Data Frame Manipulation

Juliana

October 31, 2017

Introduction

The purpose of this post is to expand upon the topic of data frame manipulation. I go about doing this by focusing on particular sets of functions. The topic of data frame manipulation is itself a very broad topic that encompasses an innumerable amount of examples and specific applications; I, therefore, chose a small variety of functions that I felt are important for any R user to know.

Discussion

I would first like to start this post by discussing a data frame application that I came across while re-looking into the base R function aggregate().

When learning R, I think it is important to be aware of both the differences and similarities between base R functions and functions that are provided by R packages.

This first example I will work with demonstrates a pretty simple application of the aggregate() function. Therefore, I will be reviewing this particular example not to show how aggregate() works, per se, but to demonstrate how group_by() and summarise() can be used to produce the same data frame. I think this is important to point out simply because it's easy to lose sight of the fact–especially if one always uses "dplyr" functions–that aggregate() can essentially do the same thing that group_by() and summarise() can do together. The aggregate() function has not been used too much in Stat 133, so I wanted to make this particular relationship between a base R function and "dplyr" functions very clear.

Relationship Between base R Function aggregate() and "dplyr" Functions group_by() and summarise()

Let's start by creating a simple data frame. Don't forget the `stringAsFactors = FALSE`!

```
my_pets <- data.frame(pets = c(rep("dog", 4),
                               rep("cat", 6),
                               rep("bunny", 10)),
                      stringsAsFactors = FALSE)

my_pets
```

```
##     pets
## 1    dog
## 2    dog
## 3    dog
## 4    dog
## 5    cat
## 6    cat
## 7    cat
## 8    cat
## 9    cat
## 10   cat
## 11 bunny
## 12 bunny
## 13 bunny
## 14 bunny
## 15 bunny
## 16 bunny
## 17 bunny
## 18 bunny
## 19 bunny
## 20 bunny
```

Now that we have created the data frame, let's suppose that we want to separate our pets into groups and count how many of each pet we have. This is where the aggregate() function will come in.

```
num_of_pets <- aggregate(my_pets, by = list(unique_pets = my_pets$pets), FUN = length)

num_of_pets
```

```
##   unique_pets pets
## 1       bunny   10
## 2         cat    6
## 3         dog    4
```

Voila! We get a new and improved data frame in which we can clearly see that we have 10 bunnies, 6 cats,and 4 dogs.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
num_of_pets2 <- summarise(group_by(my_pets, pets), num_of_each = length(pets))

num_of_pets2
```

```
## # A tibble: 3 x 2
##    pets num_of_each
##   <chr>       <int>
## 1 bunny          10
## 2   cat           6
## 3   dog           4
```

Great! We see that we have gotten the same output. To summarise, we can think of the relationship between these functions as such: the "by" parameter in the arrange() function is equivalent to the group_by() function in "dplyr" while the "FUN" paramter in arrange() is equivalent to summarise().

Creating Dates and Extending our Understanding of aggregate()

I will now be moving on to another example involving a data frame containing dates and which also inolves an application of the aggregate() function. Before doing so, I would like to give credit for this example to David Kun, as it was the example he wrote in his own article on aggregate() that inspired the example presented below. A link to his article can be found in the References section of this blog post.

As you will see below, the function as.Date() can be used to generate dates.

```
# Generating a data frame with 1 year's worth of dates

my_dates <- data.frame(dates = as.Date("2017-01-12", format = "%Y-%m-%d") + 0:365)

head(my_dates, 10)
```

```
##         dates
## 1  2017-01-12
## 2  2017-01-13
## 3  2017-01-14
## 4  2017-01-15
## 5  2017-01-16
## 6  2017-01-17
## 7  2017-01-18
## 8  2017-01-19
## 9  2017-01-20
## 10 2017-01-21
```

I find this example interesting for two reasons. One, because it shows how it is actually quite simple to generate a data frame full of dates. And two, because it shows us that we can generate a year's worth of dates by simply adding a "+" after as.Date() and specifying the number of days that we would like to have in our data frame. In this case, I chose exactly 365.

Now, let's use aggregate() on my_dates.

```
min_date <- aggregate(my_dates, by = list(month = substr(my_dates$dates, 1, 7)), FUN = min)

min_date
```

```
##      month      dates
## 1  2017-01 2017-01-12
## 2  2017-02 2017-02-01
## 3  2017-03 2017-03-01
## 4  2017-04 2017-04-01
## 5  2017-05 2017-05-01
## 6  2017-06 2017-06-01
## 7  2017-07 2017-07-01
## 8  2017-08 2017-08-01
## 9  2017-09 2017-09-01
## 10 2017-10 2017-10-01
## 11 2017-11 2017-11-01
## 12 2017-12 2017-12-01
## 13 2018-01 2018-01-01
```

From the data frame displayed above, we can see that we have grouped the data frame by the month of each individual year and next to each month displayed the earliest possible date for that month. Notice that the date next to "2017-01" and "2018-01" are different. This is because our starting date was "2017-01-12," meaning that the earliest date in my_dates for the month of January in 2017 is "2017-01-12", whereas the earliest date in January of 2018 is "2018-01-01". In order to group by month and year, we use the **substr()** function. The 1 and 7 that are written as substr() parameters tell R to include characters 1 through 7 of each date, which encompass both the year and month. The substr() function in this case is very helpful. Without it, we wouldn't be able to group by month since each date when taken as a whole is unique.

Let's try to generate the same data frame using summarise() and group_by() like we did above!

```
# Generating a data frame with the earliest date for each month

summarise(group_by(my_dates, month = substr(dates, 1, 7)), dates = min(dates))
```

```
## # A tibble: 13 x 2
##      month      dates
##      <chr>     <date>
##  1 2017-01 2017-01-12
##  2 2017-02 2017-02-01
##  3 2017-03 2017-03-01
##  4 2017-04 2017-04-01
##  5 2017-05 2017-05-01
##  6 2017-06 2017-06-01
##  7 2017-07 2017-07-01
##  8 2017-08 2017-08-01
##  9 2017-09 2017-09-01
## 10 2017-10 2017-10-01
## 11 2017-11 2017-11-01
## 12 2017-12 2017-12-01
## 13 2018-01 2018-01-01
```

Awesome! We see once again how aggregate() and the functions group_by() and summarise() are really similar and allow us to produce the same data frame!

Ok, now for another aggregate() example.

Let’s create two data frames. The first will represent the category into which a scoring of a particular vacation spot falls. The second data frame will represent the actual score given to a vacation spot by someone who went there.

```
location_ratings <- data.frame(rankings = c(rep("breathtaking", 7), rep("so-so", 5), rep("ugly", 12)), stringsAsFactors =  FALSE)

raw_scores <- data.frame(scores = c(rnorm(7, mean = 9.8, sd = 0.3), rnorm(5, mean = 5, sd = 0.4), rnorm(12, mean = 1.5, sd = 0.5)), stringsAsFactors = FALSE)
```

Let’s create a data frame in which the scores are grouped into categories.

```
categorized_scores <- aggregate(raw_scores, by = location_ratings, FUN = function(raw_scores){
  max(raw_scores) * 2
})

categorized_scores
```

```
##       rankings    scores
## 1 breathtaking 20.614320
## 2        so-so 11.199507
## 3         ugly  6.075541
```

The main point of this example is to show that you **don’t** have to use the same data frame for the aggregate() function and its “by” parameter. In other words, the column variable or variables by which you decide to group does not have to come from the data frame that is used as the first argument of the aggregate() function. As shown above, you can also create your own function for the FUN parameter.

Untidy Data and “tidyr” Functions

I will noW be moving on to talk discuss what, in the R community, is referred to as untidy data and demonstrate the application of a couple of functions from the “tidyr” package created by Hadley Wickham.

Because in the real world the data files that one works with will not always be in a tidy format in which rows serve as observations and columns as variables, it is important to learn how to transform data into a tidy format.

Let’s start by loading the packages we’ll be working with and importing the data contained within treats.csv. In the data we’ll be looking at, there will be 4 inidividuals, the different types of treats they consume (cake, chocolate, and ice cream), the number of servings they have of each (1 or 2), and the number of miles they run when they consume 1 or 2 servings. In order to display the data in a tidy format, we’d ideally like to have one column for each of the following: names, treats, servings, and miles run.

```
library(tidyr)
```

```
## Warning: package 'tidyr' was built under R version 3.4.2
```

```
library(dplyr)

treats_data <- read.csv("../Post1/treats.csv", stringsAsFactors = FALSE)
treats_data
```

```
##      name cake_1 cake_2 choco_1 choco_2 icream_1 icream_2
## 1  Samara    3.0   6.00     1.0       4      5.0        9
## 2   Nancy    0.0   2.00     0.5       3      1.0        4
## 3 Sheldon    0.5   0.75     1.0       2      1.5        2
## 4 Brianna    4.0   6.00     2.0       5      6.0       10
```

After importing treats.csv, we can see that the data is not in a tidy format. To begin with, there isn’t a miles column to indicate the number of miles that each individual runs. Instead, every row is made up exclusively of values representing the number of miles run by each individual. We also see that the the type of treat and number of servings have been combined in the header row. Additionally, instead of having just one column or variable for all of the treats, we see that each treat-serving combination serves as a column of its own.

Let's start tidying up the data!

This is where the gather() function comes in.

