

15.2.5 Transform, Group, and Reshape Data Using the Tidyverse Package

Jeremy is really starting to get the hang of this R language! Colleen, Jeremy's go-to source of advice for all things R, suggests he explore the tidyverse package of libraries to help him transform, group, and reshape his data.

One of the most successful optimized packages for R is the [tidyverse](https://www.tidyverse.org/) (<https://www.tidyverse.org/>) package. The tidyverse package contains libraries such as dplyr, tidyr, and ggplot2. These packages work together to help simplify the process of creating transformed data columns, grouping data using factors, reshaping our two-dimensional data structures, and visualizing our results using plots.

Throughout this module and beyond, you'll find more and more uses for the tidyverse libraries as you become more comfortable with R and begin generating more complex and robust RScripts. For now, we'll concentrate on leveraging the tidyverse libraries to help us transform, group, and reshape our R data frames.

Transform

Raw data is often insufficient in telling the full story. Usually when we are analyzing data, we want to perform calculations and incorporate the calculations back into the raw data to ease in downstream analysis. Tidyverse's [dplyr](https://dplyr.tidyverse.org/) (<https://dplyr.tidyverse.org/>) library transforms R data.

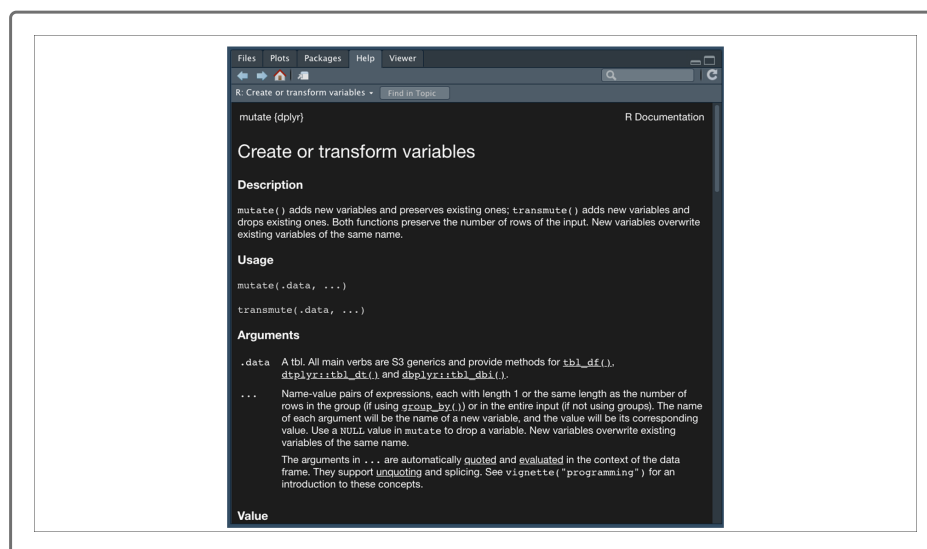
The dplyr library contains a wide variety of functions that can be chained together to transform data quickly and easily. Using dplyr is slightly more

complex than the simple assignment statement in R because dplyr allows the user to chain together functions in a single statement using their own pipe operator (`%>%`).

By chaining functions together, the user does not need to assign intermediate vectors and tables. Instead, all of the data transformation can be performed in a single assignment function that is easy to read and interpret. To transform a data frame and include new calculated data columns, we'll use the `mutate()` function.

