Data Wrangling and Tidy Data

August 3, 2024

Table of contents


```
rm(list = ls())
library(tidyverse)
library(tidymodels)
#install.packages("randomNames") #cannot have install.packages() in a notebook
library(randomNames)
library(kableExtra)

options(kable_styling_bootstrap_options = c("hover", "striped"))

theme_set(theme_bw())

ikea <- read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/

################################
#Generate Data
################################
set.seed(300)
items <- ikea %>%
  count(category) %>%
```

```
  sample_n(5) %>%
  pull(category)

items_dup <- items %>%
  append(c("Outdoor Furniture", "Nursary Furnature"))

products <- tibble(items = items_dup, prices = c("$125", "$675", "$1,350", "$475", "$725", "$

clients <- tibble(client_id = seq(1, 10, 1),
                  name = randomNames(10),
                  phone_number = paste0("(",
                                        sample(111:777, 10),
                                        ") ",
                                        sample(111:777, 10),
                                        "-",
                                        sample(1111:7777, 10)))

numRows <- 126735

orders <- tibble(client_id = sample(1:10, size = numRows, replace = TRUE), product = sample(

purchase_history <- crossing(client_id = clients$client_id, items = products$items, year = c
  mutate(quantity = 150 + (-1)^(sample(0:1, 280, replace = TRUE))*rpois(280, 50)) %>%
  left_join(products) %>%
  mutate(prices = str_replace(prices, "\\$", ""),
         prices = str_replace(prices, ",", ""),
         prices = as.numeric(prices),
         total_purchased = quantity*prices) %>%
  select(-quantity, - prices)

#purchase_history

historical_purchases <- purchase_history %>%
  group_by(client_id, year) %>%
  summarize(total_value = sum(total_purchased)) %>%
  ungroup() %>%
  pivot_wider(names_from = year, values_from = total_value)

##################################
#End Data Generation
##################################
```

```
#Remove all intermediate variables
rm(list = setdiff(ls(), c("clients", "orders", "products", "purchase_history", "historical_pu
```

Objectives

This notebook gives an overview of some of the most common *problems* that data come with, and some methods for dealing with those issues.

- Inconsistent data recording issues
- Unexpected data types
- Multi-variable columns
- Wide- and Long-format data
- Data across multiple tables

Set-Up

Run all of the code in the initial code chunk, line by line. You will likely encounter an error on line 20 because you don't have the `randomNames` package installed on your machine. In line 19, un-comment `install.packages("randomNames")` and run it. Now re-run line 20 and continue running each line in the code chunk. You can examine the code if you like, but it is not necessary.

The code generates several tables of data, exhibiting some common challenges when dealing with data. The result of running the code in that chunk will be five tables in your *Global Environment*: `clients`, `orders`, `products`, `purchase_history`, and `historical_purchases`. Use your basic data exploration functions (`glimpse()`, `head()`, `tail()`, `summary()`, etc.) to understand these data frames.

Challenge 1: Inconsistent Data Recording

Take a look at the `products` data frame. The data frame only has seven rows, but among those rows are seemingly duplicated records. Luckily, while the item categories are duplicated (with different spelling and capitalization), the prices are consistent. We have several options available:

```
products %>%
  kable() %>%
  kable_styling()
```

items	prices
TV & media furniture	$125
Outdoor furniture	$675
Beds	$1,350
Nursery furniture	$475
Sideboards, buffets & console tables	$725
Outdoor Furniture	$675
Nursary Furnature	$475

- Manually drop the duplicated rows
  - Our data frame is small enough for this, but with a larger data frame, this might be tedious.
  - If this course of action is to be taken, we must make sure that we are not accidentally deleting data.
- We can use `case_when()` to manually override the values in some cells.

