Lunch Hour Learning Guide, Sessions 3-5, Spring 2025

Cleaning Data in R - Parts 1, 2 & 3

Sean Morey Smith

2025-02-12

Abstract

One of the most challenging tasks in the research process is cleaning the data. This is especially true when you are working with data that you did not collect/create!

To become a proficient R user, it is helpful to learn a process for evaluating your data and determining what cleaning is needed. These sessions describe that process, with tips and tricks for making cleaning more effective and efficient.

Before Starting

Create a new, self-contained R project in your chosen sub-directory, where you will store your work from this session. For guidance, see the instructions from Session 1.

Create a sub-directory called "data" in your project directory. Save the titles.csv and top_100_billboard.csv files in your "data" directory. They are available in a zip at
https://library.rice.edu/sites/default/files/materials/data.zip

Session 3: Cleaning Data in R - Part 1

What You Will Learn

- How to get an overview of loaded data
- How to rename variables in a data frame
- How to keep and drop variables (columns) and rows from a data frame

Evaluating Your Data

This session focuses on messy data within an otherwise "tidy" structure. A tidy dataset has one row per observation, where an observation is defined as a single member of a sample (e.g., one person, country, organism, etc.). In addition, tidy data involve one variable per column (i.e., one measure) and one value per cell. For example, last name and first name would be stored in separate columns, rather than together in a single "name" column.

The next session will address how to handle data that are not already in this tidy structure. For now, we will focus on tasks like cleaning column names to make them more useful, removing unnecessary rows and columns, and determining what to do with missing data.

In this session, we will use two datasets: one built-in (`iris`) and one that we will import (`titles`). Note that you can view a list of built-in datasets with the function `data()`.

```r
data()
```

Let's evaluate the `iris` dataset first. You may want to begin by looking at any documentation that accompanies a dataset, such as a data dictionary, code book, or README file. In this case, you can use `help()` because `iris` is a built-in dataset.

```r
help(iris)
```

```
## starting httpd help server ... done
```

Step 1: Examine Data Documentation and Structure

According to the help file, the `iris` dataset gives the measurements in cm of four variables: sepal length and width and petal length and width. The sample is composed of 50 flowers from each of three species of iris. The data are already stored in a rectangular data frame and meet the criteria for a tidy structure. Nonetheless, let's practice saving the data as a data frame and looking at its structure:

```r
iris <- data.frame(iris)
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

There are four numeric variables and one factor. Although the column names are easy to read, they are a little long and include periods rather than underscores (the latter of which is a better coding practice for naming variables). You will probably want to address this, so let's put it on our list of cleaning tasks.

Step 2: Perform a Preliminary Check for Missing Values

You will also want to check for any missing values. R has a built-in value `NA` to indicate missing data.

```r
anyNA(iris)
```

```
## [1] FALSE
```

The `FALSE` result indicates that there are no `NA` values. However, `anyNA()`this function`does not pick up on missing values that are encoded as something other than`NA'. We will look at other ways to check for missing data later.

Step 3: Perform a Preliminary Check for Extreme Values

You can do this by looking at the summary of data:

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

The summary gives us a sense of whether there might be any extreme values due to measurement error, data entry error, or outliers.

You may need to draw upon your (or others') domain expertise to determine whether any of the summary statistics seems unusual. For now, let's assume that the range seems acceptable and that there are no obvious outliers or errors. However, we will verify this assumption later.

Step 4: Create a Cleaning Script and Summarize What Cleaning Is Needed

The current dataset is already pretty clean. The only improvement you might make is to rename the variables. However, with a messy dataset, it is recommended that you create a cleaning script file that summarizes the cleaning needed.

Renaming Variables

Now you will proceed with cleaning! As noted earlier, the column names (variables) are a little long and include periods. You want to rename them, so use the `rename()` function from the `dplyr` package within `tidyverse`.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.0     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
iris_rn <- iris %>%
    rename(slength = Sepal.Length, swidth = Sepal.Width,
           plength = Petal.Length, pwidth = Petal.Width)
str(iris_rn)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ slength: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ swidth : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ plength: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ pwidth : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species: Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Note that the new name is listed first, followed by an equal sign and the old name.

The `rename()` function is useful for renaming selected variables. However, sometimes you want to rename all the variables at once; in this case, `rename()` is less efficient because it requires typing the old variable names. Instead, you can use the `names()` function in base R.

```
names(iris_rn) <- c("s_length", "s_width",
                    "p_length", "p_width", "species")
str(iris_rn)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ s_length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ s_width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ p_length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ p_width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Note that this code assigns the `names()` of `iris_rn` to the vector of character values specified *in order*.

Now the column names are concise, use exclusively lowercase, and use underscores rather than periods.

To save the clean data frame to a new file, use `write.csv()`.

```
write.csv(x = iris_rn, "data/iris_rn_data.csv", row.names = FALSE)
```

Importing a New Dataset

For the next few exercises, you will use the titles.csv dataset, which contains Netflix movie and TV show data compiled by Soero (2022) on the website Kaggle. Save the file in your "/data" folder within the current R project.

Read the dataset into the R session:

```
titles <- read.csv("data/titles.csv", stringsAsFactors = FALSE)
```

Keeping and Dropping Columns and Rows

In many situations, you will want to work with a smaller subset of your data. To avoid scrolling and trying to remember column names, you can drop columns from view with `dplyr` functions.

