning messages, so missing data is the culprit. How to proceed depends on your desired analysis. If you really need complete sets of date/time information, you might delete the rows or columns containing NAs.

As your last step, you'll use the tidyr function unite() to combine the venue_city and venue_state columns into one column with the two values separated by a comma and a space. For example, "PORTLAND" "MAINE" should become "PORTLAND, MAINE".

- Combine the venue_city and venue_state columns of sales5 into a new column called venue_city_state, containing the city and state names separated by a comma and a space. Call the resulting data frame sales6.
- View the first 6 rows of sales6.

tidyr is loaded

 # Combine the venue_city and venue_state columns

sales6 <- unite(sales5, venue_city_state, venue_city, venue_state, sep=", ")

 # View the head of sales6

head(sales6)

_

Chapter 2

Using readxl

The Massachusetts Bay Transportation Authority ("MBTA" or just "the T" for short) manages America's oldest subway, as well as Greater Boston's commuter rail, ferry, and bus systems.

It's your first day on the job as the T's data analyst and you've been tasked with analyzing average ridership through time. You're in luck, because this chapter of the course will guide you through cleaning a set of MBTA ridership data!

The dataset is stored as an Excel spreadsheet called mbta.xlsx in your working directory. You'll use the read_excel() function from Hadley Wickham's readxl package to import it.

The first time you import a dataset, you might not know how many rows need to be skipped. In this case, the first row is a title (see **this Excel screenshot**), so you'll need to skip the first row. You can also download the whole Excel spreadsheet **here**.

- Load the readxl package.
- Import the ridership data using read_excel(). Set the skipargument to 1 and store the result to mbta.

 # Load readxl

```
library(readxl)
```

 # Import mbta.xlsx and skip first row: mbta

```
mbta <- read_excel ("mbta.xlsx", skip = 1)
```

_

Examining the data

Your new boss at the T has tasked you with analyzing the ridership data. Of course, you're going to clean the dataset first. The first step when cleaning a dataset is to explore it a bit.

The mbta data frame is already loaded in your workspace. Pay particular attention to how the rows and columns are organized and to the locations of missing values.

If you want to see what the dataset looks like in Excel, you can take another look at the **Excel screenshot**.

- View the structure of mbta.
- View the first 6 rows of mbta.
- View a summary of mbta.

mbta is pre-loaded

 # View the structure of mbta

str(mbta)

 # View the first 6 rows of mbta

head(mbta)

 # View a summary of mbta

summary(mbta)

_

Removing unnecessary rows and columns

It appears that the data are organized with observations stored as columns rather than as rows. You can fix that.

First, though, you can address the missing data. All of the NA values are stored in the All Modes by Qtr row. This row really belongs in a different data frame; it is a quarterly average of weekday MBTA ridership. Since this dataset tracks monthly average ridership, you'll remove that row.

Similarly, the 7th row (Pct Chg / Yr) and the 11th row (TOTAL) are not really observations as much as they are analysis. Go ahead and remove the 7th and 11th rows as well.

The first column also needs to be removed because it's just listing the row numbers.

In case you were wondering, this dataset is stored as a tibble which is just a specific type of data frame. For more information, see **here**.

- Remove the first, seventh, and eleventh rows of mbta (All Modes By Qtr, Pct Chg / Yr, and TOTAL). Name the resulting data frame mbta2.

 # Remove rows 1, 7, and 11 of mbta: mbta2

mbta2 <- mbta[-c(1, 7, 11),]

 # Remove the first column of mbta2: mbta3

mbta3 <- mbta2[,2:ncol(mbta2)]Remove the first column of mbta2. Name the resulting data frame mbta3.

_

Observations are stored in columns

Recall from a few exercises back that in your T ridership data, variables are stored in rows instead of columns. If you forget what that looked like, go ahead and enter head(mbta3) in the console and/or look at the **screenshot**.

The different modes of transportation (commuter rail, bus, subway, ferry, ...) are **variables**, providing information about each month's average ridership. The months themselves are **observations**. You can tell which is which because as you go through time, the month changes, but the modes of transport offered by the T do not.