Type the following code into the R console to look at the `mutate()` documentation in the Help pane:

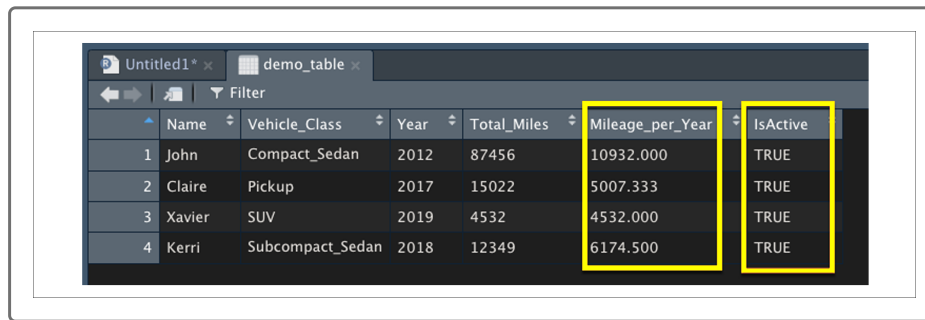
```
> library(tidyverse)
> ?mutate()
```



The documentation for the `mutate()` function (generally any dplyr function) is a little obscure, but it makes more sense looking at the examples. We can think of the `mutate()` function as a series of smaller assignment statements that are separated by commas. Each of the assigned names appears as a new column in our raw data frame.

For example, if we want to use our coworker vehicle data from the `demo_table` and add a column for the mileage per year, as well as label all vehicles as active, we would use the following statement:

```
> demo_table <- demo_table %>% mutate(Mileage_per_Year=Total_Miles/(2020-Yea
```



| | Name | Vehicle_Class | Year | Total_Miles | Mileage_per_Year | IsActive |
|---|--------|------------------|------|-------------|------------------|----------|
| 1 | John | Compact_Sedan | 2012 | 87456 | 10932.000 | TRUE |
| 2 | Claire | Pickup | 2017 | 15022 | 5007.333 | TRUE |
| 3 | Xavier | SUV | 2019 | 4532 | 4532.000 | TRUE |
| 4 | Kerri | Subcompact_Sedan | 2018 | 12349 | 6174.500 | TRUE |

For each name and expression, a new column is generated within our returned data frame. This returned data frame is then assigned to our original data structure `demo_table`.

This process of mutating and transforming columns can be applied at any time (or even multiple times) within a dplyr pipe, and it can even reference columns that were generated from previous `mutate()` functions. These transformed data frames can be used in further downstream analysis, visualizations, and modeling. However, transformed data frames are not designed to summarize data. To summarize our data frames in R, we'll need to use a grouping function.

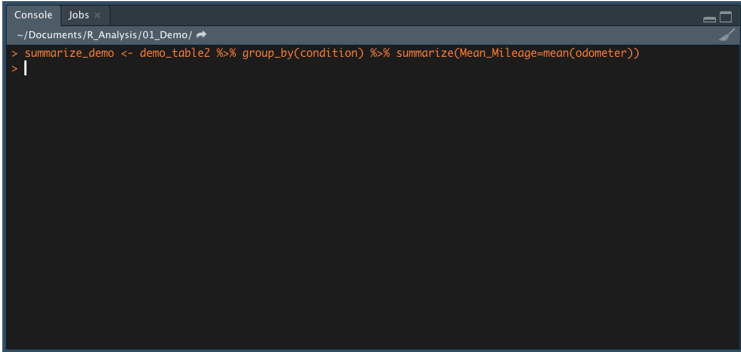
Group Data

Just like in Python, grouping across a factor allows us to quickly summarize and characterize our dataset around a factor of interest (also known as a character vector in R, or list of strings in Python). The most straightforward way to perform a grouping on an R data frame is to use dplyr's `group_by()` function. The `group_by()` function tells dplyr which factor (or list of factors in order) to group our data frame by.

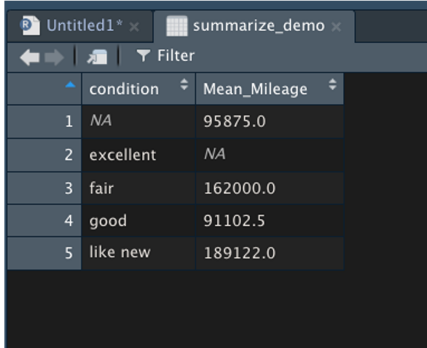
The behavior of `group_by()` is almost identical to that of the Pandas `DataFrame.groupby()` function, with the exception of using a value or vector instead of a list. Once we have our `group_by` data structure, we use the dplyr `summarize()` function to create columns in our summary data frame.