First, examine the `titles` data frame to see what it contains.

```
str(titles)
```

```
## 'data.frame':    5850 obs. of  9 variables:
##  $ title       : chr  "Five Came Back: The Reference Films" "Taxi Driver" "Deliverance" "Monty Pytho
##  $ type        : chr  "SHOW" "MOVIE" "MOVIE" "MOVIE" ...
##  $ release_year: int  1945 1976 1972 1975 1967 1969 1979 1971 1967 1980 ...
##  $ runtime     : int  51 114 109 91 150 30 94 102 110 104 ...
##  $ genre       : chr  "documentation" "drama" "drama" "fantasy" ...
##  $ country     : chr  "US" "US" "US" "GB" ...
##  $ rating      : chr  "TV-MA" "R" "R" "PG" ...
##  $ imdb_score  : num  NA 8.2 7.7 8.2 7.7 8.8 8 7.7 7.7 5.8 ...
##  $ tmdb_score  : num  NA 8.18 7.3 7.81 7.6 ...
```

You want to keep the following variables: `title`, `type`, `release_year`, `genre`, `rating`, and `imdb_score`. You also want to reorder some of these variables.

```
titles_2 <- titles %>%
    select(title, type, rating, genre, release_year, imdb_score)
str(titles_2)
```

```
## 'data.frame':    5850 obs. of  6 variables:
##  $ title       : chr  "Five Came Back: The Reference Films" "Taxi Driver" "Deliverance" "Monty Pytho
##  $ type        : chr  "SHOW" "MOVIE" "MOVIE" "MOVIE" ...
##  $ rating      : chr  "TV-MA" "R" "R" "PG" ...
##  $ genre       : chr  "documentation" "drama" "drama" "fantasy" ...
##  $ release_year: int  1945 1976 1972 1975 1967 1969 1979 1971 1967 1980 ...
##  $ imdb_score  : num  NA 8.2 7.7 8.2 7.7 8.8 8 7.7 7.7 5.8 ...
```

Note that you must assign the results to a new variable (`titles_2`); otherwise, the changes will only be visible in the current output. The new data frame object `titles_2` includes only those variables you selected, in the order you specified.

Now that you have a smaller number of variables, you might also want to keep or drop certain rows. You can use `filter()` to accomplish this task.

For example, let's filter to include only movies from 2000 and later. Because all rows will be movies, we can also drop the type column by calling `select(-)` on the name of the column. The `-` tells R to select all columns except the named column.

```
movies_2000 <- titles_2 %>%
    filter(type == "MOVIE" & release_year >= 2000) %>%
    select(-type)
```

This code results in a smaller subset of movies. Note that this is the equivalent of indexing; the function `filter()` is a bit more flexible than bracket notation and requires fewer characters.

Another common task is to omit rows that do not meet some criterion. For example, from this data frame of movies from 2000 on, you might want to omit those with particularly infrequent ratings. As an interim step, you will need to determine what the ratings are and how frequently they occur in this data frame. Use `count()` from the `dplyr` package.

```
movies_2000 %>%
    count(rating)
```

```
##   rating    n
## 1      G  118
## 2  NC-17   15
## 3     PG  221
## 4  PG-13  428
## 5      R  512
## 6   none 2280
```

It appears that NC-17 movies are pretty uncommon on Netflix, so you decide to omit those from the data frame. Recall that `!=` is the logical operator for "is not the same as."

```
movies_2000 <- movies_2000 %>%
    filter(rating != "NC-17")
```

Check to make sure your code worked by re-running the previous `count()` function.

```
movies_2000 %>%
    count(rating)
```

```
##   rating    n
## 1      G  118
## 2     PG  221
## 3  PG-13  428
## 4      R  512
## 5   none 2280
```

Now you have a data frame with five rating values, but most have a rating of "none"! Look at its structure.

```
str(movies_2000)
```

```
## 'data.frame':    3559 obs. of  5 variables:
##  $ title       : chr  "Snatch" "Mission: Impossible II" "The Replacements" "Phir Bhi Dil Hai Hindusta
##  $ rating      : chr  "R" "PG-13" "PG-13" "PG-13" ...
##  $ genre       : chr  "crime" "thriller" "comedy" "comedy" ...
##  $ release_year: int  2000 2000 2000 2000 2000 2000 2007 2004 2002 2003 ...
##  $ imdb_score  : num  8.3 6.1 6.6 6.1 6 7.3 7.1 7.1 8.1 7 ...
```

Removing the TV shows and "NC-17" movies has left 3,559 rows.

Session 4: Cleaning Data in R - Part 2

What You Will Learn

- How to deal with missing data
- How to clean factor levels
- How to convert (change) data types
- How to recode values

Load Packages and Data

In part 2, we will continue using the `tidyverse` package and the titles.csv dataset, containing Netflix movie and TV show data.

Load the package and read the dataset into the R session. Then reduce the data to movies from 2000 and later:

```
library(tidyverse)
titles <- read.csv("data/titles.csv", stringsAsFactors = FALSE)
movies_2000 <- titles %>%
    filter(type == "MOVIE" & release_year >= 2000) %>%
    select(-type)
```

Dealing with Missing Data

Data can be missing for many reasons: data entry error, valid or invalid skipped responses to a survey, a variable being not applicable to the observation, and other reasons. To the extent possible, you should determine the reason for missing data before deciding how to handle these cases.

Step 1: Check for Missing Data

Recall that the first step is to check for any missing data in the data frame. You can do a blanket sweep with `anyNA()`.

```
anyNA(movies_2000)
```

```
## [1] TRUE
```

In this data frame, there are `NA` values, as indicated by the `TRUE` result.

Although a `FALSE` result means that there are no `NA` values in the dataset, that does not necessarily mean that there are no missing data! Recall that researchers may encode missing data with different codes, such as 99 or 0 or even a word such as "none". Thus, be sure to check the data dictionary for such nuances.

Let's discuss how to convert those alternate missing value codes to `NA`s.

Step 2: Replace Other Missing Value Codes with `NA` (if appropriate)

In the `movies_2000` data frame, you saw that "none" was a common response to the rating variable. You want to change "none" to `NA`.

Make a copy of `movies_2000` so we don't lose data. Then, use the function `replace()` to indicate that you want to replace "none" in `rating` with `NA`.

```
movies_nas <- movies_2000
movies_nas$rating <-
    replace(movies_2000$rating, movies_2000$rating == "none", NA)
```

Finally, check the results:

```
movies_nas %>%
    count(rating)
```

```
##   rating    n
## 1      G  118
## 2  NC-17   15
## 3     PG  221
## 4  PG-13  428
## 5      R  512
## 6   <NA> 2280
```

Your re-coding worked! This option may not be appropriate in all situations. For example, in some datasets, words such as "none" may indicate 0 rather than a missing value. As usual, proceed with caution!