As is customary, you want to represent variables in columns rather than rows. The first step is to use the gather() function from the tidyr package, which will gather columns into key-value pairs.

- Load the tidyr package.
- Gather the columns of mbta3. Use the - operator to omit the modecolumn. Call your new columns month and thou_riders (for "thousand riders") and assign the result to mbta4.
- View the head of mbta4.

```
## mbta3 is pre-loaded
# Load tidyr
library(tidyr)
# Gather columns of mbta3: mbta4
mbta4 <- gather(mbta3, month, thou_riders, -mode)
# View the head of mbta4
head(mbta4)
```

_

Type conversions

In a minute, you'll put variables where they belong (as column names). But first, take this opportunity to change the average weekday ridership column, thou_riders, into numeric values rather than character strings. That way, you'll be able to do things like compare values and do math.

Coerce the ridership column, mbta4$thou_riders, into a numeric. You can just assign the result back to the same column in mbta4.

```
## mbta4 is pre-loaded
# Coerce thou_riders to numeric
mbta4 <- as.numeric(mbta4$thou_riders)
```

_

Type conversions

In a minute, you'll put variables where they belong (as column names). But first, take this opportunity to change the average weekday ridership column, thou_riders, into numeric values rather than character strings. That way, you'll be able to do things like compare values and do math.

Coerce the ridership column, mbta4$thou_riders, into a numeric. You can just assign the result back to the same column in mbta4.

mbta4 is pre-loaded

 # Coerce thou_riders to numeric

```
mbta4$thou_riders <- as.numeric(mbta4$thou_riders)
```

_

Variables are stored in both rows and columns

Now, you can finish the job you started earlier: getting variables into columns. Right now, variables are stored as "keys" in the mode column. You'll use the tidyr function spread() to make them into columns containing average weekday ridership for the given month and mode of transport.

- Use spread() to change values in the mode column of mbta4 into column names. The columns should contain the average weekday ridership values associated with that mode of transport. Call the resulting data frame mbta5.
- View the head of mbta5.

tidyr is pre-loaded

 # Spread the contents of mbta4: mbta5

```
mbta5 <- spread(mbta4, mode, thou_riders)
```

 # View the head of mbta5

```
head(mbta5)
```

_

Separating columns

Your dataset is already looking much better! Your boss saw what a great job you're doing and now wants you to do an analysis of the T's ridership during certain months across all years.

Your dataset has month names in it, so that analysis will be a piece of cake. There's only one small problem: if you want to look at ridership on the T during every January (for example), the month and year are together in the same column, which makes it a little tricky.

In this exercise, you'll separate the month column into distinct month and year columns to make life easier.

- Look at head(mbta5) to remind yourself what the month column currently looks like.
- Split the month column of mbta5 at the dash: create a new month column with only the month and a year column with only the year. Assign the resulting data frame to mbta6.
- View the head of mbta6.

tidyr and mbta5 are pre-loaded

 # View the head of mbta5

```
head(mbta5)
```

 # Split month column into month and year: mbta6

```
mbta6 <- separate(mbta5, month, c("month", "year"), sep = "-")
```

 # View the head of mbta6

head(mbta6)

_

Separating columns

Your dataset is already looking much better! Your boss saw what a great job you're doing and now wants you to do an analysis of the T's ridership during certain months across all years.

Your dataset has month names in it, so that analysis will be a piece of cake. There's only one small problem: if you want to look at ridership on the T during every January (for example), the month and year are together in the same column, which makes it a little tricky.

In this exercise, you'll separate the month column into distinct month and year columns to make life easier.

- Look at head(mbta5) to remind yourself what the month column currently looks like.
- Split the month column of mbta5 at the dash: create a new month column with only the month and a year column with only the year. Assign the resulting data frame to mbta6.
- View the head of mbta6.

tidyr and mbta5 are pre-loaded

 # View the head of mbta5

head(mbta5)

 # Split month column into month and year: mbta6

mbta6 <- separate(mbta5, month, c("year", "month"), sep = "-")

 # View the head of mbta6

head(mbta6)

_

Do your values seem reasonable?

Before you write up the analysis for your boss, it's a good idea to screen the data for any obvious mistakes and/or outliers.

