Data Cleaning, Manipulation and Visualization Materials

Brier Gallihugh, M.S.

2024-07-11

Table of contents

Installing The `Tidyverse` Package 1
Working With The `dplyr` Package (Data Manipulation) 2
Recoding Variables 3
Creating Variables 4
Filtering Variables 5
Reverse Coding 5
Working With The `stringr` Package (Working w/ Strings) 7
Text Detection With `stringr` Package 7
Original Output 7
New Output 8
Text Replacement With The `stringr` Package 8
Working With The `lubridate` Package (Date Data) 9
Converting to Date Format 9
Modifying Date Format 9
May-20-2023 Format 9
20-May-2023 Format 10
Working With The `ggplot2` Package 10
Standard Histogram With Density Curve 10
Standard Column Bar Graph 11
Standard Boxplot Graph 14
Standard Violin Plot 15
Standard Line Graph 16
Standard Column Bar Graph W/ Std. Error 19

Installing The `Tidyverse` Package

```
install.packages("tidyverse")
library(tidyverse)
```

The above code is how you want to start any R script. You always want to install and load in any packages that you may need in order to run analyses. For this part of the R workshop we will be working with what is called the `tidyverse` package. It's essentially the go to array of packages in R for data science needs (and therefore a good portion of our needs as well)

A Note On Packages & `library()` Function

You only need to install a package once (unless you update your version of R). The package gets stored on your local computer. A `library()` function call imports the installed package from your local storage. Further, you only need to call the `library()` function once per R script

Working With The `dplyr` Package (Data Manipulation)

```
library(tidyverse)
library(skimr)

dplyr_data <- dplyr::starwars
```

Lines 1-2

Call the tidyverse packages

Line 4

We will be using the starwars data set for the `dplyr` tutorial. I've assigned it to the variable dplyr_data here.

```
skim(dplyr_data)
```

Line 1

We can view some of the key variable data using the `skim()`

Name	dplyr_data
Number of rows	87
Number of columns	14
_______________________	
Column type frequency:	
character	8
list	3
numeric	3
________________________	
Group variables	None

Table 1: Data summary

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
name	0	1.00	3	21	0	87	0
hair_color	5	0.94	4	13	0	11	0
skin_color	0	1.00	3	19	0	31	0
eye_color	0	1.00	3	13	0	15	0
sex	4	0.95	4	14	0	4	0
gender	4	0.95	8	9	0	2	0
homeworld	10	0.89	4	14	0	48	0
species	4	0.95	3	14	0	37	0

Variable type: list

skim_variable	n_missing	complete_rate	n_unique	min_length	max_length
films	0	1	24	1	7
vehicles	0	1	11	0	2
starships	0	1	16	0	5

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
height	6	0.93	174.60	34.77	66	167.0	180	191.0	264	
mass	28	0.68	97.31	169.46	15	55.6	79	84.5	1358	
birth_year	44	0.49	87.57	154.69	8	35.0	52	72.0	896	

```r
head(dplyr_data)
```

Line 1

We can also use the `head()` function which simply gives you a print out of the first 5 rows of a data set.

Recoding Variables

One variable when looking at the `starwars` data set might be sex. Here we can see it is coded as both a character and as either male, female or NA. For a simple recode we might wish to

1. Transform the variables into a factor

2. Change the naming convention to maybe 1, 0 and Unknown

We can achieve this with the code below

```
dplyr_data <- dplyr_data %>%
  mutate(sex = as.factor(sex),
         sex = recode(sex,
                      'male' = '0',
                      'female' = '1'))
```

Line 2

To recode the variable `sex` we need to use the `mutate()` function and `as.factor()` functions as shown above

Lines 3-5

To recode the values for male and female to 0 and 1 respectively, we need to use the `recode()` function as shown here. ::: {.callout-tip} ### A Note On %>% Operator You may have noticed this %>% operator. This is a handy operator that essentially takes the data on the left hand side and "pipe"s it into whatever is on the right as the first argument. This is most effective when the right hand function is expecting some form of a data set :::

> A Note On `dplyr::recode()` Function
>
> The `recode()` function in the `dplyr` package uses what is called `OLD` to `NEW` syntax. This just means that when renaming variables as shown here, you want to list the original variable name followed by you new desired variable name