Step 3: Handle Missing Data

Step 3, Option A: Remove All Rows with Missing Data Depending on the reason for missing data, you may want to omit all rows with `NA` values in one or more columns. However, you should also use caution with this option, as it can substantially reduce your sample size (and your statistical power).

Furthermore, removing all missing data can add bias to the data, as observations with missing values may differ in important ways from other observations.

Nonetheless, it is useful to know how to remove all rows with missing data; the function `na.omit()` serves this purpose.

```
movies_nona <-
    na.omit(movies_nas)
anyNA(movies_nona)
```

```
## [1] FALSE
```

```
summary(movies_nona)
```

```
##     title            release_year     runtime         genre          
##  Length:1189        Min.   :2000   Min.   :  8.0   Length:1189       
##  Class :character   1st Qu.:2013   1st Qu.: 92.0   Class :character  
##  Mode  :character   Median :2017   Median :103.0   Mode  :character  
##                     Mean   :2016   Mean   :106.1                     
##                     3rd Qu.:2020   3rd Qu.:120.0                     
##                     Max.   :2022   Max.   :224.0                     
##    country             rating            imdb_score      tmdb_score    
##  Length:1189        Length:1189        Min.   :2.000   Min.   : 2.000  
##  Class :character   Class :character   1st Qu.:5.600   1st Qu.: 6.000  
##  Mode  :character   Mode  :character   Median :6.400   Median : 6.600  
##                                        Mean   :6.285   Mean   : 6.511  
##                                        3rd Qu.:7.100   3rd Qu.: 7.100  
##                                        Max.   :8.900   Max.   :10.000  
```

You have successfully removed all `NA` values from the data frame, but notice that the number of movies has dropped substantially; there was a lot of missing data in the dataset!

Step 3, Option B: Remove NAs from Specific Columns Because a blanket approach can be inappropriate for some datasets, you may want to remove rows that have NAs only in specific columns. For example, if you are doing a correlation analysis, you will need complete data for both variables, but it may matter less if other information was missing for some observations. In this case, you can specify which columns require complete data. Here, let's just check `release_year` and `imdb_score` are complete.

```
movies_complete <-
    movies_2000[complete.cases(
        movies_2000[ , c("release_year", "imdb_score")]
    ), ]
anyNA(movies_complete)
```

```
## [1] TRUE
```

```
summary(movies_complete)
```

```
##     title            release_year     runtime          genre
##  Length:3265        Min.   :2000   Min.   :  2.0   Length:3265
##  Class :character   1st Qu.:2016   1st Qu.: 88.0   Class :character
##  Mode  :character   Median :2018   Median : 99.0   Mode  :character
##                     Mean   :2017   Mean   :100.2
##                     3rd Qu.:2020   3rd Qu.:115.0
##                     Max.   :2022   Max.   :225.0
##
##    country             rating            imdb_score     tmdb_score
##  Length:3265        Length:3265        Min.   :1.50   Min.   : 1.000
##  Class :character   Class :character   1st Qu.:5.50   1st Qu.: 5.900
##  Mode  :character   Mode  :character   Median :6.30   Median : 6.500
##                                        Mean   :6.22   Mean   : 6.448
##                                        3rd Qu.:7.00   3rd Qu.: 7.100
##                                        Max.   :9.10   Max.   :10.000
##                                                       NA's   :140
```

This code will index all rows for which the specified columns are "complete" (i.e., no NAs) and return all data for those rows.

Note that running `anyNA()` indicates that there are still some NA values in your dataset, but (as shown by `summary()`) there are no NAs in the two variables of interest. Furthermore, you have lost much less data by only removing those rows with NA in the two variables of interest.

Step 3, Option C: Treat Missing Data Variable by Variable Another alternative is to treat missing data in a more tailored fashion, according to the nature of the variable and the possible reasons for missing values. You can use the function `is.na()` and its negation `!is.na()` for this purpose.

For example, we saw that many movies do not have a rating. If you omit all of those rows, you will lose a lot of data. It is preferable to keep the rows and simply report those movies as a separate category, such as "Unrated."

We will use the `movies_nas` data frame because that is the one in which we converted "none" values to NAs. Let's look at how to re-code the NA values for the rating variable. We will also add a column to the data frame to capture this new code.

First, copy the data from `movies_nas$rating` into the new variable `movies_nas$rating_new`.

```
movies_nas$rating_new <- movies_nas$rating
```

Next, recode NAs to the character value "Unrated".

```
movies_nas$rating_new[is.na(movies_nas$rating_new)] = "Unrated"
```

Then check to see the results.

```
movies_nas %>%
    count(rating_new)
```

```
##   rating_new    n
## 1          G  118
## 2      NC-17   15
## 3         PG  221
## 4      PG-13  428
## 5          R  512
## 6    Unrated 2280
```

The NA values are now coded as "Unrated." This illustrates how you can replace a value with NA and NA with a value with different functions.

For another variable, you may want to remove rows with missing data in just that variable.

For example, there are a handful of movies that do not include a genre. If you want to exclude them from the data frame, use `!is.na()`.

```
movie_genres <- movies_nas[!is.na(movies_nas$genre), ]
```

Note the use of bracket notation in both `is.na` code chunks. In the first code chunk, you are indexing all rows for which the value of `rating_new` is NA, then reassigning those values to "Unrated".

In the second code chunk, you are indexing all rows that do not have an NA value for genre and returning all columns for those rows.

Step 3, Option D: Impute Values Another option is imputing other values for missing values. However, this technique should also be used with caution and only with a clear understanding of the nature of the data and the limitations of interpreting imputed values.