There are many valid techniques for doing this; you'll practice a couple of them here.

- View a summary() of mbta6. Pay particular attention to the Boat column stats.
- Generate a histogram of the Commuter Boat ridership by calling hist()on the column mbta6$Boat.

mbta6 is pre-loaded

 # View a summary of mbta6

summary(mbta6)

 # Generate a histogram of Boat column

hist(mbta6$Boat)

_

Dealing with entry error

Think for a minute about that Boat histogram. Every month, average weekday commuter boat ridership was on either side of four thousand. Then, one month it jumped to 40 thousand without warning? Unless the Olympics were happening in Boston that month (they weren't), this value is certainly an error. You can assume that whoever was entering the data that month accidentally typed 40 instead of 4. Because it's an error, you don't want this value influencing your analysis. In this exercise, you'll locate the incorrect value and change it to 4.

After you make the change, you'll run the last two commands in the editor as-is. They use functions you may not know yet to produce some cool ridership plots: one showing the lesser-used modes of transport (take a look at the gorgeous seasonal variation in Boat ridership), and one showing all modes of transport. The plots are based on the long version of the data we produced in Exercise 4 -- a good example of using different data formats for different purposes.

If you'd like to learn how to do this on your own, check out DataCamp's *Data Visualization with ggplot2* courses!

- Create a numeric variable i to store the index of the incorrect Boat value in mbta6. Combine a call to which() with a comparison operator (i.e. >) to determine the row number.
- Overwrite the incorrect value of mbta6$Boat with a 4.
- Verify that the change was made by looking at another histogram of mbta6$Boat.
- Run the last two commands as-is to generate pretty plots.

```
# Find the row number of the incorrect value: i
i <- which(mbta6$Boat == 40, arr.ind=TRUE)
# Replace the incorrect value with 4
mbta6[i, mbta6$Boat == 40] <- 4
# Generate a histogram of Boat column
hist(mbta6$Boat)
# Look at Boat and Trackless Trolley ridership over time (don't change)
ggplot (mbta_boat, aes(x = month, y = thou_riders, col = mode)) +  geom_point() +
  scale_x_discrete(name = "Month", breaks = c(200701, 200801, 200901, 201001, 201101)) +
  scale_y_continuous(name = "Avg Weekday Ridership (thousands)")
# Look at all T ridership over time (don't change)
ggplot(mbta_all, aes(x = month, y = thou_riders, col = mode)) + geom_point() +
  scale_x_discrete(name = "Month", breaks = c(200701, 200801, 200901, 201001, 201101)) +
  scale_y_continuous(name = "Avg Weekday Ridership (thousands)")
```

_

Chapter 3

Importing the data

As a person of many talents, it's time to take on a different job: nutrition analysis! Your goal is to analyze the sugar content of a sample of foods from around the world.

A large dataset called food.csv is ready for your use in the working directory. Instead of the usual read.csv(), however, you're going to use the faster fread() from the data.table package. By default, the data will come in as a data table, but since you're used to working with data frames, you can get fread() to return one by setting data.table = FALSE.

[Note: In order to make these exercises manageable, we've taken a random subset of the original data. The dataset you'll be working with may not be large enough for fread() to make a huge difference, but be aware that there will be times when read.csv() just won't cut it.]

- Load the data.table package.
- Import food.csv using fread(), setting data.table to FALSE. Call the resulting data frame food.

 # Load data.table

require(data.table) / library(data.table)

 # Import food.csv as a data frame: df_food

food <- fread("food.csv", data.table=FALSE)

Examining the data

As usual, you'll need to get an idea of what the dataset looks like in order to know how to proceed.

- View a summary of food.
- View the head of food.
- View the structure of food.

food is pre-loaded

 # View summary of food

summary(food)

 # View head of food

head(food)

 # View structure of food

str(food)

Inspecting variables

The str(), head(), and summary() functions are designed to give you some information about a dataset without being overwhelming. However, this dataset is so large and has so many variables that even these outputs seemed pretty intimidating!

The glimpse() function from the dplyr package often formats information in a more approachable way.