There are certainly other things we could do as well. These two approaches are shown below.

```
#Filter out the duplicated rows
products %>%
  filter(!(items %in% c("Outdoor Furniture", "Nursary Furnature"))) %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

items	prices
TV & media furniture	$125
Outdoor furniture	$675
Beds	$1,350
Nursery furniture	$475
Sideboards, buffets & console tables	$725

```
#Use Case/When to overwrite values and then filter to distinct rows
products %>%
  mutate(items = case_when(
    items == "Outdoor Furniture" ~ "Outdoor furniture",
    items == "Nursary Furnature" ~ "Nursery furniture",
    TRUE ~ items
```

```
  )) %>%
  distinct() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

items	prices
TV & media furniture	$125
Outdoor furniture	$675
Beds	$1,350
Nursery furniture	$475
Sideboards, buffets & console tables	$725

The `case_when()` approach is more applicable than simply removing the rows, because it isn't often that we would actually like to remove rows. We actually do mean to remove rows here, though. Choose your favorite method and overwrite the `products` data frame with a version of itself that does not have the duplicated rows.

```
#update the products data frame here
```

Challenge 2: Unexpected Data Types

Perhaps you've noticed that the `prices` column in the `products` data frame, is being interpreted as a *character* data type. We would prefer for that column to be a numeric variable so that we can run computations with it. There are two issues causing the `prices` to be characters – the dollar sign ($) and the comma separator. We'll need to remove those and convert the column to a numeric value.

```
products %>%
  kable() %>%
  kable_styling()
```

items	prices
TV & media furniture	$125
Outdoor furniture	$675
Beds	$1,350
Nursery furniture	$475
Sideboards, buffets & console tables	$725

Outdoor Furniture	$675
Nursary Furnature	$475

We'll make use of the `str_replace()` function for this. This function allows us to search for a sub-string and, if it is found, replace it with a new substring. We'll replace any found dollar signs and commas with nothing…effectively erasing them. Once this is done, we'll use `as.numeric()` to change the data type. As a reminder, we are changing an existing column in our data frame here, so we'll make use of `mutate()`.

```
products %>%
  mutate(
    prices = str_replace(prices, "\\$", ""),
    prices = str_replace(prices, ",", ""),
    prices = as.numeric(prices)
  )  %>%
  kable() %>%
  kable_styling()
```

items	prices
TV & media furniture	125
Outdoor furniture	675
Beds	1350
Nursery furniture	475
Sideboards, buffets & console tables	725
Outdoor Furniture	675
Nursary Furnature	475

The dollar sign is a special character in `R`, so we needed to escape it with `\\$` to convey that we want to match a literal dollar sign. Edit the code cell above so that the changes you made to the `prices` column are saved and the `products` data frame is updated.

Challenge 3: Multivariable Columns

We'll take a look at the `clients` data frame. The `name` column contains `lastName, firstName` for each client.

```
clients %>%
  kable() %>%
  kable_styling()
```

client_id	name	phone_number
1	Taylor, Trevor	(284) 350-2197
2	Silva, Jonathan	(768) 647-3338
3	De Lara, Maria	(249) 478-2990
4	Knobel, Justin	(699) 237-5637
5	Weber, Kevin	(243) 263-2235
6	el-Mattar, Najma	(738) 441-7703
7	Campbell, Justin	(750) 177-2744
8	al-Barakat, Shamaail	(584) 518-7284
9	Schell, Jakob	(332) 722-4831
10	Fields, Briana	(417) 162-2994

Perhaps we'd like to have separate columns for `first_name` and `last_name`. We can use the `separate()` function for this.

```
clients %>%
  separate(name, into = c("last_name", "first_name"), sep = ", ") %>%
  kable() %>%
  kable_styling()
```

client_id	last_name	first_name	phone_number
1	Taylor	Trevor	(284) 350-2197
2	Silva	Jonathan	(768) 647-3338
3	De Lara	Maria	(249) 478-2990
4	Knobel	Justin	(699) 237-5637
5	Weber	Kevin	(243) 263-2235
6	el-Mattar	Najma	(738) 441-7703
7	Campbell	Justin	(750) 177-2744
8	al-Barakat	Shamaail	(584) 518-7284
9	Schell	Jakob	(332) 722-4831
10	Fields	Briana	(417) 162-2994

This function will separate over all instances of the separator, so we'll need a new column name for each of the resulting parts. As a reminder, we use `c()` in `R` when providing a list of values.