As an example, you saw that some movies were missing an IMDB score. You could impute the overall mean IMDB score value for those missing values.

```
movies_imputed <- movies_2000
movies_imputed$imdb_score[is.na(movies_imputed$imdb_score)] =
    mean(movies_imputed$imdb_score, na.rm = TRUE)
head(movies_imputed, n = 30)
```

```
##                                    title release_year runtime    genre country
## 1                                 Snatch         2000     103    crime      US
## 2                  Mission: Impossible II         2000     123 thriller      US
## 3                       The Replacements         2000     118   comedy      US
## 4            Phir Bhi Dil Hai Hindustani         2000     160   comedy      IN
```

```
## 5                                   Fiza        2000     170  romance      IN
## 6               Before the Flying Circus        2000      55   comedy      GB
## 7                               The Mist        2007     126   horror      US
## 8                             Mean Girls        2004      97   comedy      CA
## 9                    Catch Me If You Can        2002     141    drama      US
## 10                            Old School        2003      88   comedy      US
## 11 Anchorman: The Legend of Ron Burgundy        2004      95   comedy      US
## 12                             Inception        2010     148   action      US
## 13                          The Departed        2006     151    drama      US
## 14                             Insidious        2010     103   horror      CA
## 15                    War of the Worlds         2005     117   action      US
## 16                                Wanted        2008     110   action      DE
## 17                         Casino Royale        2006     144   action      DE
## 18           Blade Runner: The Final Cut        2007     117   action      US
## 19                          The Terminal        2004     128    drama      US
## 20                       Michael Clayton        2007     114 thriller      US
## 21                     Road to Perdition        2002     117 thriller      US
## 22                              Big Fish        2003     125   action      US
## 23                       The Hurt Locker        2008     131 thriller      US
## 24                                  Troy        2004     163   action      MT
## 25                      Sherlock Holmes         2009     129    crime      AU
## 26                        Love Actually         2003     135    drama      GB
## 27                             Get Smart        2008     110   comedy      US
## 28                     Quantum of Solace        2008     106 thriller      GB
## 29                        The Blind Side        2009     129    drama      US
## 30                   Apocalypse Now Redux       2001     196    drama      US
##    rating imdb_score tmdb_score
## 1       R   8.300000      7.800
## 2   PG-13   6.100000      6.100
## 3   PG-13   6.600000      6.600
## 4   PG-13   6.100000      6.600
## 5    none   6.000000      6.300
## 6    none   7.300000      8.500
## 7       R   7.100000      6.900
## 8   PG-13   7.100000      7.200
## 9   PG-13   8.100000      8.000
## 10      R   7.000000      6.577
## 11  PG-13   7.100000      6.700
## 12  PG-13   8.800000      8.400
## 13      R   8.500000      8.200
## 14  PG-13   6.800000      6.929
## 15  PG-13   6.500000      6.474
## 16      R   6.700000      6.500
## 17   none   8.000000      7.518
## 18      R   6.219816      9.000
## 19  PG-13   7.400000      7.300
## 20      R   7.200000      6.700
## 21      R   7.700000      7.400
## 22  PG-13   8.000000      7.769
## 23      R   7.500000      7.300
## 24      R   7.300000      7.100
## 25  PG-13   7.600000      7.200
## 26      R   7.600000      7.100
## 27  PG-13   6.500000      6.200
```

```
## 28  PG-13   6.600000       6.304
## 29  PG-13   7.600000       7.661
## 30      R   6.219816       9.800
```

Note that all IMDB values that were previously `NA` have been replaced with the overall IMDB score mean. For this to work, you must include the argument `na.rm = TRUE` within the `mean()` function (which drops `NA` values from the calculation of the mean).

Once again, this is a blanket approach to a single variable. It is also possible to do more precise imputation. For example, you could impute the mean based on rating or genre or some other category. We will look at this situation in the programming session, as it is a good application for loop functions.

Working with Factors

The final topic in this lesson is working with factors. Factors are categorical variables that do not have inherent numerical value (nominal data, such as ethnicity, sex, or disciplinary field) or that have imprecise numerical value (ordinal data, such as small/medium/large or first/second/third).

Although the values look like a character data type, R treats them as levels of a categorical variable "behind the scenes." Factors are useful for certain analyses, so it is helpful to know how to work with them and to clean them up when needed!

In many datasets, some variables should be treated as factors and others not. Currently, the default when importing data is to *not* treat character (string) data as factors, but this varies across verions of R so it's good to always specify whether you want characters (strings) loaded as factors or not. Let's look at how to handle various use cases.

Importing String Data as Factors

When you imported the titles file, you set the argument `stringsAsFactors` to `FALSE`. This is usually a safe bet for most imports, as it prevents you from accidentally converting a string variable with unique character values to a multi-level factor.

Let's look at what would happen if you set `stringsAsFactors` to `TRUE` with the current dataset.

```
titles_fct <- read.csv("data/titles.csv", stringsAsFactors = TRUE)
str(titles_fct)
```

```
## 'data.frame':    5850 obs. of  9 variables:
##  $ title       : Factor w/ 5798 levels "'76","#ABtalks",..: 1667 4410 1290 3098 4626 3096 2689 1349
##  $ type        : Factor w/ 2 levels "MOVIE","SHOW": 2 1 1 1 1 2 1 1 1 1 ...
##  $ release_year: int  1945 1976 1972 1975 1967 1969 1979 1971 1967 1980 ...
##  $ runtime     : int  51 114 109 91 150 30 94 102 110 104 ...
##  $ genre       : Factor w/ 18 levels "action","animation",..: 5 6 6 8 17 3 3 16 4 13 ...
##  $ country     : Factor w/ 96 levels "AE","AF","AR",..: 90 90 90 28 28 28 28 90 90 90 ...
##  $ rating      : Factor w/ 12 levels "G","NC-17","none",..: 9 6 6 4 3 7 6 6 6 6 ...
##  $ imdb_score  : num  NA 8.2 7.7 8.2 7.7 8.8 8 7.7 7.7 5.8 ...
##  $ tmdb_score  : num  NA 8.18 7.3 7.81 7.6 ...
```

One obvious problem is that the `title` observation is now a factor, resulting in 5,798 levels of that factor! It is usually better to import the data without factors and then convert selected variables to factors. Naturally, there's a function for that.