Creating Variables

Creating variables in R can be done a couple of ways. One is a little clunky (from a code perspective) and the other is more elegant. I'll cover the more clunky way first followed by the more elegant way second. I'll illustrate this by creating a variable that takes the `mass` variable from the `starwars` data set and reduces it by 10 units

```
dplyr_data$mass_10a <- dplyr_data$mass - 10


dplyr_data <- dplyr_data %>%
  mutate(mass_10b = mass - 10)
```

Line 1

This ways is relatively simply because you can think of it as a simple formula notation. However, it's a little clunky because typically adding a $ operator is considered poor coding practice

Lines 4-5

The more elegant way to create a variable is to simply again use the `mutate()` function

A Note On the $ Operator

The $ operator simply says from the data set on the left of the operator, please find (or create) the variable on the right. In this case, from the `dplyr_data` data, create the variable `mass_10a`

Filtering Variables

Keeping with the `starwars` data set, we might wish to revisit our earlier mutate of the male and female `sex` variable categories. Suppose for an analysis we wish to only include the male and female `starwars` characters? For this we might wish to filter so that our data only contains males and females. The code below will illustrate exactly how to do this

```
dplyr_data_mf <- dplyr_data %>%
  filter(sex == 1 | sex == 0)
```

Lines 1-2

Here we have a `filter()` function that takes an argument for which conditions to include [==]. In this case we have when sex = 1 OR [|] when sex = 0.

A Note on Syntax

The `filter()` function uses the notation == to serve as "equals". You may also tell `filter()` what NOT to include with the notation `!=`

Reverse Coding

It is not uncommon for many of you to work with scales that might require some form of reverse coding. This can be accomplished using the following syntax. What is left will be the original dataframe with added columns for the items that we've reverse coded. They will have a "_R" variable name for ease of use

```
library(tidyverse)
library(psych)

df <- data.frame(Q1 = c(1,3,4,5,6,7),
                 Q2 = c(3,4,5,5,7,7),
                 Q3 = c(1,2,2,4,1,1))

reverse_key <- c(1,-1,1)

df_R <- data.frame(reverse.code(keys = reverse_key,
                   items = df[,c("Q1","Q2","Q3")],
                   mini = 1,
                   maxi = 7)) %>%
  rename("Q2_R" = "Q2.")
```

```
df <- right_join(df,df_R,
                keep = FALSE)

print(df)
```

Line 2

The `psych` package contains a `reverse.code()` function for scale items

Lines 4-6

There are no convenient pre-built data sets for this so I've created a quick toy one called `df` with the variables Q1, Q2, and Q3

Line 8

The `reverse.code()` function requires a `keys` argument which is essentially a numerical vector of length of the reverse coded items that correspond sequentially to which items are (`-1`) and aren't (`1`) reverse coded

Line 10

This is the start of the `reverse.code()` function within a new dataframe

Line 11

I've subset (only included) the scale items here using this notation

Line 12

Mini refers to the lowest possible value for the scale (i.e., 1)

Line 13

Maxi refers to the highest possible value for the scale (i.e., 7)

Line 14

I've added a `rename()` function to rename the reverse coded items from "`ItemX.`" to "`ItemX_R`" so we can track which items are the reverse coded one's later

Line 16

Joining the two data frames into the original one so we only have to worry about the original data set

Line 17

Refers to keeping the keys used to join the two data frames (i.e., unique identifiers). We don't want to keep them here

```
  Q1 Q2 Q3 Q2_R
1  1  3  1    5
2  3  4  2    4
3  4  5  2    3
4  5  5  4    3
```

```
5  6  7  1    1
6  7  7  1    1
```

Working With The stringr Package (Working w/ Strings)

The `stringr` package is primarily used when working with what are known as strings of data. Essentially text box types of free response options. For example maybe in a Qualtrics form you allow someone to list “Other” as their religious belief system but ask them upon that selection choice to type out a better word. Same might be true for gender for example. Below we’ll use the `words` data set to some basic text manipulation with the first 10 rows of data. On the right, we will see our original data set. However, on the right we will see that data set ultimately filtered by whether or not there is an `ab` in the `words` variable for a given observation.