Yet another option is to just look at the column names to see what kinds of data you have. As you look at the names, pay particular attention to any pairs that look like duplicates.

- Load the dplyr package.
- Get a glimpse of food.
- View the column names of food.

 # Load dplyr

library(dplyr)

 # View a glimpse of food

glimpse(food)

 # View column names of food

```
names(food)
```

_

Removing duplicate info

Wow! That's a lot of variables. To summarize, there's some information on what and when information was added (1:9), meta information about food (10:17, 22:27), where it came from (18:21, 28:34), what it's made of (35:52), nutrition grades (53:54), some unclear (55:63), and some nutritional information (64:159).

There are also many different pairs of columns that contain duplicate information. Luckily, you have a trusty assistant who went through and identified duplicate columns for you.

A vector has been created for you that lists out all of the duplicates; all you need to do is remove those columns from the dataset. Don't forget, you can use the - operator to specify columns to omit, e.g.:

```
my_df[, -3] # Omit third column
```

- Run the first line of code as-is to define the duplicates vector.
- Use duplicates to remove those columns from food. Call the resulting data frame food2.

 # Define vector of duplicate cols (don't change)

```
duplicates <- c(4, 6, 11, 13, 15, 17, 18, 20, 22,
                24, 25, 28, 32, 34, 36, 38, 40,
                44, 46, 48, 51, 54, 65, 158)
```

 # Remove duplicates from food: food2

```
food2 <- food[, (-duplicates)]
```

_

Removing useless info

Your dataset is much more manageable already.

In addition to duplicate columns, there are many columns containing information that you just can't use. For example, the first few columns contain internal codes that don't have any meaning to us. There are also some column names that aren't clear enough to tell what they contain.

All of these columns can be deleted. Once again, your assistant did a splendid job finding the indices for you.

- Run the first line of code as-is to define the useless vector.
- Remove the useless columns from food2. Call the resulting data frame food3.

food2 is pre-loaded

 # Define useless vector (don't change)

```
useless <- c(1, 2, 3, 32:41)
```

 # Remove useless columns from food2: food3

```
food3 <- food2[, (-useless)]
```

_

Finding columns

Looking much nicer! Recall from the first exercise that you are assuming you will be analyzing the sugar content of these foods. Therefore, your next step is to look at a summary of the nutrition information. All of the columns with nutrition info contain the character string "100g"as part of their name, which makes it easy to identify them.

- Create a vector called nutrition containing the column indices of the nutrition data. To do this, use a stringr function on the column names of food3 to determine which columns contain "100g" in their name.
- View a summary of the nutrition columns.

```
## stringr and food3 are pre-loaded
# Create vector of column indices: nutrition
nutrition <- str_detect(names(food3), "100g")
# View a summary of nutrition columns
summary(food3[, nutrition])
```

_

Replacing missing values

Unfortunately, the summary revealed that the nutrition data are mostly NAvalues. After consulting with the lab technician, it appears that much of the data is missing because the food just doesn't have those nutrients.

But all is not lost! The lab tech also said that for sugar content, zero values are sometimes entered explicitly, but sometimes the values are just left empty to denote a zero. A statistical miracle!

In this exercise, you'll replace all NA values with zeroes in the sugars_100g column and make histograms to visualize the result. Then, you will exclude the observations which have no sugar to see how the distribution changes.

- Use is.na() to create a vector indicating which elements in the sugars_100g column of food3 are currently NA. Call the vector missing.
- Subset food3$sugars_100g using missing to replace the missing values with zeros. You can just overwrite the values directly with the <-operator.
- Create a histogram of the sugars_100g variable in food3 with the breaks argument equal to 100.
- Create food4, a subset of food3 for which food3$sugars_100g is larger than 0, excluding the foods with zero sugar. You can use single bracket subsetting to do this.
- Create a histogram of the sugars_100g variable in food4 with the breaks argument equal to 100.