```
treats_data2 <- gather(treats_data, key = treat_serving, value = miles, cake_1:icream_2)

treats_data2
```

```
##       name treat_serving miles
## 1   Samara        cake_1  3.00
## 2    Nancy        cake_1  0.00
## 3  Sheldon        cake_1  0.50
## 4  Brianna        cake_1  4.00
## 5   Samara        cake_2  6.00
## 6    Nancy        cake_2  2.00
## 7  Sheldon        cake_2  0.75
## 8  Brianna        cake_2  6.00
## 9   Samara       choco_1  1.00
## 10   Nancy       choco_1  0.50
## 11 Sheldon       choco_1  1.00
## 12 Brianna       choco_1  2.00
## 13  Samara       choco_2  4.00
## 14   Nancy       choco_2  3.00
## 15 Sheldon       choco_2  2.00
## 16 Brianna       choco_2  5.00
## 17  Samara      icream_1  5.00
## 18   Nancy      icream_1  1.00
## 19 Sheldon      icream_1  1.50
## 20 Brianna      icream_1  6.00
## 21  Samara      icream_2  9.00
## 22   Nancy      icream_2  4.00
## 23 Sheldon      icream_2  2.00
## 24 Brianna      icream_2 10.00
```

As you can see, the gather() function produces what, in the R help documentation, are referred to as "key-value pairs". What we set equal to key and value become the names of the new columns we create, as we can see in the data frame above. The code that comes after the value assignment (in this case, "cake_1:icream_2") will become the input to the new column created by the key assignment. In other words, what were previously column/variable names will become the values of one column, namely the column whose name has been assigned to key. This is why we see a column named "treat_serving" that contains all of the treat-serving combinations. The values that fall under the value column, which in our case is called "miles," are those values that had initially been listed under the variables that now makeup the key column.

The data frame definitely looks better, but we still need to separate the type of treat from the number of servings consumed in a day. In order to do this, we will be using the separate() function.

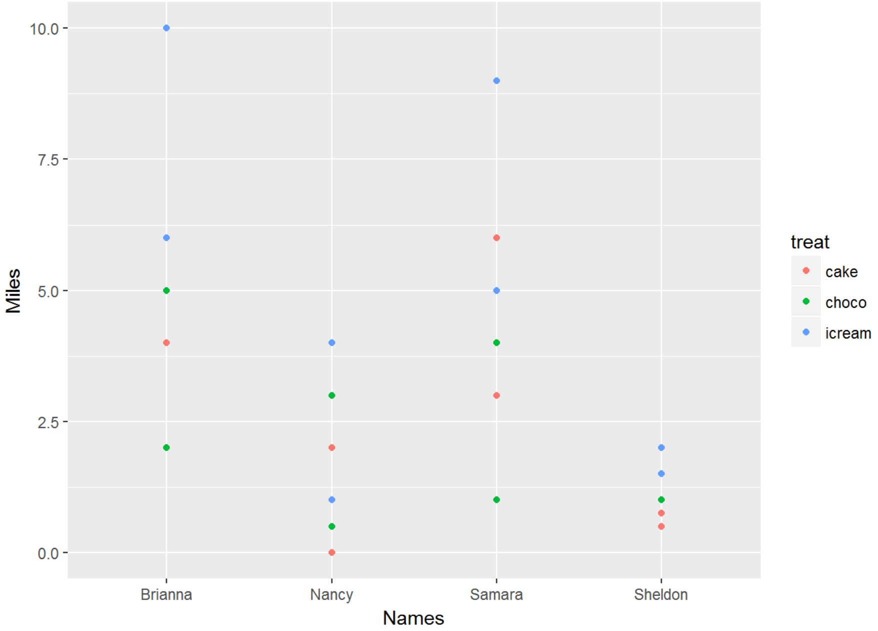
```
treats_data3 <- separate(treats_data2, treat_serving, into = c("treat", "serving"), sep = "_")

treats_data3
```

```
##       name  treat serving miles
## 1   Samara   cake       1  3.00
## 2    Nancy   cake       1  0.00
## 3  Sheldon   cake       1  0.50
## 4  Brianna   cake       1  4.00
## 5   Samara   cake       2  6.00
## 6    Nancy   cake       2  2.00
## 7  Sheldon   cake       2  0.75
## 8  Brianna   cake       2  6.00
## 9   Samara  choco       1  1.00
## 10   Nancy  choco       1  0.50
## 11 Sheldon  choco       1  1.00
## 12 Brianna  choco       1  2.00
## 13  Samara  choco       2  4.00
## 14   Nancy  choco       2  3.00
## 15 Sheldon  choco       2  2.00
## 16 Brianna  choco       2  5.00
## 17  Samara icream       1  5.00
## 18   Nancy icream       1  1.00
## 19 Sheldon icream       1  1.50
## 20 Brianna icream       1  6.00
## 21  Samara icream       2  9.00
## 22   Nancy icream       2  4.00
## 23 Sheldon icream       2  2.00
## 24 Brianna icream       2 10.00
```

We finally got the data frame that we wanted!

Now, that we've created a tidy data frame, we can easily generate a scatterplot using ggplot2 that will show us the distribution of the miles run by each indvidual after having eaten a particular treat.

```
# Load the package ggplot2
library(ggplot2)

# Scatterplot showing the miles run by each individual

ggplot(treats_data3, aes(name, miles)) + geom_point(aes(col = treat)) + labs(x = "Names", y = "Miles")
```



Data Frames: from Wide to Long and Long to Wide

In addition to switching from what could be called an untidy to a tidy format, we can also interpret what we just did as having gone from a "wide" to a "long" format. A data frame that has a wide format will have more than 1 measurement in each row. In the original data frame, treats_data, every row was made up of only measurments. In other words, every column was a measurment variable. With a long data frame, on the other hand, you have only one measurment in each row, whereas the rest of the columns represent categorical variables. Our last data frame, treats_data3, can be classified as a long data frame since the only measurement in each row is the number of miles run. Don't get confused by the "serving" column. Even though this column is made up of numbers, it still qualifies as a categorical variable. In her article on data wrangling, Sharon Machlis provides a good way to think about it. She essentially says that if it doesn't make sense to plot the values of a column on their own, then that column is more than likely not a measurement column. In the case of treats_data3, it wouldn't make sense to plot the repeating 1's and 2's of the "serving" column; it would, however, make sense to plot the number of miles run, especially if the plots were split up into miles run for 1 serving and miles run for 2 servings.

Moving forward, I would like to demonstrate how to go from a "wide" data frame to a "long" one using the package "reshape2" and the function melt().

```
# Let's start by importing the data we will use for our data frame

weight_loss <- read.csv("../Post1/weight_loss_data.csv", stringsAsFactors = FALSE)
```

```
## Warning in read.table(file = file, header = header, sep = sep, quote =
## quote, : incomplete final line found by readTableHeader on '../Post1/
## weight_loss_data.csv'
```

```
weight_loss
```

```
##      name start_year             gym starting_weight end_weight
## 1   Nancy       2015             UFC             180        155
## 2 Sheldon       2014         24_HrFit            220        185
## 3  Samara       2016  Planet_Fitness             170        135
## 4 Brianna       2013         24_HrFit            180        150
##   goal_weight
## 1         150
## 2         180
## 3         135
## 4         140
```

From the data frame weight_loss, we can clearly see that we have more than 1 measurement column; in this case, there are 3. This is where the function melt() will come in! After installing the package "reshape2," I will load it into my Rmd file.

```
# Load "reshape2" into Rmd file

library(reshape2)
```

```
## Warning: package 'reshape2' was built under R version 3.4.2
```

```
##
## Attaching package: 'reshape2'
```

```
## The following object is masked from 'package:tidyr':
##
##     smiths
```

In the code below, I will demonstrate how to use the melt() function in order to get the desired “long” data frame.