Text Detection With stringr Package

```r
library(tidyverse)

stringr_data <- data.frame(stringr::words %>%
                             head(10)) %>%
  rename("words" = "stringr..words.....head.10.")

stringr_data_original <- stringr_data %>%
  mutate(match = str_detect(words,pattern = "ab"))


stringr_data_new <- stringr_data_original %>%
  filter(match == TRUE)
```

Lines 3-4

Here I am converting the stringr data into a data frame and selecting the first 10 observations for simplicity.

Line 5

I’m also using the `rename()` function to change the preset variable name to “words”.

Line 7

I’m “piping” the stringr_data into the `mutate()` function

Line 8

This line shows that I am creating a variable called `match` that will output a `TRUE` or `FALSE` if in the column `words` there is a pattern of `"ab"`.

Lines 11-12

I am filtering the column `match` by whether or not it is `TRUE` (i.e., whether an observation consists of the pattern “ab”)

Original Output

```
      words match
1         a FALSE
2      able  TRUE
3     about  TRUE
4  absolute  TRUE
5    accept FALSE
6   account FALSE
7   achieve FALSE
8    across FALSE
9       act FALSE
10   active FALSE
```

New Output

```
     words match
1     able  TRUE
2    about  TRUE
3 absolute  TRUE
```

Text Replacement With The `stringr` Package

While we've seen how to pull out matching observations using text responses, maybe we want to actually modify the responses. We can do that as well. We will demonstrate using the new data frame consisting of 3 words. Let's as an example replace the pattern "ab" with nothing. We see how to do that below

```r
stringr_data_new <- stringr_data_new %>%
  mutate(across(.cols="words",
                .fns=str_replace,
                pattern = "ab",
                replacement = ""))

print(stringr_data_new)
```

Line 2

Here I am specifing that I wish to apply a function to the `words` column

Line 3

The function I wish to apply is the `str_replace` function which takes two arguments (`pattern` and `replacement` which I'm about to specify)

Line 4

I specify the pattern I'm looking for as `"ab"`

Line 5

I specify what I would like to replace that pattern with. In this case I don't want anything so I just put `""`

```
   words match
1     le  TRUE
2    out  TRUE
3 solute  TRUE
```

Working With The `lubridate` Package (Date Data)

Personally, I don't work with date data very often. Usually time simply isn't a variable I'm interested in. However, for many of you who may be clinical or health focused, this is likely not your experience. Lets see how we can use the `lubridate` package to mess with date formatted data

Converting to Date Format

```
library(tidyverse)

lubridate_data <- lubridate::lakers

lubridate_data <- lubridate_data %>%
  mutate(across(.cols=date,
                .fns=ymd)) %>%
  mutate(date_myd = format(as.Date(date),"%m-%d-%Y"))
```

Lines 6-7

Here I am saying I wish to apply the function `ymd()` to the `date` column

Line 8

For this line, I am saying I wish to create a new variable called `date_myd` by formatting the `date` variable both as a date AND then formatted to a mm-dd-yyyy format. That corresponds to the `"%m-%d-%Y"` string we see on this line.

Modifying Date Format

We can see here that we've converted a numeric value in the format (YYYYMMDD) into a date in the "Year-Month-Date" format. This even looks a little more appealing to the eye especially as you're scanning the date. However, what if you don't like YYYY-MM-DD format and would rather have something like MM-DD-YYYY format instead as is common in the US? Below you can see how to take the format we just used and convert it to the more US common syntax shown on the left. On the right, we can see how to do it for the more EU common syntax of DD-MM-YY

```
lubridate_data <- lubridate_data %>%
  mutate(date_dmy = format(as.Date(date),"%d-%m-%Y"))
```

Lines 1-2

Here I am doing the same as earlier but I am changing the format code to be dd-mm-yyyy using the string `"%d-%m-%Y"`

May-20-2023 Format

```
[1] "10-28-2008"
```

20-May-2023 Format

```
[1] "28-10-2008"
```

Working With The `ggplot2` Package

Standard Histogram With Density Curve

```
library(tidyverse)
library(jtools)

gender <- rep(c("male","female"),50)
test <- rnorm(100,mean = 75,sd=2)

df <- data.frame(gender,test)

density_plot <- ggplot(df,aes(x = test)) +
  geom_histogram(aes(y=after_stat(density)),binwidth = 1) +
  stat_function(fun = dnorm,
                args = list(mean = mean(test),
                            sd = sd(test)),
                col = "blue",
                linewidth = 1) +
  jtools::theme_apa() +
  labs(title = "Figure 1. Histogram of Test Scores",
       x = "Test Scores",
       y = "Score Density")

ggsave("histogram.png")
print(density_plot)
```