Challenge 4: Wide- and Long-Format Data

Tidy data is data that satisfies the following properties:

- Every column is a measured variable.
- Each row corresponds to a single record or observational unit.

Data is said to be in a wide format if columns aren't necessarily measured variables. Instead, the columns themselves correspond to a variable. While we typically prefer long-format data for analysis, there are certainly uses for wide-formats. In particular, financial data is usually presented or stored in wide format. Additionally, it is possible for data to be *too* long. We can use `pivot_longer()` to move from wide-format to long format and we can use `pivot_wider()` to move from long format to wide.

Let's take a look at the `historical_purchases` data frame.

```
historical_purchases %>%
  head()  %>%
  kable() %>%
  kable_styling()
```

client_id	2018	2019	2020	2021
1	833875	791600	673325	640800
2	713725	800850	566900	662300
3	662375	575775	514150	597475
4	563050	814400	615950	645775
5	684500	708400	604825	835100
6	644025	734475	809775	759575

That data frame contains purchase totals for each client across multiple years. The data is in *wide* format because, for example `"2018"` is not a variable we can measure for each `client_id`. That column name, `"2018"` is the value of a hidden `year` variable. We'll use `pivot_longer()` to transform our data, creating a `year` column and `total_purchases` column.

```
historical_purchases %>%
  pivot_longer(-client_id, names_to = "year", values_to = "total_purchases") %>%
  head()  %>%
  kable() %>%
  kable_styling()
```

client_id	year	total_purchases
1	2018	833875
1	2019	791600
1	2020	673325
1	2021	640800
2	2018	713725
2	2019	800850

Now our data frame has become *longer*! The `year` variable is currently stored as a character, we can convert it to a number using `mutate()` and `as.numeric()`.

```
historical_purchases %>%
  pivot_longer(-client_id, names_to = "year", values_to = "total_purchases") %>%
  mutate(year = as.numeric(year)) %>%
  head()  %>%
  kable() %>%
  kable_styling()
```

client_id	year	total_purchases
1	2018	833875
1	2019	791600
1	2020	673325
1	2021	640800
2	2018	713725
2	2019	800850

Just so you can see how `pivot_wider()` works, let's pivot this tidy data frame back to a wide format, but include a column for each `client_id` rather than `year`.

```
historical_purchases %>%
  pivot_longer(-client_id, names_to = "year", values_to = "total_purchases") %>%
  mutate(year = as.numeric(year)) %>%
  pivot_wider(id_cols = year, names_from = client_id, values_from = total_purchases) %>%
  kable() %>%
  kable_styling()
```

year	1	2	3	4	5	6	7	8	9	10
2018	833875	713725	662375	563050	684500	644025	759025	626100	716350	537425
2019	791600	800850	575775	814400	708400	734475	556975	594625	630225	433850
2020	673325	566900	514150	615950	604825	809775	734250	662200	761675	575950
2021	640800	662300	597475	645775	835100	759575	773275	715150	634100	735050

Moving between wide and long formats is a pretty useful skill. I use it often when I am summarizing data and would like to display a summary table in a report. These summary tables sometimes look more pleasing and are easier to digest in a wide format.

Data Across Multiple Tables

It isn't often the case that all of your data is neatly confined to a single table. In fact, there are very good database-design and infrastructure reasons for this not to be the case. Database Engineers and Administrators work very hard to *normalize* their databases. This normalization doesn't have anything to do with finding means or scaling – it is a design strategy that removes redundancy and reduces storage size requirements.

The advantage to space-saving is fairly obvious, but the redundancy-reduction may not be. Consider, for example that one of our clients updated their phone number. If we had all of our data stored in a single table, then we would need to update their phone number in every single record corresponding to that client. Because our data has been normalized, however, the only place phone numbers are stored is in the `clients` data frame. We only need to update that client's phone number in that one small table, because that is the only place the phone number has been stored.