```
weight_loss_long <- melt(weight_loss, id.vars = c("name", "start_year", "gym"), variable.name = "weight_info", val
ue.name = "pounds")

weight_loss_long
```

```
##       name start_year             gym     weight_info pounds
## 1    Nancy       2015             UFC starting_weight    180
## 2  Sheldon       2014        24_HrFit starting_weight    220
## 3   Samara       2016  Planet_Fitness starting_weight    170
## 4  Brianna       2013        24_HrFit starting_weight    180
## 5    Nancy       2015             UFC      end_weight    155
## 6  Sheldon       2014        24_HrFit      end_weight    185
## 7   Samara       2016  Planet_Fitness      end_weight    135
## 8  Brianna       2013        24_HrFit      end_weight    150
## 9    Nancy       2015             UFC     goal_weight    150
## 10 Sheldon       2014        24_HrFit     goal_weight    180
## 11  Samara       2016  Planet_Fitness     goal_weight    135
## 12 Brianna       2013        24_HrFit     goal_weight    140
```

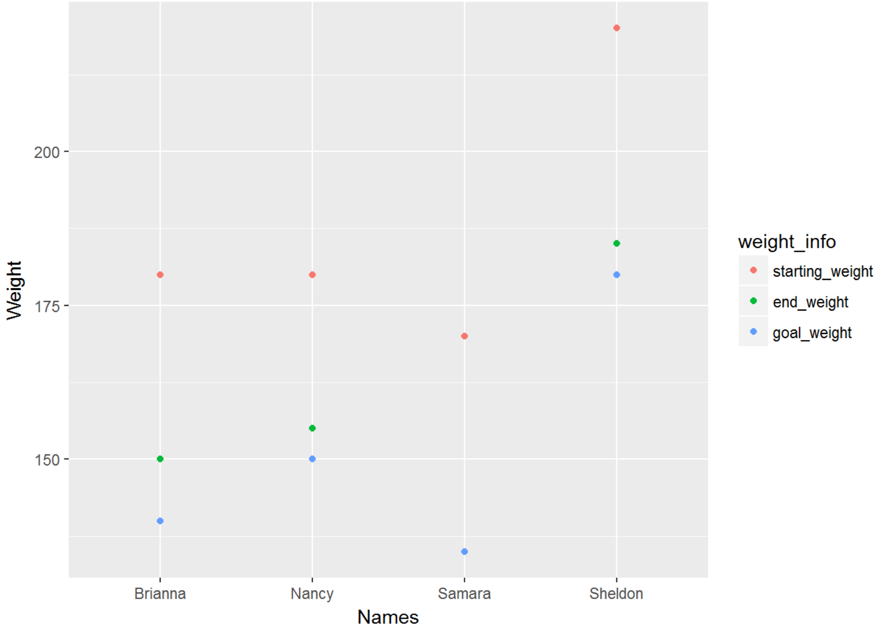
Awesome! We now have only one measurement column, namely “pounds”, as opposed to three! From the code above, we can see that we did not need to explicitly write out the column names that we intended to become the values of our new column “weight_info”. The melt() function will automatically convert the column names that are not specified in the vector that is set equal to id.vars into the input for the new categorical column. The parameter variable.name allows us to name the new categorical column, while the parameter value.name allows us to do the same for the new measurement column.

```
# If you feel more comfortable explicitly writing out which column names you wish to become the input for the new
categorical column "weight_info", you can use the following code, which includes the parameter measure.vars

weight_loss_long <- melt(weight_loss, id.vars = c("name", "start_year", "gym"), measure.vars = c("starting_weight"
, "end_weight", "goal_weight"), variable.name = "weight_info", value.name = "pounds")

weight_loss_long
```

```
##       name start_year             gym     weight_info pounds
## 1    Nancy       2015             UFC starting_weight    180
## 2  Sheldon       2014        24_HrFit starting_weight    220
## 3   Samara       2016  Planet_Fitness starting_weight    170
## 4  Brianna       2013        24_HrFit starting_weight    180
## 5    Nancy       2015             UFC      end_weight    155
## 6  Sheldon       2014        24_HrFit      end_weight    185
## 7   Samara       2016  Planet_Fitness      end_weight    135
## 8  Brianna       2013        24_HrFit      end_weight    150
## 9    Nancy       2015             UFC     goal_weight    150
## 10 Sheldon       2014        24_HrFit     goal_weight    180
## 11  Samara       2016  Planet_Fitness     goal_weight    135
## 12 Brianna       2013        24_HrFit     goal_weight    140
```

Now that our data frame is in a “long” format, let’s create another scatterplot like the one we created for treats_data3!

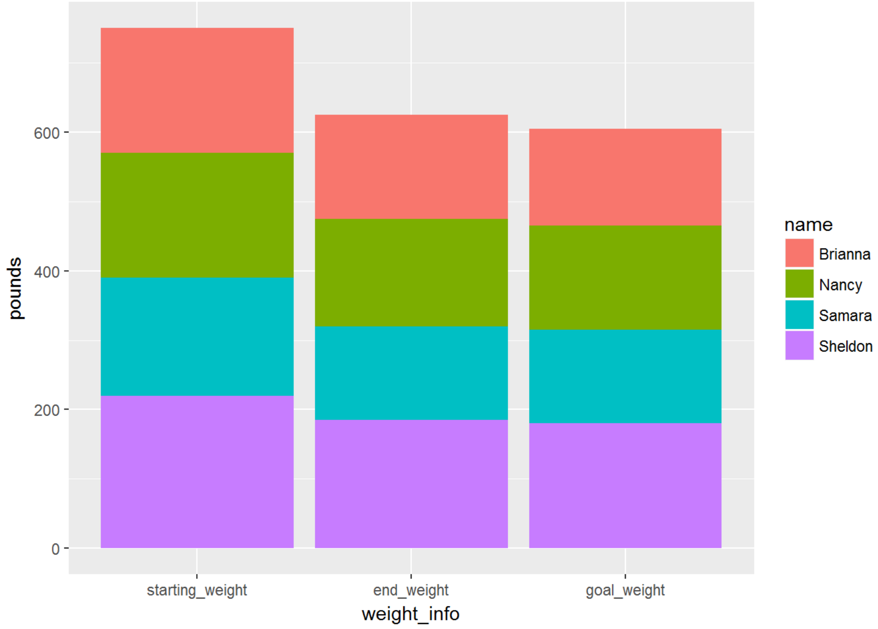
```
# Scatterplot showing the different weights of each individual

ggplot(weight_loss_long, aes(name, pounds)) + geom_point(aes(col = weight_info)) + labs(x = "Names", y = "Weight")
```



The “long” format of weight_loss_long allows us to generate a scatterplot that clearly shows how far each individual was from their target weight. It appears that Samara was the only one to hit her goal weight. We do not see a green dot for her, because her goal weight and end weight overlap.

We could also use weight_loss_long to produce a bar chart that compares the total number of pounds for starting_weight, end_weight, and goal_weight.

```
# Bar chart comparing the total in pounds for the different categories within weight_info

ggplot(weight_loss_long,aes(x = weight_info, y = pounds, fill = name)) + geom_bar(stat = 'identity')
```



Now let’s try going from a long data frame to a wide one!

The function we’ll be using this time is dcast().

```
# This code will give us the graph we started with

weight_loss_wide <- dcast(weight_loss_long, name + start_year + gym ~ weight_info, value.var = "pounds")

weight_loss_wide
```

```
##      name start_year             gym starting_weight end_weight
## 1 Brianna       2013        24_HrFit             180        150
## 2   Nancy       2015             UFC             180        155
## 3  Samara       2016  Planet_Fitness             170        135
## 4 Sheldon       2014        24_HrFit             220        185
##   goal_weight
## 1         140
## 2         150
## 3         135
## 4         180
```

The first argument of the function is the data frame you will be changing. The column names written before the ~ (and separated by a "+") are the columns you'd like to keep, while the column name placed after the ~ indicates the column whose unique values will be used to create new columns. The values for these new columns, in our case, will come from the column "pounds" (the measurement column), hence the assignment of this variable name to the parameter value.var.

Here one can see that what we did with the gather() function from the package "tidyr" was essentially the same as what we did with the melt() function from the package "reshape2"! The same goes for the functions spread() and dcast()! I did not provide an example for spread(), but it essentially does the same thing as dcast().

Family of join() Functions

Now that we've talked about "tidyr" and "reshape2", I'd like to move on to discuss some of the joining functions provided by "dplyr". Merging has been very briefly touched upon in Stat 133, so I'd like to expand upon this.

I'd like to start off by saying that although I will be discussing the join fuctions provided by "dplyr", the same operations can be performed using the function merge(), which is a base R function.

I will be reviewing the following functions:

- inner_join()
- semi_join()
- left_join()
- right_join()
- full_join()

Ok, let's start with inner_join()!

In order to show how inner_join() works, I will be using two data frames from above, weight_loss_long and treats_data3. We will see that inner_join() will return the rows that are in both weight_loss_long and treats_data3 as well as all of the columns in each of those data frames. We know that the "name" column of each data frame contains the exact same names, so it is by the "name" variable that we are joining the two data frames.