Lines 4-5,7

Creation of a basic data set consisting of 100 observations of 2 variables (gender and test)

Line 9

Initial ggplot2 taking the arguments for df as the data and test as our variable to create a histogram of

Line 10

The `geom_histogram()` tells `ggplot2` what type of geom to draw using the `aes()` data above. The `aes(y=after_stat(density))` tells ggplot to convert the y axis as a function of density (vs count which is the default)

Line 11

This `stats_function` allows us to graph a statistic onto the graph. In this case we want it to graph a normal distribution (the `dnorm` function) of the variable we care about.

Lines 12-13

The `stats_function` takes an `args()` function that we have to give it the *mean* and *sd* of the variable we care about. This is shown here

Lines 14-15

These provide some general aesthetic choices so we've specified the curve to be colored blue with a relatively small line width of 1.

Line 16

The `theme_apa()` function simply modifies the `ggplot2` graph to roughly align with APA formatting

Lines 17-19

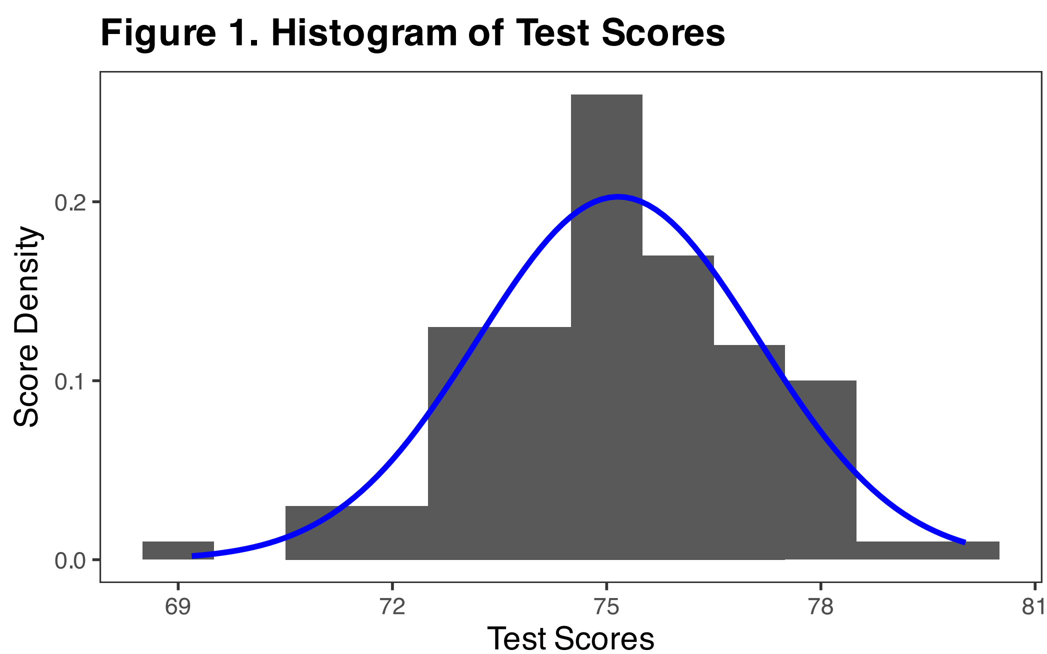
The `labs()` function allows us to add labels to our prospective histogram

Line 21

This will save the built graphic as a .png file

Line 22

This will print the `ggplot2` column plot



Standard Column Bar Graph

```
library(tidyverse)
library(jtools)
```

```r
col_data <- mtcars

skimr::skim(col_data)
```

Lines 4,6

The mtcars data set comes with the `ggplot2` package. Finally I used the `skim()` function to take a quick look at the data

Name	col_data
Number of rows	32
Number of columns	11
_______________________	
Column type frequency:	
numeric	11
________________________	
Group variables	None