For example, if we want to group our used car data by the condition of the vehicle and determine the average mileage per condition, we would use the following dplyr statement:

```
> summarize_demo <- demo_table2 %>% group_by(condition) %>% summarize(Mean_M
```



```
~/Documents/R_Analysis/01_Demo/
> summarize_demo <- demo_table2 %>% group_by(condition) %>% summarize(Mean_Mileage=mean(odometer))
> |
```



| | condition | Mean_Mileage |
|---|-----------|--------------|
| 1 | NA | 95875.0 |
| 2 | excellent | NA |
| 3 | fair | 162000.0 |
| 4 | good | 91102.5 |
| 5 | like new | 189122.0 |

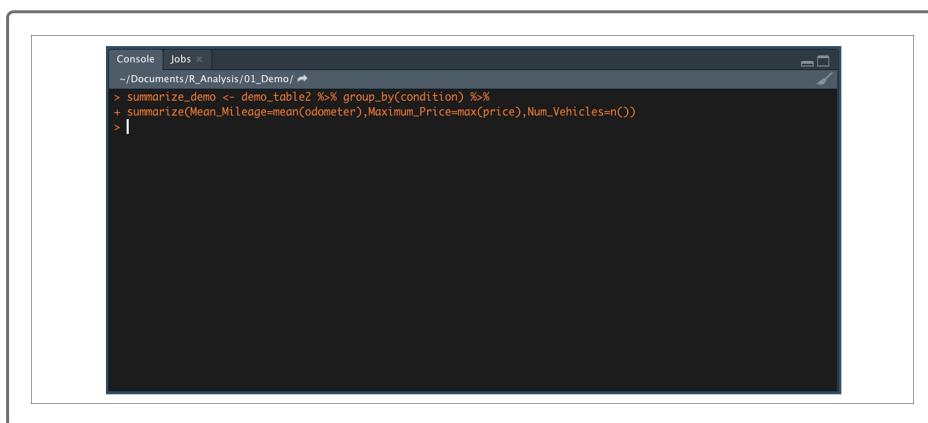
NOTE

Dplyr's pipe operator allows us to move our connected functions to a new line while computing everything in the same assignment function. This allows you to write RScripts that are not too wide for your RScript window.

Using the `summarize()` function is very similar to using the `mutate()` function on a raw data frame: for each name and summary function, we create a column in a summary data frame. Our simplest summary functions will use statistics summary functions such as `mean()`, `median()`, `sd()`, `min()`, `max()`, and `n()` (used to calculate the number of rows in each category).

However, the dplyr `summarize()` documentation provides a more comprehensive list of functions that can be used to summarize our data. For example, if in addition to our previous summary table we wanted to add the maximum price for each condition, as well as add the vehicles in each category, our statement would look as follows:

```
> summarize_demo <- demo_table2 %>% group_by(condition) %>% summarize(Mean_M
```

A screenshot of an RStudio interface showing a data frame named 'summarize_demo'. The table has 5 rows and 4 columns: 'condition', 'Mean_Mileage', 'Maximum_Price', and 'Num_Vehicles'.

| | condition | Mean_Mileage | Maximum_Price | Num_Vehicles |
|---|-----------|--------------|---------------|--------------|
| 1 | NA | 95875.0 | 11995 | 2 |
| 2 | excellent | NA | 14500 | 2 |
| 3 | fair | 162000.0 | 2250 | 1 |
| 4 | good | 91102.5 | 26700 | 4 |
| 5 | like new | 189122.0 | 9995 | 1 |

NOTE

The `summarize()` function takes an additional argument, `.groups`. This allows you to control the the grouping of the result. The four possible values are:

- `.groups = "drop_last"` drops the last grouping level (default)
- `.groups = "drop"` drops all grouping levels and returns a tibble
- `.groups = "keep"` preserves the grouping of the input
- `.groups = "rowwise"` turns each row into its own group

Reshape Data

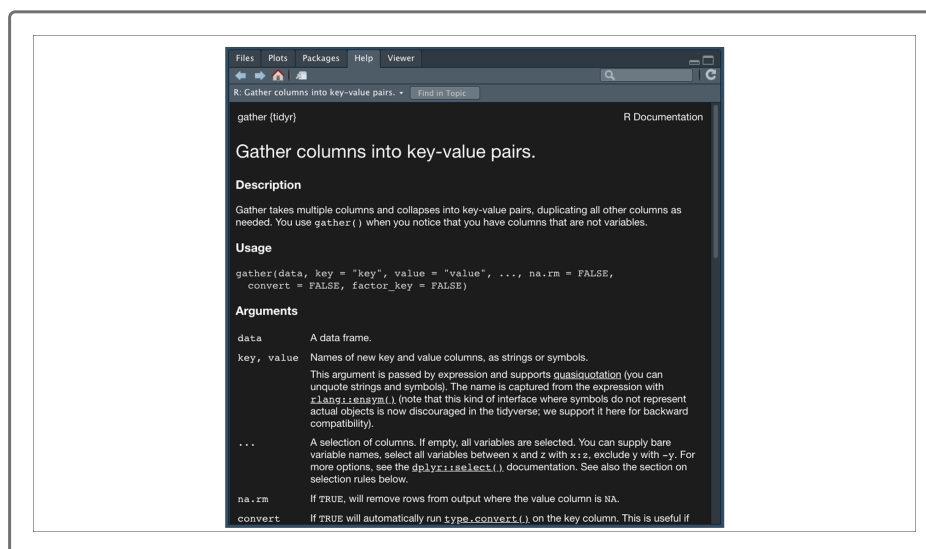
When performing more involved data analytics and visualizations, there may be situations where the shape and design of our data frame is overcomplicated or incompatible with the libraries and functions we wish to use. For example, a "wide" data frame with few rows and many columns

and factors may be difficult to visualize. Or a "long" data frame may be difficult to analyze if it contains multiple rows for a single subject.

Thankfully, the `tidyr` library from the tidyverse has the `gather()` and `spread()` functions to help reshape our data. The `gather()` function is used to transform a wide dataset into a long dataset.

Type the following code into the R console to look at the `gather()` documentation in the Help pane:

```
> ?gather()
```



NOTE

In addition to seeing `gather()` in use outside of class, you may also encounter the functions `pivot_longer()` and `pivot_wider()`.