```
head(inner_join(weight_loss_long, treats_data3), 26)
```

```
## Joining, by = "name"
##       name start_year             gym     weight_info pounds  treat
## 1    Nancy       2015             UFC starting_weight    180   cake
## 2    Nancy       2015             UFC starting_weight    180   cake
## 3    Nancy       2015             UFC starting_weight    180  choco
## 4    Nancy       2015             UFC starting_weight    180  choco
## 5    Nancy       2015             UFC starting_weight    180 icream
## 6    Nancy       2015             UFC starting_weight    180 icream
## 7  Sheldon       2014        24_HrFit starting_weight    220   cake
## 8  Sheldon       2014        24_HrFit starting_weight    220   cake
## 9  Sheldon       2014        24_HrFit starting_weight    220  choco
## 10 Sheldon       2014        24_HrFit starting_weight    220  choco
## 11 Sheldon       2014        24_HrFit starting_weight    220 icream
## 12 Sheldon       2014        24_HrFit starting_weight    220 icream
## 13  Samara       2016  Planet_Fitness starting_weight    170   cake
## 14  Samara       2016  Planet_Fitness starting_weight    170   cake
## 15  Samara       2016  Planet_Fitness starting_weight    170  choco
## 16  Samara       2016  Planet_Fitness starting_weight    170  choco
## 17  Samara       2016  Planet_Fitness starting_weight    170 icream
## 18  Samara       2016  Planet_Fitness starting_weight    170 icream
## 19 Brianna       2013        24_HrFit starting_weight    180   cake
## 20 Brianna       2013        24_HrFit starting_weight    180   cake
## 21 Brianna       2013        24_HrFit starting_weight    180  choco
## 22 Brianna       2013        24_HrFit starting_weight    180  choco
## 23 Brianna       2013        24_HrFit starting_weight    180 icream
## 24 Brianna       2013        24_HrFit starting_weight    180 icream
## 25   Nancy       2015             UFC      end_weight    155   cake
## 26   Nancy       2015             UFC      end_weight    155   cake
##    serving miles
## 1        1  0.00
## 2        2  2.00
## 3        1  0.50
## 4        2  3.00
## 5        1  1.00
## 6        2  4.00
## 7        1  0.50
## 8        2  0.75
## 9        1  1.00
## 10       2  2.00
## 11       1  1.50
## 12       2  2.00
## 13       1  3.00
## 14       2  6.00
## 15       1  1.00
## 16       2  4.00
## 17       1  5.00
## 18       2  9.00
## 19       1  4.00
## 20       2  6.00
## 21       1  2.00
## 22       2  5.00
## 23       1  6.00
## 24       2 10.00
## 25       1  0.00
## 26       2  2.00
```

In order to avoid getting the message that tells us we’re joining by “name”, we can include the “by” parameter in the inner_join() function.

```
head(inner_join(weight_loss_long, treats_data3, by =  "name"), 26)
```

```
##       name start_year             gym     weight_info pounds  treat
## 1    Nancy       2015             UFC starting_weight    180   cake
## 2    Nancy       2015             UFC starting_weight    180   cake
## 3    Nancy       2015             UFC starting_weight    180  choco
## 4    Nancy       2015             UFC starting_weight    180  choco
## 5    Nancy       2015             UFC starting_weight    180 icream
## 6    Nancy       2015             UFC starting_weight    180 icream
## 7  Sheldon       2014        24_HrFit starting_weight    220   cake
## 8  Sheldon       2014        24_HrFit starting_weight    220   cake
## 9  Sheldon       2014        24_HrFit starting_weight    220  choco
## 10 Sheldon       2014        24_HrFit starting_weight    220  choco
## 11 Sheldon       2014        24_HrFit starting_weight    220 icream
## 12 Sheldon       2014        24_HrFit starting_weight    220 icream
## 13  Samara       2016 Planet_Fitness starting_weight    170   cake
## 14  Samara       2016 Planet_Fitness starting_weight    170   cake
## 15  Samara       2016 Planet_Fitness starting_weight    170  choco
## 16  Samara       2016 Planet_Fitness starting_weight    170  choco
## 17  Samara       2016 Planet_Fitness starting_weight    170 icream
## 18  Samara       2016 Planet_Fitness starting_weight    170 icream
## 19 Brianna       2013        24_HrFit starting_weight    180   cake
## 20 Brianna       2013        24_HrFit starting_weight    180   cake
## 21 Brianna       2013        24_HrFit starting_weight    180  choco
## 22 Brianna       2013        24_HrFit starting_weight    180  choco
## 23 Brianna       2013        24_HrFit starting_weight    180 icream
## 24 Brianna       2013        24_HrFit starting_weight    180 icream
## 25   Nancy       2015             UFC      end_weight    155   cake
## 26   Nancy       2015             UFC      end_weight    155   cake
##    serving miles
## 1        1  0.00
## 2        2  2.00
## 3        1  0.50
## 4        2  3.00
## 5        1  1.00
## 6        2  4.00
## 7        1  0.50
## 8        2  0.75
## 9        1  1.00
## 10       2  2.00
## 11       1  1.50
## 12       2  2.00
## 13       1  3.00
## 14       2  6.00
## 15       1  1.00
## 16       2  4.00
## 17       1  5.00
## 18       2  9.00
## 19       1  4.00
## 20       2  6.00
## 21       1  2.00
## 22       2  5.00
## 23       1  6.00
## 24       2 10.00
## 25       1  0.00
## 26       2  2.00
```

Let’s consider another scenario. Suppose we want to join both data frames but that instead of being called “name” the names column of weight_loss_long was called “NAMES”?

```
# Change the name of the first column in weight_loss_long from "name" to "NAMES"

colnames(weight_loss_long)[1] <- "NAMES"
```

In order to be able to join the two data frames despite the fact that they have no identical column names as they did above, we use the following code:

```
head(inner_join(weight_loss_long, treats_data3, by =  c("NAMES" = "name")), 50)
```

```
##      NAMES start_year             gym     weight_info pounds  treat
## 1    Nancy       2015             UFC starting_weight    180   cake
## 2    Nancy       2015             UFC starting_weight    180   cake
## 3    Nancy       2015             UFC starting_weight    180  choco
## 4    Nancy       2015             UFC starting_weight    180  choco
## 5    Nancy       2015             UFC starting_weight    180 icream
## 6    Nancy       2015             UFC starting_weight    180 icream
## 7  Sheldon       2014        24_HrFit starting_weight    220   cake
## 8  Sheldon       2014        24_HrFit starting_weight    220   cake
## 9  Sheldon       2014        24_HrFit starting_weight    220  choco
## 10 Sheldon       2014        24_HrFit starting_weight    220  choco
## 11 Sheldon       2014        24_HrFit starting_weight    220 icream
## 12 Sheldon       2014        24_HrFit starting_weight    220 icream
## 13  Samara       2016 Planet_Fitness starting_weight    170   cake
## 14  Samara       2016 Planet_Fitness starting_weight    170   cake
## 15  Samara       2016 Planet_Fitness starting_weight    170  choco
## 16  Samara       2016  Planet_Fitness starting_weight    170  choco
## 17  Samara       2016  Planet_Fitness starting_weight    170 icream
## 18  Samara       2016  Planet_Fitness starting_weight    170 icream
## 19 Brianna       2013        24_HrFit starting_weight    180   cake
## 20 Brianna       2013        24_HrFit starting_weight    180   cake
## 21 Brianna       2013        24_HrFit starting_weight    180  choco
## 22 Brianna       2013        24_HrFit starting_weight    180  choco
## 23 Brianna       2013        24_HrFit starting_weight    180 icream
## 24 Brianna       2013        24_HrFit starting_weight    180 icream
## 25   Nancy       2015             UFC      end_weight    155   cake
## 26   Nancy       2015             UFC      end_weight    155   cake
## 27   Nancy       2015             UFC      end_weight    155  choco
## 28   Nancy       2015             UFC      end_weight    155  choco
## 29   Nancy       2015             UFC      end_weight    155 icream
## 30   Nancy       2015             UFC      end_weight    155 icream
## 31 Sheldon       2014        24_HrFit      end_weight    185   cake
## 32 Sheldon       2014        24_HrFit      end_weight    185   cake
## 33 Sheldon       2014        24_HrFit      end_weight    185  choco
## 34 Sheldon       2014        24_HrFit      end_weight    185  choco
## 35 Sheldon       2014        24_HrFit      end_weight    185 icream
## 36 Sheldon       2014        24_HrFit      end_weight    185 icream
## 37  Samara       2016  Planet_Fitness      end_weight    135   cake
## 38  Samara       2016  Planet_Fitness      end_weight    135   cake
## 39  Samara       2016  Planet_Fitness      end_weight    135  choco
## 40  Samara       2016  Planet_Fitness      end_weight    135  choco
## 41  Samara       2016  Planet_Fitness      end_weight    135 icream
## 42  Samara       2016  Planet_Fitness      end_weight    135 icream
## 43 Brianna       2013        24_HrFit      end_weight    150   cake
## 44 Brianna       2013        24_HrFit      end_weight    150   cake
## 45 Brianna       2013        24_HrFit      end_weight    150  choco
## 46 Brianna       2013        24_HrFit      end_weight    150  choco
## 47 Brianna       2013        24_HrFit      end_weight    150 icream
## 48 Brianna       2013        24_HrFit      end_weight    150 icream
## 49   Nancy       2015             UFC     goal_weight    150   cake
## 50   Nancy       2015             UFC     goal_weight    150   cake
##    serving miles
## 1        1  0.00
## 2        2  2.00
## 3        1  0.50
## 4        2  3.00
## 5        1  1.00
## 6        2  4.00
## 7        1  0.50
## 8        2  0.75
## 9        1  1.00
## 10       2  2.00
## 11       1  1.50
## 12       2  2.00
## 13       1  3.00
## 14       2  6.00
## 15       1  1.00
## 16       2  4.00
## 17       1  5.00
## 18       2  9.00
## 19       1  4.00
## 20       2  6.00
## 21       1  2.00
## 22       2  5.00
## 23       1  6.00
## 24       2 10.00
## 25       1  0.00
## 26       2  2.00
## 27       1  0.50
## 28       2  3.00
## 29       1  1.00
## 30       2  4.00
## 31       1  0.50
## 32       2  0.75
## 33       1  1.00
## 34       2  2.00
## 35       1  1.50
## 36       2  2.00
## 37       1  3.00
## 38       2  6.00
## 39       1  1.00
## 40       2  4.00
## 41       1  5.00
## 42       2  9.00
## 43       1  4.00
## 44       2  6.00
## 45       1  2.00
## 46       2  5.00
## 47       1  6.00
## 48       2 10.00
## 49       1  0.00
## 50       2  2.00
## 50        2  2.00
```

If you try to join the two data frames without including the code we set equal to the “by” parameter, you’ll get an error!

I’ll stil be providing one more example using inner_join(), but this is a good place to introduce the semi_join() function.