Converting a String Variable to a Factor

For example, you can convert `movies_2000$rating` to a factor. The previous `count()` indicated that there are six possible values of this variable: "G", "NC-17", PG", "PG-13", "R", and "none." You will make these the levels of the factor.

```
movies_2000$rating <- factor(movies_2000$rating,
                             levels = c("G", "PG", "PG-13", "R", "NC-17", "none"),
                             ordered = TRUE)
str(movies_2000)
```

```
## 'data.frame':    3574 obs. of  8 variables:
##  $ title       : chr  "Snatch" "Mission: Impossible II" "The Replacements" "Phir Bhi Dil Hai Hindusta
##  $ release_year: int  2000 2000 2000 2000 2000 2000 2007 2004 2002 2003 ...
##  $ runtime     : int  103 123 118 160 170 55 126 97 141 88 ...
##  $ genre       : chr  "crime" "thriller" "comedy" "comedy" ...
##  $ country     : chr  "US" "US" "US" "IN" ...
##  $ rating      : Ord.factor w/ 6 levels "G"<"PG"<"PG-13"<..: 4 3 3 3 6 6 4 3 3 4 ...
##  $ imdb_score  : num  8.3 6.1 6.6 6.1 6 7.3 7.1 7.1 8.1 7 ...
##  $ tmdb_score  : num  7.8 6.1 6.6 6.6 6.3 ...
```

Note four things about this code:

1. The function `factor()` is applied to the desired column (`movies_2000$rating`).
2. The argument `levels` is set to a vector of character values that specify the levels (i.e., categories) of this factor. Make sure that these levels correspond to the actual character values in the data.
3. The argument ordered is set to `TRUE`, meaning that the five levels are in the order that we want them to appear. Creating an ordered factor has implications for data visualization, as un-ordered factors are generally shown in alphabetical order.
4. The entire code is assigned to the column `movies_2000$rating`, which will overwrite the existing structure of that variable.

It is also possible to check the levels in a factor and to rename them with the function `levels()`. First, simply view the existing levels.

```
levels(movies_2000$rating)
```

```
## [1] "G"     "PG"    "PG-13" "R"     "NC-17" "none"
```

Now let's capitalize the first letter of the level "none" and then check the levels again:

```
levels(movies_2000$rating) <- c("G", "PG", "PG-13", "R", "NC-17", "None")
levels(movies_2000$rating)
```

```
## [1] "G"     "PG"    "PG-13" "R"     "NC-17" "None"
```

Note that this script can also be used to re-code levels, but use caution! Make sure you confirm the correct coding with the documentation in your data dictionary. Also, it is strongly recommended to address missing data before dealing with factors and not to include `NA` as a factor level!

Changing Data Types

When you are working with a dataset that you have not created yourself, there is a good chance that you will need to make some adjustments to the default data types that R has assigned to variables when reading in the data. We saw this previously when we changed character data to a factor, but you can change factors (or any numeric data) to character data as well with the function `as.character()`.

First, remember that we loaded titles.csv with `stringsAsFactors = TRUE` into `titles_fct`. Check the structure to be sure.

```
str(titles_fct)
```

```
## 'data.frame':    5850 obs. of  9 variables:
##  $ title       : Factor w/ 5798 levels "'76","#ABtalks",..: 1667 4410 1290 3098 4626 3096 2689 1349
##  $ type        : Factor w/ 2 levels "MOVIE","SHOW": 2 1 1 1 1 2 1 1 1 1 ...
##  $ release_year: int  1945 1976 1972 1975 1967 1969 1979 1971 1967 1980 ...
##  $ runtime     : int  51 114 109 91 150 30 94 102 110 104 ...
##  $ genre       : Factor w/ 18 levels "action","animation",..: 5 6 6 8 17 3 3 16 4 13 ...
##  $ country     : Factor w/ 96 levels "AE","AF","AR",..: 90 90 90 28 28 28 28 90 90 90 ...
##  $ rating      : Factor w/ 12 levels "G","NC-17","none",..: 9 6 6 4 3 7 6 6 6 6 ...
##  $ imdb_score  : num  NA 8.2 7.7 8.2 7.7 8.8 8 7.7 7.7 5.8 ...
##  $ tmdb_score  : num  NA 8.18 7.3 7.81 7.6 ...
```

You decide that you want titles to be character data rather than factors, so change the data type.

```
titles_fct$title <- as.character(titles_fct$title)
str(titles_fct)
```

```
## 'data.frame':    5850 obs. of  9 variables:
##  $ title       : chr  "Five Came Back: The Reference Films" "Taxi Driver" "Deliverance" "Monty Pytho
##  $ type        : Factor w/ 2 levels "MOVIE","SHOW": 2 1 1 1 1 2 1 1 1 1 ...
##  $ release_year: int  1945 1976 1972 1975 1967 1969 1979 1971 1967 1980 ...
##  $ runtime     : int  51 114 109 91 150 30 94 102 110 104 ...
##  $ genre       : Factor w/ 18 levels "action","animation",..: 5 6 6 8 17 3 3 16 4 13 ...
##  $ country     : Factor w/ 96 levels "AE","AF","AR",..: 90 90 90 28 28 28 28 90 90 90 ...
##  $ rating      : Factor w/ 12 levels "G","NC-17","none",..: 9 6 6 4 3 7 6 6 6 6 ...
##  $ imdb_score  : num  NA 8.2 7.7 8.2 7.7 8.8 8 7.7 7.7 5.8 ...
##  $ tmdb_score  : num  NA 8.18 7.3 7.81 7.6 ...
```

Similarly, you can convert values to numeric values (depending on how they are originally coded). For example, if you wanted to convert decimals to whole numbers, you could use `as.integer()`:

```
titles_fct$tmdb_score <- as.integer(titles_fct$tmdb_score)
str(titles_fct)
```

```
## 'data.frame':    5850 obs. of  9 variables:
##  $ title       : chr  "Five Came Back: The Reference Films" "Taxi Driver" "Deliverance" "Monty Pytho
##  $ type        : Factor w/ 2 levels "MOVIE","SHOW": 2 1 1 1 1 2 1 1 1 1 ...
##  $ release_year: int  1945 1976 1972 1975 1967 1969 1979 1971 1967 1980 ...
##  $ runtime     : int  51 114 109 91 150 30 94 102 110 104 ...
##  $ genre       : Factor w/ 18 levels "action","animation",..: 5 6 6 8 17 3 3 16 4 13 ...
##  $ country     : Factor w/ 96 levels "AE","AF","AR",..: 90 90 90 28 28 28 28 90 90 90 ...
```

```
##  $ rating      : Factor w/ 12 levels "G","NC-17","none",..: 9 6 6 4 3 7 6 6 6 6 ...
##  $ imdb_score  : num  NA 8.2 7.7 8.2 7.7 8.8 8 7.7 7.7 5.8 ...
##  $ tmdb_score  : int  NA 8 7 7 7 8 7 7 7 6 ...
```

Note, however, that this will not round the number; it will simply drop the decimal.