 # Find indices of sugar NA values: missing

```
missing <- is.na(food3$sugars_100g)
```

 # Replace NA values with 0

```
food3$sugars_100g[missing] <- 0
```

 # Create first histogram

```
hist(food3$sugars_100g, breaks = 100)
```

 # Create food4

```
food4 <- food3[food3$sugars_100g > 0, ]
```

 # Create second histogram

```
hist(food4$sugars_100g, breaks = 100)
```

_

Dealing with messy data

Your analysis of sugar content was so impressive that you've now been tasked with determining how many of these foods come in some sort of plastic packaging. (No good deed goes unpunished, as they say.)

Your dataset has information about packaging, but there's a bit of a problem: it's stored in several different languages (Spanish, French, and English). This takes messy data to a whole new level! There is no R package to selectively translate, but what if you could just work with the messy data directly?

You're in luck! The root word for *plastic* is same in English (plastic), French (plastique), and Spanish (plastico). To get a general idea of how many of these foods are packaged in plastic, you can look through the packaging column for the string "plasti".

- Create a vector to locate entries in the packaging column of food3 containing the string "plasti". Call the resulting logical vector plastic.
- Calculate the sum of plastic to count how many of the foods are packaged in plastic.

stringr is loaded

 # Find entries containing "plasti": plastic

```
plastic <- str_detect(food3$packaging, "plasti")
```

 # Print the sum of plastic

```
sum(plastic)
```

_

Chapter 4

Importing the data

In this chapter, you'll work with attendance data from public schools in the US, organized by school level and state, during the 2007-2008 academic year. The data contain information on average daily attendance (ADA) as a percentage of total enrollment, school day length, and school year length.

The data were given to you in an Excel spreadsheet, which you can **download** or view a **screenshot** of. Either way, take a moment to look at the spreadsheet.

Do you see any symptoms of untidy data? At first glance, it looks like the first row is a description of the data, the second row is a variable itself that groups multiple columns together, and the fourth row gives numbers for the columns, which might look nice in a spreadsheet but isn't very useful for you, the analyst. You'll take it one step at a time to import the data using the gdata package. The name of this spreadsheet is "attendance.xls" and is available in your working directory.

- Load the gdata package.
- Read in the .xls file and store it as an object called att.

 # Load the gdata package

```
library(gdata)
```

 # Import the spreadsheet: att

```
att <- read.xls("attendance.xls")
```

_

Examining the data

For your reference, **here** is an image of the spreadsheet you are responsible for. Now that you've successfully imported the data into R, you can get a better idea of how the spreadsheet was read in.

- Print the column names of att.
- Print the first 6 rows of att.
- Print the last 6 rows of att.
- Print the structure of att to get some information about your data.

 # Print the column names

```
names(att)
```

 # Print the first 6 rows

```
head(att)
```

 # Print the last 6 rows

```
tail(att)
```

 # Print the structure

```
str(att)
```

_

Removing unnecessary rows

Again, for reference, **here** is an image of the first 22 rows of the original spreadsheet you were given. Looking at this image, you can see that rows 1, 4, 11, and 17 of the spreadsheet are useless. But is it safe to do something like the following?

att2 <- att[-c(1, 4, 11, 17),]

No! From the last exercise, you might have realized that the read.xls()function actually imported the first row of the original data frame as the variable name for the first column. Did you notice that the first 6 rows of att aren't the same as the first six rows you saw in the original spreadsheet? What about the 11th and 17th rows?

When you're importing a messy spreadsheet into R, it's good practice to compare the original spreadsheet with what you've imported. It turns out that, by default, the read.xls() function skips empty rows such as the 11th and 17th.

After viewing your dataframe, you realize you still need to get rid of the third row of att, as well as rows 56 through 59.

- Create a vector of row numbers to be removed. Call it remove.
- Create att2 by removing these rows from att.

 # Create remove

remove <- c(3, 56:59)

 # Create att2

att2 <- att[(-remove),]

_

Removing useless columns

Once more, for reference, **here** is an image of the first 22 rows of the original spreadsheet. You can see here that the *columns* 3, 5, 7, 9, 11, 13, 15, and 17 (or columns C, E, G, I, K, M, O, Q in Excel) don't contain the values of average daily attendance (ADA). You'll get rid of them in this exercise.

- Create a vector remove containing the indices of the unwanted columns.
- Create att3, a subset of att2 without the unwanted columns.