Table 5: Data summary

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
mpg	0	1	20.09	6.03	10.40	15.43	19.20	22.80	33.90	
cyl	0	1	6.19	1.79	4.00	4.00	6.00	8.00	8.00	
disp	0	1	230.72	123.94	71.10	120.83	196.30	326.00	472.00	
hp	0	1	146.69	68.56	52.00	96.50	123.00	180.00	335.00	
drat	0	1	3.60	0.53	2.76	3.08	3.70	3.92	4.93	
wt	0	1	3.22	0.98	1.51	2.58	3.33	3.61	5.42	
qsec	0	1	17.85	1.79	14.50	16.89	17.71	18.90	22.90	
vs	0	1	0.44	0.50	0.00	0.00	0.00	1.00	1.00	
am	0	1	0.41	0.50	0.00	0.00	0.00	1.00	1.00	
gear	0	1	3.69	0.74	3.00	3.00	4.00	4.00	5.00	
carb	0	1	2.81	1.62	1.00	2.00	2.00	4.00	8.00	

```r
col_data <- col_data %>%
  group_by(cyl) %>%
  summarize(n = n(),
            mpg_average = mean(mpg))

col_plot <- ggplot(col_data,aes(x = as.factor(cyl),
                                y = mpg_average,
                                fill = as.factor(cyl))) +
  geom_col(color = "black") +
  labs(x = "Number of Cylinders",
       y = "Average Fuel Economy (mpg)",
       title = "Figure 2. Average Fuel Economy by Cylinder Count",
       caption = "Source: Data from the mtcars data set") +
  jtools::theme_apa() +
  theme(plot.caption = element_text(hjust = 0)) +
  scale_fill_manual(values = c("grey50","grey80","grey100"))

ggsave("col_plot.png")
print(col_plot)
```

Lines 1-4

I want to modify my data so that I have it grouped by `cyl` and `n()` and `average_mpg` are calculated

Lines 6-8

I'm starting to layer my column plot with this. `aes()` is where you put your important data (e.g., x and y variables)

Line 9

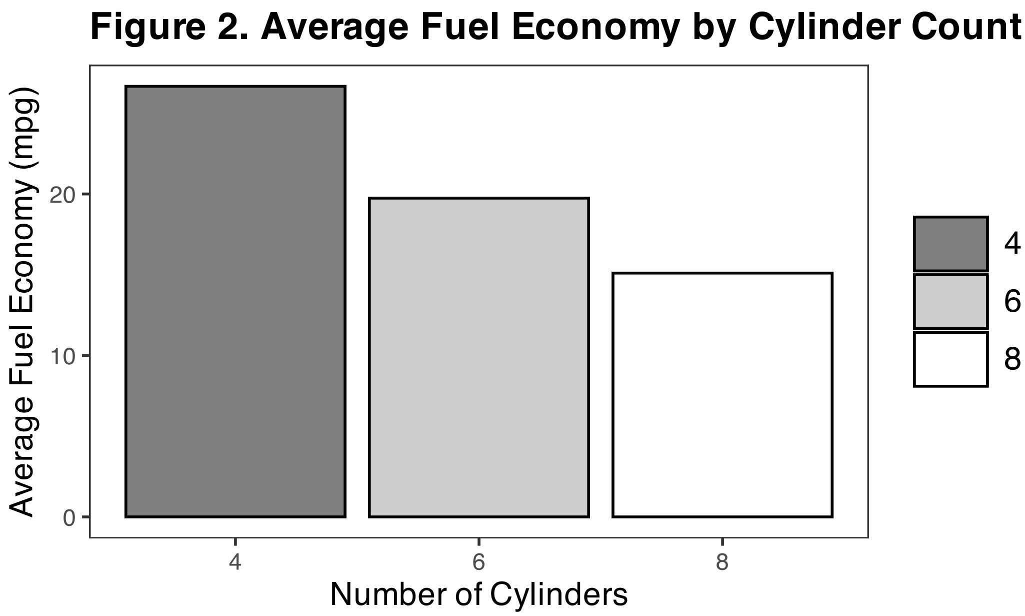
The `geom_col()` tells `ggplot2` what type of geom to draw using the `aes()` data above

Line 15

The `theme(plot.caption = element_text(hjust = 0))` just left justifies the caption

Line 16

The `scale_fill_manual()` tells `ggplot2` what to assign for the `fill` variable in the `aes()` function