Other relevant functions for coercion are `as.double()` and `as.factor()`.

Recoding Variables

A common issue is the need to recode the values of a variable. For example, you might want to make a variable dichotomous (i.e., binary: two values, such as Yes/No or 1/0).

To recode a variable with binary values, use `ifelse()`. For example, you want to code movies made in the United States as 1 and all others as 0. Instead of recoding the original variable `titles_fct$country`, you create a new variable with the recoded values: `titles_fct$country2`.

```r
titles_fct$country2 <- ifelse(titles_fct$country == "US", 1, 0)
str(titles_fct)
```

```
## 'data.frame':    5850 obs. of  10 variables:
##  $ title       : chr  "Five Came Back: The Reference Films" "Taxi Driver" "Deliverance" "Monty Pytho
##  $ type        : Factor w/ 2 levels "MOVIE","SHOW": 2 1 1 1 1 2 1 1 1 1 ...
##  $ release_year: int  1945 1976 1972 1975 1967 1969 1979 1971 1967 1980 ...
##  $ runtime     : int  51 114 109 91 150 30 94 102 110 104 ...
##  $ genre       : Factor w/ 18 levels "action","animation",..: 5 6 6 8 17 3 3 16 4 13 ...
##  $ country     : Factor w/ 96 levels "AE","AF","AR",..: 90 90 90 28 28 28 28 90 90 90 ...
##  $ rating      : Factor w/ 12 levels "G","NC-17","none",..: 9 6 6 4 3 7 6 6 6 6 ...
##  $ imdb_score  : num  NA 8.2 7.7 8.2 7.7 8.8 8 7.7 7.7 5.8 ...
##  $ tmdb_score  : int  NA 8 7 7 7 8 7 7 7 6 ...
##  $ country2    : num  1 1 1 0 0 0 0 1 1 1 ...
```

If you want to do more complex recoding, you can specify each condition and use the `dplyr` functions `mutate()` and `recode()`.

For example, you want to recode `rating` into three groups: child-friendly (CF), parental guidance (PG), and not suitable for children (NS). This recoded variable will be called `movies$age_group`.

```r
movies <- titles_fct %>%
    filter(type == "MOVIE") %>%
    mutate(age_group = recode(rating,
                              "G" = "CF",
                              "PG" = "PG",
                              "PG-13" = "PG",
                              "none" = "none",
                              .default = "NS"))
str(movies)
```

```
## 'data.frame':    3744 obs. of  11 variables:
##  $ title       : chr  "Taxi Driver" "Deliverance" "Monty Python and the Holy Grail" "The Dirty Dozen
##  $ type        : Factor w/ 2 levels "MOVIE","SHOW": 1 1 1 1 1 1 1 1 1 1 ...
##  $ release_year: int  1976 1972 1975 1967 1979 1971 1967 1980 1961 1966 ...
##  $ runtime     : int  114 109 91 150 94 102 110 104 158 117 ...
```

```
##  $ genre       : Factor w/ 18 levels "action","animation",..: 6 6 8 17 3 16 4 13 1 18 ...
##  $ country     : Factor w/ 96 levels "AE","AF","AR",..: 90 90 28 28 28 90 90 90 28 90 ...
##  $ rating      : Factor w/ 12 levels "G","NC-17","none",..: 6 6 4 3 6 6 6 6 3 5 ...
##  $ imdb_score  : num  8.2 7.7 8.2 7.7 8 7.7 7.7 5.8 7.5 7.3 ...
##  $ tmdb_score  : int  8 7 7 7 7 7 7 6 7 7 ...
##  $ country2    : num  1 1 0 0 0 1 1 1 0 1 ...
##  $ age_group   : Factor w/ 4 levels "CF","NS","none",..: 2 2 4 3 2 2 2 2 3 4 ...
```

Note that you have to specify "none" as a code; otherwise, it will be recoded as "NS". However, if you had used the data frame in which we had previously coded "none" as `NA`, those cells would remain `NA` without your specifying it in the code.

Session 5: Cleaning Data in R - Part 3

What You Will Learn

- How to fix character data (e.g., case and white space inconsistency)
- How to transform data formats (long-to-wide and wide-to-long)

Load Packages

Make sure the `tidyverse` package is loaded.

```
library(tidyverse)
```

Fixing Character Data

Fixing Leading/Trailing White Space

Two common issues surround character data. First, white space may appear before or after a character value in the original file. When this occurs, R needs to be told to omit that white space; otherwise, the white space will be encoded as a character, which can lead to issues. Let's look at how to address this when importing a new dataset.

```
messy_songs <- read.csv("data/top_100_billboard.csv", stringsAsFactors = FALSE)
```