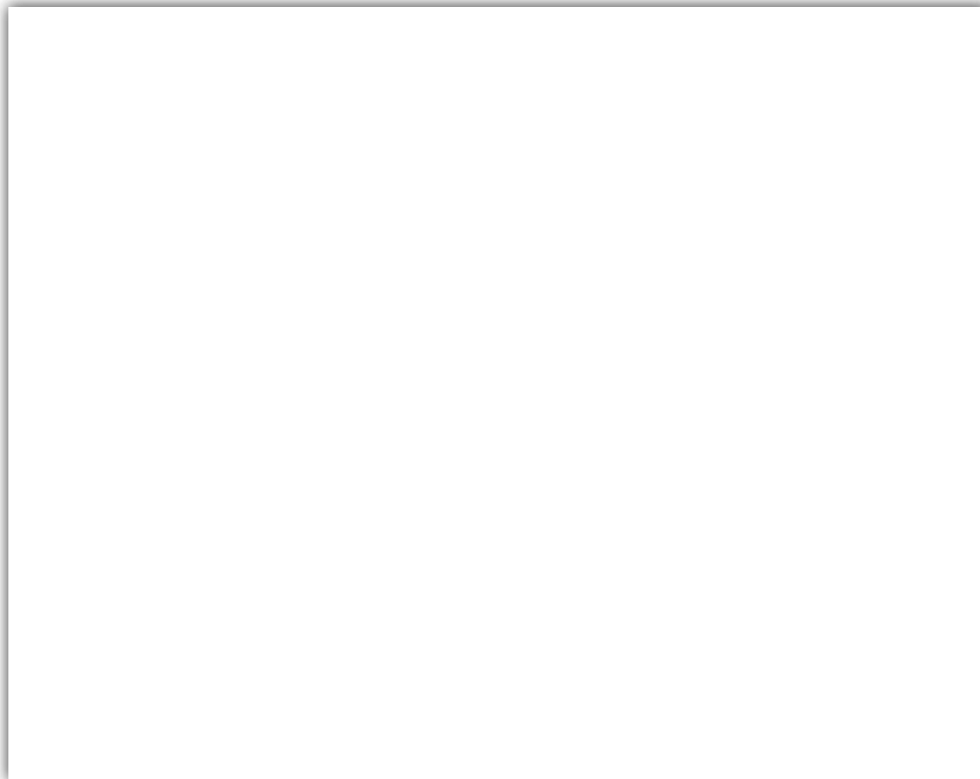
`pivot_longer()` lengthens the data by increasing the number rows and decreasing the number of columns, while `pivot_wider()` will perform an inverse transformation.

For additional information, visit the [official documentation](https://tidyr.tidyverse.org/reference/pivot_longer.html) (https://tidyr.tidyverse.org/reference/pivot_longer.html) for these functions.

To properly reshape an R data frame, the `gather()` function requires a few arguments:

- **data**

- **key**
- **value**
- ...



One of the easiest ways to learn how to reshape data is to practice with a simple dataset.

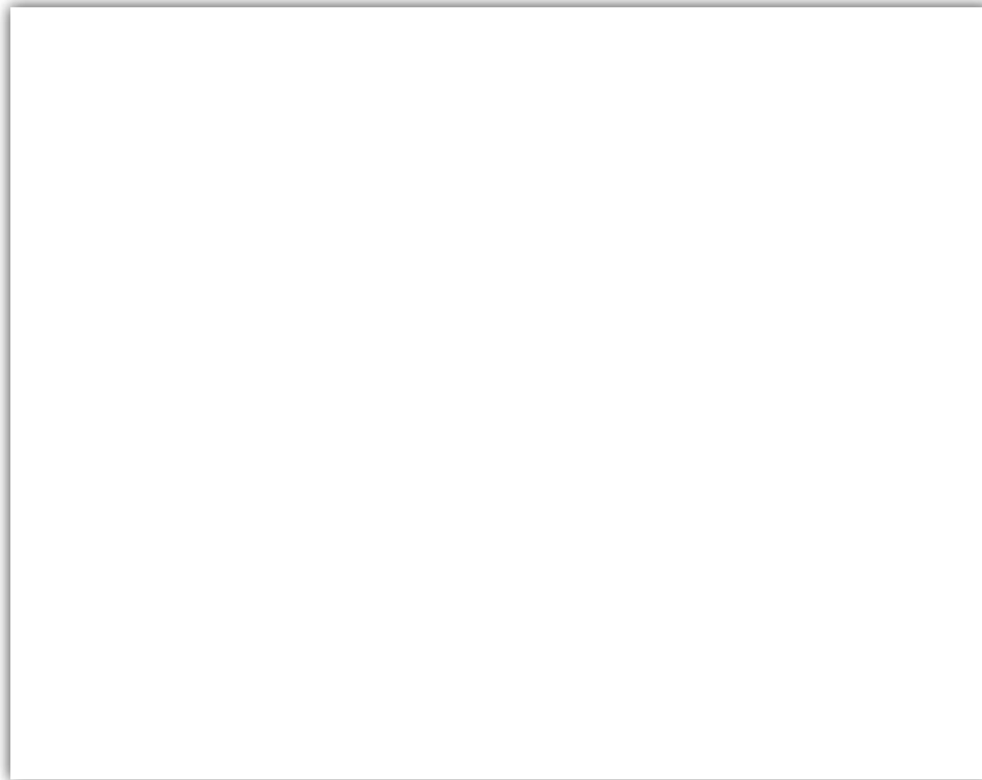
For this example, first [download the demo2.csv](https://2u-data-curriculum-team.s3.amazonaws.com/dataviz-online/module_15/demo2.csv) (https://2u-data-curriculum-team.s3.amazonaws.com/dataviz-online/module_15/demo2.csv).