```
semi_join(weight_loss_long, treats_data3, by =  c("NAMES" = "name"))
```

```
##      NAMES start_year             gym     weight_info pounds
## 1    Nancy       2015             UFC starting_weight    180
## 2  Sheldon       2014        24_HrFit starting_weight    220
## 3   Samara       2016  Planet_Fitness starting_weight    170
## 4  Brianna       2013        24_HrFit starting_weight    180
## 5    Nancy       2015             UFC      end_weight    155
## 6  Sheldon       2014        24_HrFit      end_weight    185
## 7   Samara       2016  Planet_Fitness      end_weight    135
## 8  Brianna       2013        24_HrFit      end_weight    150
## 9    Nancy       2015             UFC     goal_weight    150
## 10 Sheldon       2014        24_HrFit     goal_weight    180
## 11  Samara       2016  Planet_Fitness     goal_weight    135
## 12 Brianna       2013        24_HrFit     goal_weight    140
```

The only difference between semi_join() and inner_join() is that semi_join only includes the columns of the data frame passed in as the first argument.

Just to solidfy our understanding of inner_join(), as mentioned above I’ll provide one more example using this function.

Let’s suppose that the “NAMES” column of weight_loss_long only contains the names “Samara” and “Brianna.”

```
weight_loss_long <- weight_loss_long[-c(1,2,5,6,9,10), ]

weight_loss_long
```

```
##      NAMES start_year             gym     weight_info pounds
## 3   Samara       2016  Planet_Fitness starting_weight    170
## 4  Brianna       2013        24_HrFit starting_weight    180
## 7   Samara       2016  Planet_Fitness      end_weight    135
## 8  Brianna       2013        24_HrFit      end_weight    150
## 11  Samara       2016  Planet_Fitness     goal_weight    135
## 12 Brianna       2013        24_HrFit     goal_weight    140
```

Now we’re going to join weight_loss_long and treats_data3 once more!

```
inner_join(treats_data3, weight_loss_long, by =  c("name" = "NAMES"))
```

```
##       name  treat serving miles start_year            gym     weight_info
## 1   Samara   cake       1     3       2016 Planet_Fitness starting_weight
## 2   Samara   cake       1     3       2016 Planet_Fitness      end_weight
## 3   Samara   cake       1     3       2016 Planet_Fitness     goal_weight
## 4  Brianna   cake       1     4       2013       24_HrFit starting_weight
## 5  Brianna   cake       1     4       2013       24_HrFit      end_weight
## 6  Brianna   cake       1     4       2013       24_HrFit     goal_weight
## 7   Samara   cake       2     6       2016 Planet_Fitness starting_weight
## 8   Samara   cake       2     6       2016 Planet_Fitness      end_weight
## 9   Samara   cake       2     6       2016 Planet_Fitness     goal_weight
## 10 Brianna   cake       2     6       2013       24_HrFit starting_weight
## 11 Brianna   cake       2     6       2013       24_HrFit      end_weight
## 12 Brianna   cake       2     6       2013       24_HrFit     goal_weight
## 13  Samara  choco       1     1       2016 Planet_Fitness starting_weight
## 14  Samara  choco       1     1       2016 Planet_Fitness      end_weight
## 15  Samara  choco       1     1       2016 Planet_Fitness     goal_weight
## 16 Brianna  choco       1     2       2013       24_HrFit starting_weight
## 17 Brianna  choco       1     2       2013       24_HrFit      end_weight
## 18 Brianna  choco       1     2       2013       24_HrFit     goal_weight
## 19  Samara  choco       2     4       2016 Planet_Fitness starting_weight
## 20  Samara  choco       2     4       2016 Planet_Fitness      end_weight
## 21  Samara  choco       2     4       2016 Planet_Fitness     goal_weight
## 22 Brianna  choco       2     5       2013       24_HrFit starting_weight
## 23 Brianna  choco       2     5       2013       24_HrFit      end_weight
## 24 Brianna  choco       2     5       2013       24_HrFit     goal_weight
## 25  Samara icream       1     5       2016 Planet_Fitness starting_weight
## 26  Samara icream       1     5       2016 Planet_Fitness      end_weight
## 27  Samara icream       1     5       2016 Planet_Fitness     goal_weight
## 28 Brianna icream       1     6       2013       24_HrFit starting_weight
## 29 Brianna icream       1     6       2013       24_HrFit      end_weight
## 30 Brianna icream       1     6       2013       24_HrFit     goal_weight
## 31  Samara icream       2     9       2016 Planet_Fitness starting_weight
## 32  Samara icream       2     9       2016 Planet_Fitness      end_weight
## 33  Samara icream       2     9       2016 Planet_Fitness     goal_weight
## 34 Brianna icream       2    10       2013       24_HrFit starting_weight
## 35 Brianna icream       2    10       2013       24_HrFit      end_weight
## 36 Brianna icream       2    10       2013       24_HrFit     goal_weight
##    pounds
## 1     170
## 2     135
## 3     135
## 4     180
## 5     150
## 6     140
## 7     170
## 8     135
## 9     135
## 10    180
## 11    150
## 12    140
## 13    170
## 14    135
## 15    135
## 16    180
## 17    150
## 18    140
## 19    170
## 20    135
## 21    135
## 22    180
## 23    150
## 24    140
## 25    170
## 26    135
## 27    135
## 28    180
## 29    150
## 30    140
## 31    170
## 32    135
## 33    135
## 34    180
## 35    150
## 36    140
```

Great! We see that in the joined data frame we only have rows for “Samara” and “Brianna.”

Let’s also re-use semi_join() after having removed the names “Sheldon” and “Nancy” from weight_loss_long

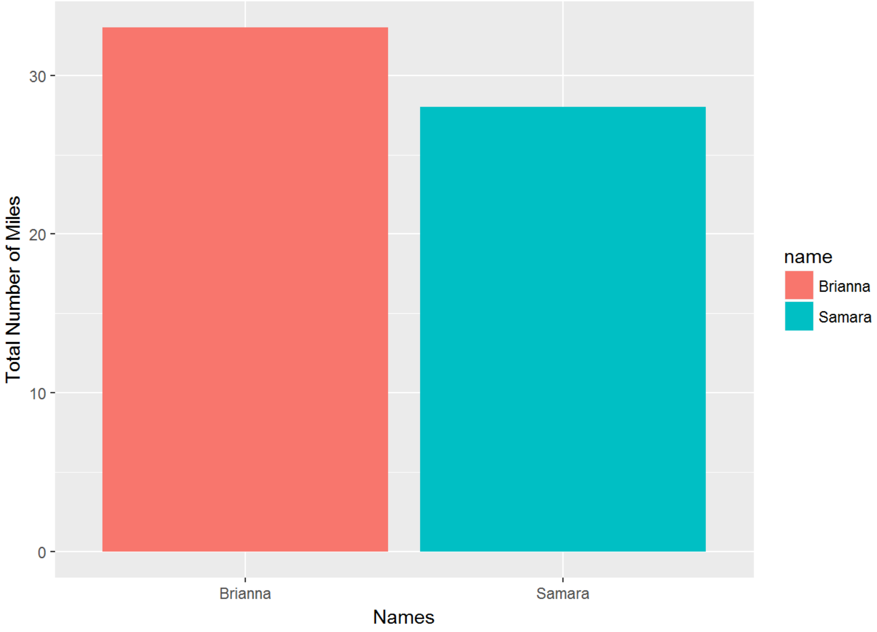
```
semi_j_df <- semi_join(treats_data3, weight_loss_long, by = c("name" = "NAMES"))

semi_j_df
```

```
##        name  treat serving miles
## 1    Samara   cake       1     3
## 2   Brianna   cake       1     4
## 3    Samara   cake       2     6
## 4   Brianna   cake       2     6
## 5    Samara  choco       1     1
## 6   Brianna  choco       1     2
## 7    Samara  choco       2     4
## 8   Brianna  choco       2     5
## 9    Samara icream       1     5
## 10  Brianna icream       1     6
## 11   Samara icream       2     9
## 12  Brianna icream       2    10
```

Here we can see that semi_join() returns a data frame that is almost identical to treats_data3, except for the fact that it has excluded the rows corresponding to the names “Sheldon” and “Nancy” since those names don’t exist in weight_loss_long.

Let’s create another barchart comparing the total number of miles that Samara and Brianna will have run after they have, at some point, consumed every treat_serving combination.