Source: Data from the mtcars data set

Standard Boxplot Graph

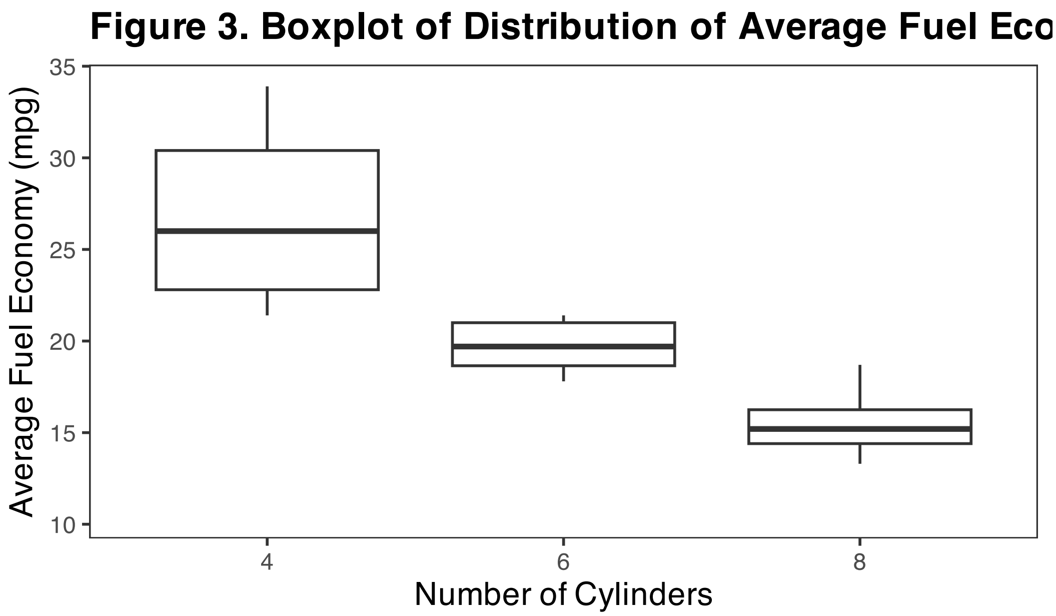
```
library(tidyverse)
library(jtools)
bplot_data <- mtcars

box_plot <- ggplot(bplot_data,aes(x = as.factor(cyl),
                                  y = mpg)) +
  geom_boxplot(outlier.shape = NA) +
  labs(x = "Number of Cylinders",
       y = "Average Fuel Economy (mpg)",
       title = "Figure 3. Boxplot of Distribution of Average Fuel Economy by
Cylinder Count",
       caption = "Source: Data from the mtcars data set") +
  jtools::theme_apa() +
  theme(plot.caption = element_text(hjust = 0)) +
  scale_fill_manual(values = c("grey50","grey80","grey100"))

ggsave("box_plot.png")
print(box_plot)
```

Line 7

The beauty of `ggplot2` is that there is a lot of overlap between different geom. The data to make a column chart vs a box plot in `ggplot2` is just the `geom_boxplot` vs `geom_col` function calls shown here