Once you have put the file in your active directory, load the `demo2.csv` file into your R environment and look at the top of the data frame:

```
> demo_table3 <- read.csv('demo2.csv', check.names = F, stringsAsFactors = F)
```

| | Vehicle | buying_price | maintenance_cost | number_of_doors | number_of_seats | luggage_boot_size | safety_rating |
|---|---------|--------------|------------------|-----------------|-----------------|-------------------|---------------|
| 1 | 1 | 3 | 2 | 4 | 2 | 2 | 2 |
| 2 | 2 | 3 | 2 | 2 | 5 | 2 | 1 |
| 3 | 3 | 1 | 4 | 2 | 5 | 1 | 3 |
| 4 | 4 | 4 | 4 | 2 | 2 | 1 | 2 |
| 5 | 5 | 3 | 3 | 3 | 4 | 3 | 3 |
| 6 | 6 | 2 | 1 | 2 | 2 | 1 | 1 |
| 7 | 7 | 1 | 3 | 5 | 2 | 2 | 2 |
| 8 | 8 | 3 | 1 | 2 | 4 | 3 | 2 |

For the following checkpoint [download Vehicle Data.xlsx](https://2u-data-curriculum-team.s3.amazonaws.com/dataviz-online/module_15/Vehicle_Data.xlsx) [\(https://2u-data-curriculum-team.s3.amazonaws.com/dataviz-online/module_15/Vehicle_Data.xlsx\)](https://2u-data-curriculum-team.s3.amazonaws.com/dataviz-online/module_15/Vehicle_Data.xlsx)



The `demo_table3` data frame contains survey results from 250 vehicles that were collected by a rental company. The rental company evaluated seven different metrics for each vehicle and rated each metric on a scale of 1 to 5, with 1 representing low and 5 representing high.

This data would be considered a wide format because different metrics were collected from a single vehicle, and each metric (also known as a variable in this case) was stored as a separate column. To change this dataset to a long format, we would use `gather()` to reshape this dataset.

Type the following function into the R console:

```
> long_table <- gather(demo_table3, key="Metric", value="Score", buying_price:p
```

Alternatively, you may type the following function in the R console:

```
> long_table <- demo_table3 %>% gather(key="Metric", value="Score", buying_pri
```