```
# Compare total number of miles run by Brianna and Samara after they have eaten every treat-serving combo

ggplot(semi_j_df, aes(x = name, y = miles, fill = name)) + geom_bar(stat = 'identity') + labs(x = "Names", y= "Tot
al Number of Miles")
```



Brianna clearly ends up running more!

Ok, let’s see how using left_join() gives us a different output than that produced by inner_join()!

```
# Using left_join() with treats_data3 as the x parameter (first argument) and weight_loss_long as the y parameter
(second argument)

left_join1 <- left_join(treats_data3, weight_loss_long, by =  c("name" = "NAMES"))

head(left_join1, 20)
```

```
##       name treat serving miles start_year            gym     weight_info
## 1   Samara  cake       1  3.00       2016 Planet_Fitness starting_weight
## 2   Samara  cake       1  3.00       2016 Planet_Fitness      end_weight
## 3   Samara  cake       1  3.00       2016 Planet_Fitness     goal_weight
## 4    Nancy  cake       1  0.00         NA           <NA>            <NA>
## 5  Sheldon  cake       1  0.50         NA           <NA>            <NA>
## 6  Brianna  cake       1  4.00       2013       24_HrFit starting_weight
## 7  Brianna  cake       1  4.00       2013       24_HrFit      end_weight
## 8  Brianna  cake       1  4.00       2013       24_HrFit     goal_weight
## 9   Samara  cake       2  6.00       2016 Planet_Fitness starting_weight
## 10  Samara  cake       2  6.00       2016 Planet_Fitness      end_weight
## 11  Samara  cake       2  6.00       2016 Planet_Fitness     goal_weight
## 12   Nancy  cake       2  2.00         NA           <NA>            <NA>
## 13 Sheldon  cake       2  0.75         NA           <NA>            <NA>
## 14 Brianna  cake       2  6.00       2013       24_HrFit starting_weight
## 15 Brianna  cake       2  6.00       2013       24_HrFit      end_weight
## 16 Brianna  cake       2  6.00       2013       24_HrFit     goal_weight
## 17  Samara choco       1  1.00       2016 Planet_Fitness starting_weight
## 18  Samara choco       1  1.00       2016 Planet_Fitness      end_weight
## 19  Samara choco       1  1.00       2016 Planet_Fitness     goal_weight
## 20   Nancy choco       1  0.50         NA           <NA>            <NA>
##    pounds
## 1     170
## 2     135
## 3     135
## 4      NA
## 5      NA
## 6     180
## 7     150
## 8     140
## 9     170
## 10    135
## 11    135
## 12     NA
## 13     NA
## 14    180
## 15    150
## 16    140
## 17    170
## 18    135
## 19    135
## 20     NA
```

```
# Using left_join() with weight_loss_long as the x parameter (first argument) and treats_data3 as the y parameter
(second argument)

left_join2 <- left_join(weight_loss_long, treats_data3, by =  c("NAMES" = "name"))

head(left_join2, 24)
```

```
##      NAMES start_year            gym     weight_info pounds  treat
## 1   Samara       2016 Planet_Fitness starting_weight    170   cake
## 2   Samara       2016 Planet_Fitness starting_weight    170   cake
## 3   Samara       2016 Planet_Fitness starting_weight    170  choco
## 4   Samara       2016 Planet_Fitness starting_weight    170  choco
## 5   Samara       2016 Planet_Fitness starting_weight    170 icream
## 6   Samara       2016 Planet_Fitness starting_weight    170 icream
## 7  Brianna       2013       24_HrFit starting_weight    180   cake
## 8  Brianna       2013       24_HrFit starting_weight    180   cake
## 9  Brianna       2013       24_HrFit starting_weight    180  choco
## 10 Brianna       2013       24_HrFit starting_weight    180  choco
## 11 Brianna       2013       24_HrFit starting_weight    180 icream
## 12 Brianna       2013       24_HrFit starting_weight    180 icream
## 13  Samara       2016 Planet_Fitness      end_weight    135   cake
## 14  Samara       2016 Planet_Fitness      end_weight    135   cake
## 15  Samara       2016 Planet_Fitness      end_weight    135  choco
## 16  Samara       2016 Planet_Fitness      end_weight    135  choco
## 17  Samara       2016 Planet_Fitness      end_weight    135 icream
## 18  Samara       2016 Planet_Fitness      end_weight    135 icream
## 19 Brianna       2013       24_HrFit      end_weight    150   cake
## 20 Brianna       2013       24_HrFit      end_weight    150   cake
## 21 Brianna       2013       24_HrFit      end_weight    150  choco
## 22 Brianna       2013       24_HrFit      end_weight    150  choco
## 23 Brianna       2013       24_HrFit      end_weight    150 icream
## 24 Brianna       2013       24_HrFit      end_weight    150 icream
##    serving miles
## 1        1     3
## 2        2     6
## 3        1     1
## 4        2     4
## 5        1     5
## 6        2     9
## 7        1     4
## 8        2     6
## 9        1     2
## 10       2     5
## 11       1     6
## 12       2    10
## 13       1     3
## 14       2     6
## 15       1     1
## 16       2     4
## 17       1     5
## 18       2     9
## 19       1     4
## 20       2     6
## 21       1     2
## 22       2     5
## 23       1     6
## 24       2    10
```

Ok, so there are a few things going on here. To begin with, we see that left_join1 and left_join2 do *not* look the same (contain the exact same contents)! This is because left_join() returns every row from the data frame passed in as the first argument as well as every column from *both* data frames. In left_join1 we we are returning all of the rows from treats_data3 as well as all of the columns from treats_data3 *and* weight_loss_long. This is why we see NA values in our data frame. There are rows in treats_data3 that are not in weight_loss_long, namely those for “Nancy” and “Sheldon”. Therefore, we get NA in the columns from weight_loss_long for both of those names. We do not, on the other hand, get any NA values in left_join2, because for every name in weight_loss_long there is a corresponding name in treats_data3 and thus input available for that name in every column of treats_data3.

Let’s move onto right_join()! The only difference between right_join() and left_join() is that right_join() will return all of the rows from the data frame that is passed in as the second argument of right_join(), as opposed to the first (hence the “right” join).

Therefore, if we want to get a data frame that is identical to left_join1 using the right_join() function, all we have to do is the following:

```
right_join1 <- right_join(weight_loss_long, treats_data3, by =  c("NAMES" = "name"))

head(right_join1, 20)
```

```
##      NAMES start_year            gym     weight_info pounds treat serving
## 1   Samara       2016 Planet_Fitness starting_weight    170  cake       1
## 2   Samara       2016 Planet_Fitness      end_weight    135  cake       1
## 3   Samara       2016 Planet_Fitness     goal_weight    135  cake       1
## 4    Nancy         NA           <NA>            <NA>     NA  cake       1
## 5  Sheldon         NA           <NA>            <NA>     NA  cake       1
## 6  Brianna       2013       24_HrFit starting_weight    180  cake       1
## 7  Brianna       2013       24_HrFit      end_weight    150  cake       1
## 8  Brianna       2013       24_HrFit     goal_weight    140  cake       1
## 9   Samara       2016 Planet_Fitness starting_weight    170  cake       2
## 10  Samara       2016 Planet_Fitness      end_weight    135  cake       2
## 11  Samara       2016 Planet_Fitness     goal_weight    135  cake       2
## 12   Nancy         NA           <NA>            <NA>     NA  cake       2
## 13 Sheldon         NA           <NA>            <NA>     NA  cake       2
## 14 Brianna       2013       24_HrFit starting_weight    180  cake       2
## 15 Brianna       2013       24_HrFit      end_weight    150  cake       2
## 16 Brianna       2013       24_HrFit     goal_weight    140  cake       2
## 17  Samara       2016 Planet_Fitness starting_weight    170 choco       1
## 18  Samara       2016 Planet_Fitness      end_weight    135 choco       1
## 19  Samara       2016 Planet_Fitness     goal_weight    135 choco       1
## 20   Nancy         NA           <NA>            <NA>     NA choco       1
##    miles
## 1   3.00
## 2   3.00
## 3   3.00
## 4   0.00
## 5   0.50
## 6   4.00
## 7   4.00
## 8   4.00
## 9   6.00
## 10  6.00
## 11  6.00
## 12  2.00
## 13  0.75
## 14  6.00
## 15  6.00
## 16  6.00
## 17  1.00
## 18  1.00
## 19  1.00
## 20  0.50
```

Now to briefly dicsuss full_join()! As the name implies, full_join() simply returns a data frame that contains all of the rows and columns from the data frames passed in as the first and second arguments of the function.