Source: Data from the mtcars data set

Standard Violin Plot

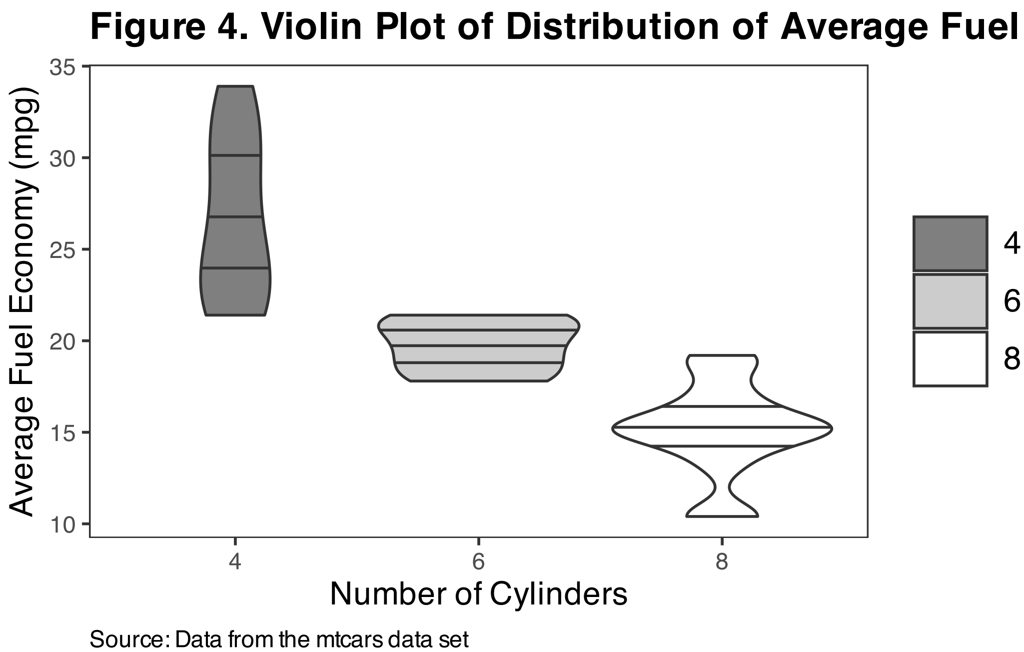
```
library(tidyverse)
library(jtools)
violin_data <- mtcars

violin_plot <- ggplot(violin_data,aes(x = as.factor(cyl),
                                      y = mpg,
                                      fill = as.factor(cyl))) +
  geom_violin(draw_quantiles = c(.25,.50,.75)) +
  labs(x = "Number of Cylinders",
       y = "Average Fuel Economy (mpg)",
       title = "Figure 4. Violin Plot of Distribution of Average Fuel Economy
by Cylinder Count",
       caption = "Source: Data from the mtcars data set") +
  jtools::theme_apa() +
  theme(plot.caption = element_text(hjust = 0)) +
  scale_fill_manual(values = c("grey50","grey80","grey100"))

ggsave("violin.png")
violin_plot
```

Line 8

The `draw_quartiles` function takes a numeric list to represent the quartiles you want. I've chosen the most common of 25%, 50% and 75% but you can input any set of 3 values you'd like



Standard Line Graph

```
library(tidyverse)
library(jtools)
library(skimr)

line_data <- txhousing
skimr::skim(line_data)
```

Lines 5-6

We're now using a Texas housing data set found the `ggplot2` package. We can take a look at it by using the `skim()` function in the `skimr` package

Name	line_data
Number of rows	8602
Number of columns	9
_______________________	
Column type frequency:	
character	1
numeric	8
________________________	
Group variables	None

Table 7: Data summary

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
city	0	1	4	21	0	46	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
year	0	1.00	2007.30	4.50	2000	2003.00	2007.00	2011.00	2015.0	
month	0	1.00	6.41	3.44	1	3.00	6.00	9.00	12.0	
sales	568	0.93	549.56	1110.74	6	86.00	169.00	467.00	8945.0	
volume	568	0.93	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
median	616	0.93	128131.44	37359.58	50000	100000.00	123800.00	150000.00	304200.0	
listings	1424	0.83	3216.90	5968.33	0	682.00	1283.00	2953.75	43107.0	
inventory	1467	0.83	7.17	4.61	0	4.90	6.20	8.15	55.9	
date	0	1.00	2007.75	4.50	2000	2003.83	2007.75	2011.67	2015.5	

```
line_data <- line_data %>%
  group_by(year) %>%
  summarize(total_sales = sum(sales, na.rm = TRUE))

line_plot <- ggplot(line_data,aes(x = year,
                                  y = total_sales)) +
```

```r
  geom_point() +
  geom_line() +
  labs(x = "Year",
       y = "Total Housing Sales",
       title = "Figure 5. Total Texas Housing Sales By Year",
       caption = "Source: Data from the ggplot2 data set") +
  scale_x_continuous(breaks = seq(2000,2015,2)) +
  jtools::theme_apa() +
  theme(plot.caption = element_text(hjust = 0))

ggsave("line.png")
print(line_plot)
```

Lines 1-3

It might be useful to see how sales have changed over time within Texas. As such we might want to summarize the total number of home sales by year. How to do this is illustrated here with a `group_by()` and `summarize()` function.