The drawback to working with data stored this way is that we'll need to pull information from across multiple tables in order to obtain the data we'd like to work with. For example, perhaps we'd like a version of the `orders` data frame which includes the ordering client's name and phone number, along with the total dollar value of the order they made. The data we need is spread across the `orders`, `clients`, and `products` data frames.

```
orders %>%
  head() %>%
  kable() %>%
  kable_styling()
```

client_id	product	quantity
1	Outdoor furniture	2

2	Outdoor furniture	4
9	TV & media furniture	4
4	Beds	2
3	TV & media furniture	2
5	Outdoor furniture	2

```
clients  %>%
  kable() %>%
  kable_styling()
```

client_id	name	phone_number
1	Taylor, Trevor	(284) 350-2197
2	Silva, Jonathan	(768) 647-3338
3	De Lara, Maria	(249) 478-2990
4	Knobel, Justin	(699) 237-5637
5	Weber, Kevin	(243) 263-2235
6	el-Mattar, Najma	(738) 441-7703
7	Campbell, Justin	(750) 177-2744
8	al-Barakat, Shamaail	(584) 518-7284
9	Schell, Jakob	(332) 722-4831
10	Fields, Briana	(417) 162-2994

```
products %>%
  kable() %>%
  kable_styling()
```

items	prices
TV & media furniture	$125
Outdoor furniture	$675
Beds	$1,350
Nursery furniture	$475
Sideboards, buffets & console tables	$725
Outdoor Furniture	$675
Nursary Furnature	$475

We can use "joins" to bring all of this information into a single table using the following steps.

- Begin with the `orders` table.
- Add columns for the client's name and phone number by *joining* the `clients` data frame onto the `orders` table.
- Add a column for product `prices` by *joining* the `products` data frame onto the resulting table.
- Use `mutate()` to compute the total dollar value for each order.

There are lots of different types of joins that can be made between data frames. I find that what I usually want is a `left_join()` – that is, start with the table on the left and lets join new variables on to it. We'll be using that here. The `left_join()` will add new variables on to our data frame by matching values on shared columns (or columns that we explicitly define as shared via the `by` argument).

```
orders %>%
  left_join(clients) %>%
  left_join(products, by = c("product" = "items")) %>%
  head()  %>%
  kable() %>%
  kable_styling()
```

```
Joining with `by = join_by(client_id)`
```

client_id	product	quantity	name	phone_number	prices
1	Outdoor furniture	2	Taylor, Trevor	(284) 350-2197	$675
2	Outdoor furniture	4	Silva, Jonathan	(768) 647-3338	$675
9	TV & media furniture	4	Schell, Jakob	(332) 722-4831	$125
4	Beds	2	Knobel, Justin	(699) 237-5637	$1,350
3	TV & media furniture	2	De Lara, Maria	(249) 478-2990	$125
5	Outdoor furniture	2	Weber, Kevin	(243) 263-2235	$675

When using joins, it is important to verify that everything has worked as you expected. While not fool-proof, I usually check to ensure that the number of columns and number of rows in the resulting data frame are what I expected.

Summary

There's lots of ways our data can be inconvenient. Wrangling data is a really important skill to gain. This notebook gave you exposure to some of the most common data wrangling/cleaning tasks.

- Using `case_when()` to update individual values within a column, using some criteria.
- Using `distinct()` to remove duplicate rows in a data frame.
  - Be careful when removing rows of data. Make sure that you are truly removing duplicate records rather than distinct records which are identical by chance.
- Using `str_replace()` to remove or replace characters in a column.
- Using `as.numeric()` to convert a column to a numeric data type.
  - We can also use `as.character()` to convert a column to a string data type or `as.factor()` to convert explicitly to a categorical variable.
- Using `separate()` in the case that a column contains more than one variable.
  - We can use `unite()` to combine multiple columns into a single column.
- Use `pivot_longer()` to convert from wide-format to long-format.
  - We can use `pivot_wider()` to convert from long to wide.
- If our data is split across multiple tables, we can use `*_join()` to join variables from multiple tables together into a single table.
  - There are lots of varieties of join – the `left_join()` begins with the table on the left and adds variables on to it. This is by far the most frequent join that I've used.

This notebook hasn't covered every wrangling tool you'll need, but it has given you a toolbox that will help in a variety of common scenarios.