```
full_join1 <- full_join(treats_data3, weight_loss_long, by = c("name" = "NAMES"))

head(full_join1, 30)
```

```
##       name treat serving miles start_year            gym     weight_info
## 1   Samara  cake       1  3.00       2016 Planet_Fitness starting_weight
## 2   Samara  cake       1  3.00       2016 Planet_Fitness      end_weight
## 3   Samara  cake       1  3.00       2016 Planet_Fitness     goal_weight
## 4    Nancy  cake       1  0.00         NA           <NA>            <NA>
## 5  Sheldon  cake       1  0.50         NA           <NA>            <NA>
## 6  Brianna  cake       1  4.00       2013       24_HrFit starting_weight
## 7  Brianna  cake       1  4.00       2013       24_HrFit      end_weight
## 8  Brianna  cake       1  4.00       2013       24_HrFit     goal_weight
## 9   Samara  cake       2  6.00       2016 Planet_Fitness starting_weight
## 10  Samara  cake       2  6.00       2016 Planet_Fitness      end_weight
## 11  Samara  cake       2  6.00       2016 Planet_Fitness     goal_weight
## 12   Nancy  cake       2  2.00         NA           <NA>            <NA>
## 13 Sheldon  cake       2  0.75         NA           <NA>            <NA>
## 14 Brianna  cake       2  6.00       2013       24_HrFit starting_weight
## 15 Brianna  cake       2  6.00       2013       24_HrFit      end_weight
## 16 Brianna  cake       2  6.00       2013       24_HrFit     goal_weight
## 17  Samara choco       1  1.00       2016 Planet_Fitness starting_weight
## 18  Samara choco       1  1.00       2016 Planet_Fitness      end_weight
## 19  Samara choco       1  1.00       2016 Planet_Fitness     goal_weight
## 20   Nancy choco       1  0.50         NA           <NA>            <NA>
## 21 Sheldon choco       1  1.00         NA           <NA>            <NA>
## 22 Brianna choco       1  2.00       2013       24_HrFit starting_weight
## 23 Brianna choco       1  2.00       2013       24_HrFit      end_weight
## 24 Brianna choco       1  2.00       2013       24_HrFit     goal_weight
## 25  Samara choco       2  4.00       2016 Planet_Fitness starting_weight
## 26  Samara choco       2  4.00       2016 Planet_Fitness      end_weight
## 27  Samara choco       2  4.00       2016 Planet_Fitness     goal_weight
## 28   Nancy choco       2  3.00         NA           <NA>            <NA>
## 29 Sheldon choco       2  2.00         NA           <NA>            <NA>
## 30 Brianna choco       2  5.00       2013       24_HrFit starting_weight
##    pounds
## 1     170
## 2     135
## 3     135
## 4      NA
## 5      NA
## 6     180
## 7     150
## 8     140
## 9     170
## 10    135
## 11    135
## 12     NA
## 13     NA
## 14    180
## 15    150
## 16    140
## 17    170
## 18    135
## 19    135
## 20     NA
## 21     NA
## 22    180
## 23    150
## 24    140
## 25    170
## 26    135
## 27    135
## 28     NA
## 29     NA
## 30    180
```

In our case, both full_join1 and left_join1 are the same!

Removing NA Values

Let’s suppose that we want to eliminate the rows containing the NA values in full_join1.

In our case, we know that doing so will output a data frame with the same contents as left_join2 (where we have data for “Samara” and “Brianna”–but *do not* have data for “Nancy” and “Sheldon”–for all of the columns in both treats_data3 and weight_loss_long). For the sake of demonstrating the use of complete.cases() and na.omit(), however, I’ll go ahead and use full_join1.

I’ll start off by saying that na.omit() is significantly easier to use, as you only have to pass a data frame into the function in order to remove the rows containing NA values. Because na.omit() is pretty straightforward, I’ll start off by showing how to use complete.cases()

When using complete.cases() we will be selecting the columns that contain NA values.

```
# Removing rows containing NA values from full_join1 using complete.cases()

head(full_join1[complete.cases(full_join1[,5:8]), ], 25)
```

```
##       name  treat serving miles start_year            gym     weight_info
## 1   Samara   cake       1     3       2016  Planet_Fitness starting_weight
## 2   Samara   cake       1     3       2016  Planet_Fitness      end_weight
## 3   Samara   cake       1     3       2016  Planet_Fitness     goal_weight
## 6  Brianna   cake       1     4       2013        24_HrFit starting_weight
## 7  Brianna   cake       1     4       2013        24_HrFit      end_weight
## 8  Brianna   cake       1     4       2013        24_HrFit     goal_weight
## 9   Samara   cake       2     6       2016  Planet_Fitness starting_weight
## 10  Samara   cake       2     6       2016  Planet_Fitness      end_weight
## 11  Samara   cake       2     6       2016  Planet_Fitness     goal_weight
## 14 Brianna   cake       2     6       2013        24_HrFit starting_weight
## 15 Brianna   cake       2     6       2013        24_HrFit      end_weight
## 16 Brianna   cake       2     6       2013        24_HrFit     goal_weight
## 17  Samara  choco       1     1       2016  Planet_Fitness starting_weight
## 18  Samara  choco       1     1       2016  Planet_Fitness      end_weight
## 19  Samara  choco       1     1       2016  Planet_Fitness     goal_weight
## 22 Brianna  choco       1     2       2013        24_HrFit starting_weight
## 23 Brianna  choco       1     2       2013        24_HrFit      end_weight
## 24 Brianna  choco       1     2       2013        24_HrFit     goal_weight
## 25  Samara  choco       2     4       2016  Planet_Fitness starting_weight
## 26  Samara  choco       2     4       2016  Planet_Fitness      end_weight
## 27  Samara  choco       2     4       2016  Planet_Fitness     goal_weight
## 30 Brianna  choco       2     5       2013        24_HrFit starting_weight
## 31 Brianna  choco       2     5       2013        24_HrFit      end_weight
## 32 Brianna  choco       2     5       2013        24_HrFit     goal_weight
## 33  Samara icream       1     5       2016  Planet_Fitness starting_weight
##    pounds
## 1     170
## 2     135
## 3     135
## 6     180
## 7     150
## 8     140
## 9     170
## 10    135
## 11    135
## 14    180
## 15    150
## 16    140
## 17    170
## 18    135
## 19    135
## 22    180
## 23    150
## 24    140
## 25    170
## 26    135
## 27    135
## 30    180
## 31    150
## 32    140
## 33    170
```

Awesome! No more rows with NA values.

Ok, so what exactly is this code doing? To begin with, we see that we subset full_join1 in order to get all of the rows for columns 5 through 8. We only want columns 5:8, because these are the columns that contain NA values. So we see that the data frame being passed into complete.cases() is the following:

```
full_join1[,5:8]
```

```
##    start_year             gym     weight_info pounds
## 1        2016  Planet_Fitness starting_weight    170
## 2        2016  Planet_Fitness      end_weight    135
## 3        2016  Planet_Fitness     goal_weight    135
## 4          NA            <NA>            <NA>     NA
## 5          NA            <NA>            <NA>     NA
## 6        2013        24_HrFit starting_weight    180
## 7        2013        24_HrFit      end_weight    150
## 8        2013        24_HrFit     goal_weight    140
## 9        2016  Planet_Fitness starting_weight    170
## 10       2016  Planet_Fitness      end_weight    135
## 11       2016  Planet_Fitness     goal_weight    135
## 12         NA            <NA>            <NA>     NA
## 13         NA            <NA>            <NA>     NA
## 14       2013        24_HrFit starting_weight    180
## 15       2013        24_HrFit      end_weight    150
## 16       2013        24_HrFit     goal_weight    140
## 17       2016  Planet_Fitness starting_weight    170
## 18       2016  Planet_Fitness      end_weight    135
## 19       2016  Planet_Fitness     goal_weight    135
## 20         NA            <NA>            <NA>     NA
## 21         NA            <NA>            <NA>     NA
## 22       2013        24_HrFit starting_weight    180
## 23       2013        24_HrFit      end_weight    150
## 24       2013        24_HrFit     goal_weight    140
## 25       2016  Planet_Fitness starting_weight    170
## 26       2016  Planet_Fitness      end_weight    135
## 27       2016  Planet_Fitness     goal_weight    135
## 28         NA            <NA>            <NA>     NA
## 29         NA            <NA>            <NA>     NA
## 30       2013        24_HrFit starting_weight    180
## 31       2013        24_HrFit      end_weight    150
## 32       2013        24_HrFit     goal_weight    140
## 33       2016  Planet_Fitness starting_weight    170
## 34       2016  Planet_Fitness      end_weight    135
## 35       2016  Planet_Fitness     goal_weight    135
## 36         NA            <NA>            <NA>     NA
## 37         NA            <NA>            <NA>     NA
## 38       2013        24_HrFit starting_weight    180
## 39       2013        24_HrFit      end_weight    150
## 40       2013        24_HrFit     goal_weight    140
## 41       2016  Planet_Fitness starting_weight    170
## 42       2016  Planet_Fitness      end_weight    135
## 43       2016  Planet_Fitness     goal_weight    135
## 44         NA            <NA>            <NA>     NA
## 45         NA            <NA>            <NA>     NA
## 46       2013        24_HrFit starting_weight    180
## 47       2013        24_HrFit      end_weight    150
## 48       2013        24_HrFit     goal_weight    140
```

What complete.cases() does is take this data frame and output a logical vector specifying which rows have NA values for the given columns. FALSE indicates that the row has NA values. Let's see what this logical vector looks like.