This is the usual way to read in data, and everything seems normal. However, when you start to run some analyses, problems appear.

```
messy_songs %>%
    head(100) %>%
    count(artist)
```

```
##                              artist n
## 1                     Elvis Presley 1
## 2                     Andy Williams 1
## 3                      Anita Bryant 2
## 4                 Annette Funicello 1
## 5                    Barrett Strong 1
```

```
## 6                         Billy Bland 1
## 7                           Bob Luman 1
## 8                         Bobby Darin 1
## 9                        Bobby Rydell 4
## 10                          Bobby Vee 1
## 11                         Brenda Lee 4
## 12                       Brian Hyland 1
## 13                       Brook Benton 1
## 14                       Charlie Rich 1
## 15                     Chubby Checker 1
## 16                     Connie Francis 3
## 17                   Connie Francis   1
## 18                     Connie Stevens 1
## 19                      Conway Twitty 1
## 20  Dinah Washington & Brook Benton 2
## 21             Dion and the Belmonts 1
## 22                      Donnie Brooks 1
## 23                         Duane Eddy 1
## 24                      Elvis Presley 1
## 25                        Fats Domino 1
## 26                 Ferrante & Teicher 1
## 27                     Frankie Avalon 1
## 28                      Freddy Cannon 1
## 29                       Guy Mitchell 1
## 30 Hank Ballard and The Midnighters 1
## 31                       Hank Locklin 1
## 32                         Jack Scott 2
## 33                      Jackie Wilson 2
## 34                       Jeanne Black 1
## 35                         Jim Reeves 1
## 36                      Jimmy Charles 1
## 37                      Jimmy Clanton 1
## 38                        Jimmy Jones 1
## 39                      Jimmy Jones   1
## 40                          Joe Jones 1
## 41                    Johnny Burnette 1
## 42                      Johnny Horton 1
## 43                     Johnny Preston 2
## 44                   Johnny Tillotson 1
## 45         Johnny and the Hurricanes 1
## 46                         Larry Hall 1
## 47                        Larry Verne 1
## 48                        Lloyd Price 1
## 49                       Mark Dinning 1
## 50                      Marty Robbins 1
## 51                       Marv Johnson 2
## 52 Maurice Williams and the Zodiacs 1
## 53                        Neil Sedaka 1
## 54                          Paul Anka 3
## 55                         Paul Evans 1
## 56                      Percy Faith   1
## 57                        Ray Charles 1
## 58                       Ray Peterson 1
## 59                       Ricky Nelson 1
```

```
## 60                        Ron Holden 1
## 61                       Roy Orbison 1
## 62                         Sam Cooke 2
## 63                     Skip and Flip 1
## 64                      Spencer Ross 1
## 65                    Steve Lawrence 2
## 66              The Bill Black Combo 1
## 67                 The Brothers Four 1
## 68                        The Browns 1
## 69                        The Crests 1
## 70                      The Drifters 1
## 71               The Everly Brothers 4
## 72                    The Fendermen 1
## 73                    The Four Preps 1
## 74             The Hollywood Argyles 1
## 75                 The Little Dippers 1
## 76                      The Platters 1
## 77                       The Safaris 1
## 78                      The Ventures 1
## 79                       Toni Fisher 1
```

Right off the bat, you might see an issue: Elvis Presley appears at the top of the list. He also appears again in the "E"s. Does that mean he is the greatest singer of all time? R can't tell you that, but from what it's done you can tell that there was an issue with white space here, as each artist should appear only once in the list, and the list should be alphabetical.

Scrolling through the output, we see other artists names repeated, again suggesting white space issues. You need to remove the white space, and it's best to do it at the time of import by adding the argument `strip.white`.

```
clean_songs <- read.csv("data/top_100_billboard.csv",
                        stringsAsFactors = FALSE,
                        strip.white = TRUE)
```

Now when we run `count` again, we see Elvis in just one row, with the correct frequency count.

```
clean_songs %>%
    head(100) %>%
    count(artist)
```

```
##                              artist n
## 1                     Andy Williams 1
## 2                      Anita Bryant 2
## 3                 Annette Funicello 1
## 4                    Barrett Strong 1
## 5                       Billy Bland 1
## 6                         Bob Luman 1
## 7                       Bobby Darin 1
## 8                      Bobby Rydell 4
## 9                         Bobby Vee 1
## 10                       Brenda Lee 4
## 11                     Brian Hyland 1
## 12                     Brook Benton 1
```

```
## 13                        Charlie Rich 1
## 14                      Chubby Checker 1
## 15                      Connie Francis 4
## 16                      Connie Stevens 1
## 17                       Conway Twitty 1
## 18   Dinah Washington & Brook Benton 2
## 19              Dion and the Belmonts 1
## 20                       Donnie Brooks 1
## 21                          Duane Eddy 1
## 22                       Elvis Presley 2
## 23                         Fats Domino 1
## 24                  Ferrante & Teicher 1
## 25                      Frankie Avalon 1
## 26                       Freddy Cannon 1
## 27                        Guy Mitchell 1
## 28 Hank Ballard and The Midnighters 1
## 29                        Hank Locklin 1
## 30                          Jack Scott 2
## 31                       Jackie Wilson 2
## 32                        Jeanne Black 1
## 33                          Jim Reeves 1
## 34                       Jimmy Charles 1
## 35                       Jimmy Clanton 1
## 36                         Jimmy Jones 2
## 37                           Joe Jones 1
## 38                     Johnny Burnette 1
## 39                       Johnny Horton 1
## 40                      Johnny Preston 2
## 41                    Johnny Tillotson 1
## 42         Johnny and the Hurricanes 1
## 43                          Larry Hall 1
## 44                         Larry Verne 1
## 45                         Lloyd Price 1
## 46                        Mark Dinning 1
## 47                       Marty Robbins 1
## 48                        Marv Johnson 2
## 49 Maurice Williams and the Zodiacs 1
## 50                         Neil Sedaka 1
## 51                           Paul Anka 3
## 52                          Paul Evans 1
## 53                         Percy Faith 1
## 54                         Ray Charles 1
## 55                        Ray Peterson 1
## 56                        Ricky Nelson 1
## 57                          Ron Holden 1
## 58                         Roy Orbison 1
## 59                           Sam Cooke 2
## 60                       Skip and Flip 1
## 61                        Spencer Ross 1
## 62                      Steve Lawrence 2
## 63                 The Bill Black Combo 1
## 64                   The Brothers Four 1
## 65                          The Browns 1
## 66                          The Crests 1
```

```
## 67                    The Drifters 1
## 68              The Everly Brothers 4
## 69                   The Fendermen 1
## 70                  The Four Preps 1
## 71            The Hollywood Argyles 1
## 72               The Little Dippers 1
## 73                     The Platters 1
## 74                      The Safaris 1
## 75                     The Ventures 1
## 76                      Toni Fisher 1
```

As a side note: We have just counted artists in the first 100 rows of the dataset (using `head()`) for convenience.

Fixing Capitalization

Some datasets use all CAPS for their variable names and values; others use all lowercase. This is a matter of preference, but you need to be consistent in what you use. (I recommend lowercase, as it saves keystrokes.)