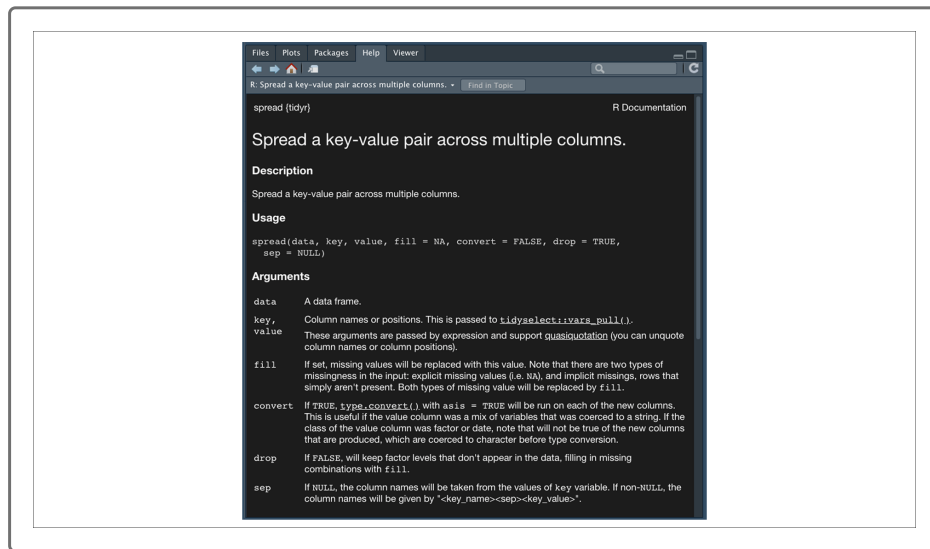

| | Vehicle | safety_rating | Metric | Score |
|----|---------|---------------|--------------|-------|
| 1 | 1 | 2 | buying_price | 3 |
| 2 | 2 | 1 | buying_price | 3 |
| 3 | 3 | 3 | buying_price | 1 |
| 4 | 4 | 2 | buying_price | 4 |
| 5 | 5 | 3 | buying_price | 3 |
| 6 | 6 | 1 | buying_price | 2 |
| 7 | 7 | 2 | buying_price | 1 |
| 8 | 8 | 2 | buying_price | 3 |
| 9 | 9 | 1 | buying_price | 1 |
| 10 | 10 | 2 | buying_price | 2 |
| 11 | 11 | 1 | buying_price | 3 |
| 12 | 12 | 2 | buying_price | 3 |
| 13 | 13 | 1 | buying_price | 4 |
| 14 | 14 | 3 | buying_price | 1 |
| 15 | 15 | 1 | buying_price | 2 |

By using the `gather()` function, we have collapsed all of the survey metrics into one Metric column and all of the values into a Score column. Because the Vehicle column was not in the arguments, it was treated as an identifier column and added to each row as a unique identifier.

Alternatively, if we have data that was collected or obtained in a long format, we can use tidy's `spread()` function to spread out a variable column of multiple measurements into columns for each variable.

Type the following code into the R console to look at the `spread()` documentation in the Help pane:

```
> ?spread()
```



To properly reshape an R data frame, the `spread()` function requires a few arguments:

- **data**
- **key**
- **value**
- **fill**

Therefore, if we want to spread out our previous long-format data frame back to its original format, we would use the following `spread()` statement:

```
> wide_table <- long_table %>% spread(key="Metric",value="Score")
```

And if we want to check if our newly created wide-format table is exactly the same as our original `demo_table3`, we can use R's `all.equal()` function:

```
Console ~/Documents/R_Analysis/01_Demo/ ➔
> all.equal(demo_table3,wide_table)
[1] "Names: 7 string mismatches"
[2] "Component 2: Mean relative difference: 0.5808824"
[3] "Component 3: Mean relative difference: 0.8276762"
[4] "Component 4: Mean relative difference: 0.5555556"
[5] "Component 5: Mean relative difference: 0.4978166"
[6] "Component 6: Mean relative difference: 0.4863636"
[7] "Component 7: Mean relative difference: 1.818182"
[8] "Component 8: Mean relative difference: 0.5152355"
```

IMPORTANT

If you ever compare two data frames that you expect to be equal, and the `all.equal()` function tells you they're not as the above example shows. Try sorting the columns of both data frames using the `order()` and `colnames()` functions and bracket notation:

```
> table <-demo_table3[,order(colnames(wide_table))]
```

Or, sort the columns using the `colnames()` functions and bracket notation only:

```
> table <- demo_table3[, (colnames(wide_table))]
```

(The comma in the bracket indicates that we're selecting all rows.)

Now that we have learned about selecting and manipulating our data in R, it's time to learn how to use R for data analysis. We'll begin by visualizing our datasets using ggplot2.

© 2020 - 2022 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.