Lines 5-6

We need to feed the `ggplot` object our `aes()` variables. For this we've selected `year` and `total_sales` as our x and y variable respectively

Line 7

We might want to add points to our line graph for readability so we can add a `geom_point()` layer

Line 8

Now we want to add our actual lines. We can do that by providing a `geom_line()` layer

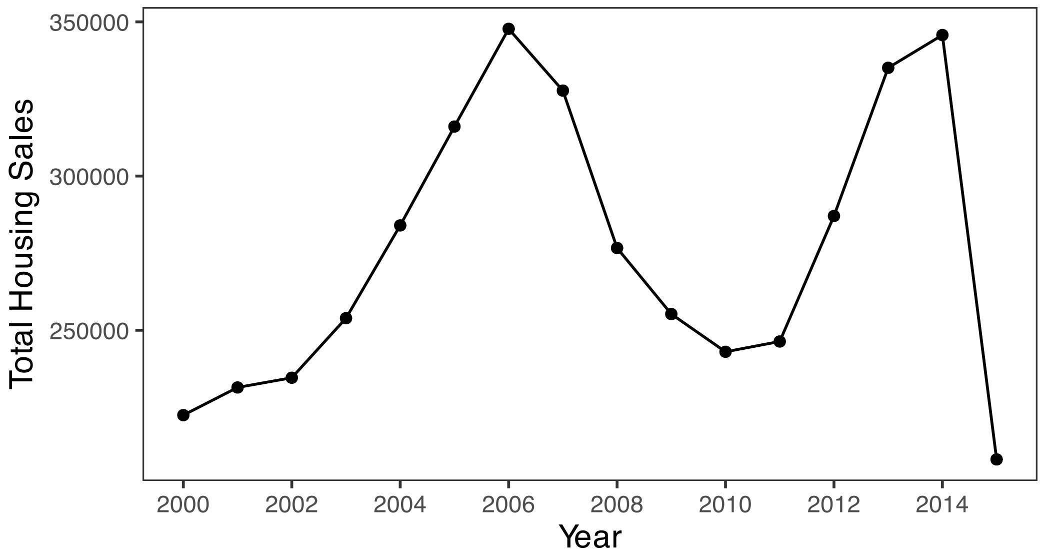
Lines 9-12

Again we are adding our typical labels here

Line 13

This `scale_x_continous` variable might seem weird. However if we look at our data we will see that our year variable is continuous rather than categorical. Further, the initial breaks skip by intervals of 5 between 2000 and 2015. As such, we may want to change this. We can do that with this function call. The `seq` function allows us to dictate the min and max of the x values and how we scale our graph. I've choosen to go by increments of 2.

Figure 5. Total Texas Housing Sales By Year



Source: Data from the ggplot2 data set

Standard Column Bar Graph W/ Std. Error

```
library(tidyverse)
library(jtools)
col_SE_data <- mtcars

col_SE_data <- col_SE_data %>%
  group_by(cyl) %>%
  summarize(n = n(),
            mpg_average = mean(mpg, na.rm = TRUE),
            sd = sd(mpg, na.rm = FALSE),
            se = sd/sqrt(n))

col_SE_plot <- ggplot(col_SE_data,aes(x = as.factor(cyl),
                                      y = mpg_average,
                                      fill = as.factor(cyl))) +
  geom_col(color = "black") +
  geom_errorbar(aes(ymax = mpg_average + se,
                    ymin = mpg_average - se), width = .5) +
  labs(x = "Number of Cylinders",
       y = "Average Fuel Economy (mpg)",
       title = "Figure 6. Average Fuel Economy by Cylinder Count",
       caption = "Source: Data from the mtcars data set") +
  scale_fill_manual(values = c("grey50","grey80","grey100")) +
  jtools::theme_apa() +
  theme(plot.caption = element_text(hjust = 0))
```

```
ggsave("col_se.png")
print(col_SE_plot)
```

Lines 5-10

For the standard error chart, we have to borrow a bit from our previous line chart syntax as we need to manually compute some group level statistics in order to calculate *SE*. Here we're grouping by `cyl` and we need to compute the *n* and *SD* to compute the SE. This syntax shows how to do this

Lines 12-14

We need to provide our `aes()` factors. Here we want `cyl`,`mpg_average` and a `fill` aesthetic (for color)

Line 15

We need to add our standard `geom_col` layer

Lines 16-17

For our error bars, we want to call `geom_errorbar` and designate our ymax (upper level) and ymin (lower level) bands. This will do that

Lines 18-21

Adding our usual labels

Line 22

Modify our colors for the column