You can convert character data to all capitals or all lowercase with functions in the `stringr` package (part of the `tidyverse`): `str_to_upper()` and `str_to_lower()`.

For example, let's say you want to convert all the variable names to uppercase. Wrap the call `names()` in `str_to_upper()`.

```
str_to_upper(names(clean_songs))
```

```
## [1] "NO"     "TITLE"  "ARTIST" "YEAR"
```

Alternatively, convert them to lower case:

```
str_to_lower(names(clean_songs))
```

```
## [1] "no"     "title"  "artist" "year"
```

You can do the same for data values, too.

```
clean_songs$title <- str_to_lower(clean_songs$title)
head(clean_songs$title)
```

```
## [1] "theme from a summer place" "he'll have to go"
## [3] "cathy's clown"             "running bear"
## [5] "teen angel"                "i'm sorry"
```

Two other good converting functions to know in the `stringr` package are `str_to_title()` and `str_to_sentence()`, which convert to "Title Case" and "Sentence case", respectively. Let's convert the song titles to Title Case.

```
clean_songs$title <- str_to_title(clean_songs$title)
head(clean_songs$title)
```

```
## [1] "Theme From A Summer Place" "He'll Have To Go"
## [3] "Cathy's Clown"             "Running Bear"
## [5] "Teen Angel"                "I'm Sorry"
```

Changing the Structure of a Data Frame

In most cases, you want your data frame to be "tidy" (one observation per row, one variable per column, and one value per cell). In contrast, a "wide" data frame contains multiple observations in the same row. For example, if you ran a pretest-posttest study and included the pre and post scores as two separate columns, this dataset would be in a wide format.

Converting it to a "long" format (which is more consistent with a tidy structure) would mean creating a column called "time" and including two rows per participant - one with a `time` of "pre", and one with `time` as "post". This is somewhat counter-intuitive to those of us who work with repeated measures data, but in most cases, R does best when each observation per individual (for the same variable) has its own row.

Now, in truth, most "tidy" data frames are intermediate in nature. Truly long data frames literally have a single observation per row - with observation ID, a column to capture what the variable is, and a column for the value. However, truly long data frames are rather impractical, and R doesn't need the data to be in that format to process the data. Instead, a happy medium will suffice: One observation (i.e., participant at a certain time point) per row, with multiple variables represented as columns.

All this being said, let's look at how to convert one format to the other and back again!

In the `clean_songs` data frame, there are 100 rows per year, with one song and ranking per row. This is in line with a tidy (mostly long) format.

Theoretically, a song could be in the top 100 two years in a row. If you wanted to explore whether any songs have rankings in two or more years, you could convert the data frame to a wide format.

Let's just look at the first two years of data (the first 200 rows).

```
early_songs <- clean_songs[1:200,]
```

This long data frame contains the top 100 songs for 1960 and 1961 (one song per rank per year).

We will transform the data frame so that the two years appear as separate columns in a "wide" data frame.

```
songs_wide <- early_songs %>%
    pivot_wider(names_from = year, values_from = no)
head(songs_wide)
```

```
## # A tibble: 6 x 4
##   title                    artist              `1960` `1961`
##   <chr>                    <chr>               <chr>  <chr>
## 1 Theme From A Summer Place Percy Faith        1      <NA>
## 2 He'll Have To Go         Jim Reeves          2      <NA>
## 3 Cathy's Clown            The Everly Brothers 3      <NA>
## 4 Running Bear             Johnny Preston      4      <NA>
## 5 Teen Angel               Mark Dinning        5      <NA>
## 6 I'm Sorry                Brenda Lee          6      <NA>
```

Note that in the `songs_wide` data frame, the column `no` (the rank number of each song) disappeared, but the values from that column were used to populate the two new columns, the names of which came from the values in the original `year` column.

Now let's get this wide data frame back to its original format. Note that we have to add the optional argument `values_drop_na = TRUE` because the original transformation resulted in rows with `NA`.

```r
songs_long <- songs_wide %>%
    pivot_longer(cols = c("1960", "1961"),
                 names_to = "year",
                 values_to = "no",
                 values_drop_na = TRUE)
head(songs_long)
```

```
## # A tibble: 6 x 4
##   title                     artist              year  no
##   <chr>                     <chr>               <chr> <chr>
## 1 Theme From A Summer Place Percy Faith         1960  1
## 2 He'll Have To Go          Jim Reeves          1960  2
## 3 Cathy's Clown             The Everly Brothers 1960  3
## 4 Running Bear              Johnny Preston      1960  4
## 5 Teen Angel                Mark Dinning        1960  5
## 6 I'm Sorry                 Brenda Lee          1960  6
```

Now the data frame looks like it originally did, albeit with `songs_long$no` as the final column rather than the first.

Note that you can rename the column where the data will go in the final `values_to` argument. For example, rather than "no", you want the column with the rank to be called "rank." You can also use `select()` to re-order the columns.

```r
songs_long_rn <- songs_wide %>%
    pivot_longer(
        cols = c("1960", "1961"),
        names_to = "year",
        values_to = "rank",
        values_drop_na = TRUE) %>%
    select(rank, everything())
head(songs_long_rn)
```

```
## # A tibble: 6 x 4
##   rank  title                     artist              year
##   <chr> <chr>                     <chr>               <chr>
## 1 1     Theme From A Summer Place Percy Faith         1960
## 2 2     He'll Have To Go          Jim Reeves          1960
## 3 3     Cathy's Clown             The Everly Brothers 1960
## 4 4     Running Bear              Johnny Preston      1960
## 5 5     Teen Angel                Mark Dinning        1960
## 6 6     I'm Sorry                 Brenda Lee          1960
```

Note that this code includes the function `everything()` inside `select()`. The effect is to return all columns in their original order, after the columns that have been specified by name. So here, "rank" is followed by everything else.

Bonus practice:

Now that you have some new cleaning skills, it's time to practice!

1. Find a new dataset. (Remember `data()`!)

2. Examine its documentation, structure, and potential sources of messiness.
3. Create a cleaning script.
4. Work through the code to address any issues.
5. Keep a list of other messy aspects of your dataset and share them with me (Sean.M.Smith@rice.edu) for a future workshop!