 # Create remove

remove <- c(3, 5, 7, 9, 11, 13, 15, 17)

 # Create att3

att3 <- att2[, (-remove)]

_

Splitting the data

In many cases, a single data frame stores multiple "tables" of information. You can often diagnose this problem by looking at the column names and noticing duplicate rows.

In this data frame, columns 1, 6, and 7 represent attendance data for US elementary schools, columns 1, 8, and 9 represent data for secondary schools, and columns 1 through 5 represent data for all schools in the US.
Each of these should be stored as its own separate data frame, so you'll split them up here.

- Subset att3 to include only data for elementary schools (columns 1, 6, and 7). Name the resulting data frame att_elem.
- Subset att3 to include only data for secondary schools (columns 1, 8, and 9). Name the resulting data frame att_sec.
- Subset att3 to include data for all schools (columns 1 through 5). Name the resulting data frame att4.

att3 is pre-loaded

 # Subset just elementary schools: att_elem

```
att_elem <- att3[,c(1,6,7)]
```

 # Subset just secondary schools: att_sec

```
att_sec <- att3[,c(1,8,9)]
```

 # Subset all schools: att4

```
att4 <- att3[, 1:5]
```

_

Replacing the names

Since you went through so much trouble finding out which row stored the variable names, you should store that row as the actual column names of the data frame. We've modified the names a bit in order to be more stylistically sound; they're stored as cnames in the editor.

- Run the first line of code as-is to define the cnames vector.
- Assign the cnames vector to the column names of att4.
- Remove the first two rows of att4 and name the result att5.
- View the column names of att5.

att4 is pre-loaded

 # Define cnames vector (don't change)

```
cnames <- c("state", "avg_attend_pct", "avg_hr_per_day",
            "avg_day_per_yr", "avg_hr_per_yr")
```

 # Assign column names of att4

```
colnames(att4) <- cnames
```

 # Remove first two rows of att4: att5

```
att5 <- att4[(-c(1:2)),]
```

 # View the names of att5

```
names(att5)
```

_

Cleaning up extra characters

One of the most irritating things about this dataset is that the state names are all stored as the same number of characters, with periods padding the ends of the shorter states. That may be helpful for reading the spreadsheet, but it makes your life harder, so you'll deal with it in this exercise.

One pitfall to avoid: . is a special character in the language of *regular expressions* (a.k.a. regex). In order to specify that you actually want to remove periods and not their regex equivalent (which is "all characters"), use \\.. This is called an "escape" sequence.

- Use the function **str_replace_all()** to replace all periods in the state column of att5 with "". Remember to use \\. to represent a period. Assign the result back to att5$state.
- Remove white space around the state names, assigning the result back to att5$state once more. There's another stringr function called **str_trim()** for this purpose.
- View the head of att5.

```
## stringr and att5 are pre-loaded
# Remove all periods in state column
att5$state <- str_replace_all(att5$state, "\\.", "")
# Remove white space around state names
att5$state <- str_trim(att5$state)
# View the head of att5
head(att5)
```

_

Some final type conversions

Finally, you'll convert the values in certain variables to numerics (instead of factors). It's worth noting that in previous chapters, your numerical data has often come in as character strings. This is just a difference between read.xls() and the other import functions you've used.

The dplyr package offers an efficient method for applying a function to many columns at once. If you'd like to learn more, check out DataCamp's course on *Data Manipulation in R with dplyr*! Since you might not have taken the dplyr course yet, we're just showing you the code here. You'll do the same thing, but using subsetting and sapply() instead.

- Review the dplyr code, just to see how it works.
- Define a vector called cols that contains indices of columns containing numerical data. Take a look at att5 if you need to!

- Using a call to sapply(), single bracket subsetting, and the vector cols, change the type of these columns to numeric with as.numeric(). Assign the result of the function call back to the subset of att5.

```
# Change columns to numeric using dplyr (don't change)
library(dplyr)
example <- mutate_at(att5, vars(-state), funs(as.numeric))
# Define vector containing numerical columns: cols
cols <- -1
# Use sapply to coerce cols to numeric
att5[, cols] <- sapply(att5[, cols], as.numeric)
```