```r
# Logical vector output by complete.cases()

complete_cases <- complete.cases(full_join1[,5:8])

complete_cases
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
## [23]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [34]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
## [45] FALSE  TRUE  TRUE  TRUE
```

Here we can see that the indices of the FALSE values in the vector complete_cases correspond to the rows that contain NA values in full_join1. For example, we see that rows 4 and 5 of full_join1 contain NA values and that the 4th and 5th indices of complete_cases are FALSE values. This is why when we use complete_cases to subset full_join1 we get only the rows without any NA values.

One can also use na.omit() to get the same output.

```r
head(na.omit(full_join1), 25)
```

```
##       name  treat serving miles start_year            gym     weight_info
## 1   Samara   cake       1     3       2016 Planet_Fitness starting_weight
## 2   Samara   cake       1     3       2016 Planet_Fitness      end_weight
## 3   Samara   cake       1     3       2016 Planet_Fitness     goal_weight
## 6  Brianna   cake       1     4       2013       24_HrFit starting_weight
## 7  Brianna   cake       1     4       2013       24_HrFit      end_weight
## 8  Brianna   cake       1     4       2013       24_HrFit     goal_weight
## 9   Samara   cake       2     6       2016 Planet_Fitness starting_weight
## 10  Samara   cake       2     6       2016 Planet_Fitness      end_weight
## 11  Samara   cake       2     6       2016 Planet_Fitness     goal_weight
## 14 Brianna   cake       2     6       2013       24_HrFit starting_weight
## 15 Brianna   cake       2     6       2013       24_HrFit      end_weight
## 16 Brianna   cake       2     6       2013       24_HrFit     goal_weight
## 17  Samara  choco       1     1       2016 Planet_Fitness starting_weight
## 18  Samara  choco       1     1       2016 Planet_Fitness      end_weight
## 19  Samara  choco       1     1       2016 Planet_Fitness     goal_weight
## 22 Brianna  choco       1     2       2013       24_HrFit starting_weight
## 23 Brianna  choco       1     2       2013       24_HrFit      end_weight
## 24 Brianna  choco       1     2       2013       24_HrFit     goal_weight
## 25  Samara  choco       2     4       2016 Planet_Fitness starting_weight
## 26  Samara  choco       2     4       2016 Planet_Fitness      end_weight
## 27  Samara  choco       2     4       2016 Planet_Fitness     goal_weight
## 30 Brianna  choco       2     5       2013       24_HrFit starting_weight
## 31 Brianna  choco       2     5       2013       24_HrFit      end_weight
## 32 Brianna  choco       2     5       2013       24_HrFit     goal_weight
## 33  Samara icream       1     5       2016 Planet_Fitness starting_weight
##    pounds
## 1     170
## 2     135
## 3     135
## 6     180
## 7     150
## 8     140
## 9     170
## 10    135
## 11    135
## 14    180
## 15    150
## 16    140
## 17    170
## 18    135
## 19    135
## 22    180
## 23    150
## 24    140
## 25    170
## 26    135
## 27    135
## 30    180
## 31    150
## 32    140
## 33    170
```

This is obviously much simpler, but it's still worth knowing how to use complete.cases()!

Exploring a Few apply() Functions

I'll now be moving on to talk about the family of apply() functions. The apply() function itself has popped up a few times in Stat 133, however, I'd like to discuss this function a bit more and introduce the following two apply() functions:

- mapply()
- lapply()

I'd like to start off with a simple apply() example in order to demonstrate how one can use apply() to perform a function either by row or by column. It is the "MARGIN" parameter of the apply() function that allows for this specification.

```
# Let's generate a randomn data frame

apply_df <- data.frame(replicate(5, sample(1:10, 6, rep = TRUE)))

apply_df
```

```
##   X1 X2 X3 X4 X5
## 1  3  4  1  2  1
## 2  1  8  8  2  9
## 3  9  1  2  3  7
## 4  3  3  8  1 10
## 5  8  2  4  7  6
## 6  1  2  2  7  4
```

```
# Multiplying by row using apply()

apply(apply_df, 1, prod)
```

```
## [1]   24 1152  378  720 2688  112
```

```
# Multiplying by column using apply()

apply(apply_df, 2, prod)
```

```
##    X1    X2    X3    X4    X5
##   648   384  1024   588 15120
```

Great! So we see that if we want to apply a function by *row* we set “MARGIN” equal to **1**, and if we want to apply a function by *column* we set “MARGIN” equal to **2**.

Ok, let’s see how mapply() works. Keep in mind that mapply() allows a function to be applied to several vectors or lists.

```
# Here is one example using mapply()
mapply_df <- as.data.frame(mapply(rep, 2:5, 3))

mapply_df
```

```
##   V1 V2 V3 V4
## 1  2  3  4  5
## 2  2  3  4  5
## 3  2  3  4  5
```

Had we not used mapply(), we would have had to do something like this:

```
# Generating mapply_df without using mapply()

as.data.frame(matrix(c(rep(2, 3), rep(3,3), rep(4,3), rep(5,3)), 3, 4))
```

```
##   V1 V2 V3 V4
## 1  2  3  4  5
## 2  2  3  4  5
## 3  2  3  4  5
```

Clearly less efficient! Ok, let’s go over one more example using mapply().

```
list_1 <- list(1:4, 6:9, 10:13)
list_2 <- list(4:7, 12:15, 10:16)

mapply(prod, list_1, list_2)
```

```
## [1]        20160     99066240 989404416000
```

So here we see that the correpsonding elements of each list are being multiplied by each other. In other words 1:4 from list_1 is being multiplied by 4:7 of list_2. The numbers within each of those vectors are also multiplied by one another before being multiplied by the vector of the other list, so when we multiply 1:4 by 4:7, we are actually multiplying the product of 1, 2, 3, and 4 by the product of 4, 5, 6, and 7.

What if we wanted to use mapply() on only part of each list as opposed to the whole list? We could do the following:

```
# Use mapply() to multiply the second element of both lists by one another

mapply(prod, list_1[2], list_2[2])
```

```
## [1] 99066240
```

It is important to note that the lists must be of the same length if we wish to apply a function to them in the way that we do with mapply(). In other words, if list_2 containted only two elements as opposed to three, trying to run the code `mapply(prod, list_1, list_2)` would **not** work.

Ok, now let’s go over how to use lapply()! This function is very similar to apply(), the main difference being that lapply() outputs a list. I’ll show one very simply example using lapply() and then one that is a bit more involved.

```
# Very simple application of lapply()
lapply_df <- data.frame(replicate(5, sample(1:10, 4, rep = TRUE)))

lapply_df
```

```
##   X1 X2 X3 X4 X5
## 1  8  4  3  4  5
## 2  5  3  4  8  2
## 3  5  8  6  7  1
## 4  6  2  8  3  8
```

```
lapply(lapply_df, median)
```

```
## $X1
## [1] 5.5
##
## $X2
## [1] 3.5
##
## $X3
## [1] 5
##
## $X4
## [1] 5.5
##
## $X5
## [1] 3.5
```

```
# Another example using lapply()

# First genererate three random data frames

df_1 <- data.frame(replicate(5, sample(4:20, 4, rep = TRUE)))

df_2 <- data.frame(replicate(5, sample(8:30, 4, rep = TRUE)))

df_3 <- data.frame(replicate(5, sample(40:50, 4, rep = TRUE)))

# Combine these data frames into a list
df_list <- list(df_1, df_2, df_3)
```

Now we'll actually use lapply()!

```
# Selecting rows 3 and 4 from each data frame of df_list

lapply(df_list,"[",3:4, )
```

```
## [[1]]
##   X1 X2 X3 X4 X5
## 3 12 20  4 15 13
## 4  5 12 12  4 16
##
## [[2]]
##   X1 X2 X3 X4 X5
## 3 14 25 30 12 25
## 4 26 13 12 16 16
##
## [[3]]
##   X1 X2 X3 X4 X5
## 3 41 44 45 50 41
## 4 40 48 46 50 46
```

```
# Selecting columns 3 and 4 from each data frame of df_list

lapply(df_list, "[", ,3:4)
```

```
## [[1]]
##   X3 X4
## 1 20 15
## 2 15 19
## 3  4 15
## 4 12  4
##
## [[2]]
##   X3 X4
## 1 19 17
## 2 30  9
## 3 30 12
## 4 12 16
##
## [[3]]
##   X3 X4
## 1 47 40
## 2 49 44
## 3 45 50
## 4 46 50
```

The output is still a list!

Take-Home Message

This post has gone over several types of data frame manipulation. We began by highlighting the relationship betwen **aggregate()** and the "dplyr" functions **summarise()** and **group_by()**. The aggregate() function was also explored in greater depth than was done in Stat 133. Afterwards, we went over how to make an **"untidy"** data frame **"tidy"** as well how to switch between **"wide"** and **"long"** formats. Lastly, we discussed the various **join()** functions provided by “dplyr” as well as a few of the functions within the **apply()** family. Overall, one can see that data frame manipulation can involve altering the **contents** of a data frame as well as its *structure*.

You got through the entire post!

References

I used the sites lised below when creating this post. I would like to give credit to David Kun, Sharon Machlis, Philippe Marchand, Karlijn Willems, Carlo Fanara, and Neil Sanders for inspiring the examples in this post.

Site 1

Site 2

Site 3

Site 4

Site 5

Site 6

Site 7

Site 8

Site 9