Lab Tutorial 4

Ladan Tazik

2023-10-05

Summary: In this lab, we're going to combine our data wrangling skills with data visualization techniques.

Towards the end of the lab, you will find a set of assignment exercises (Assignment 3). These assignments are to be completed and submitted on Canvas.

Lab 04

Pivoting Data

Consider the following data frame, Is the provided data frame in a tidy format?

```
library(tidyverse)
untidy_table <-
  tibble::tribble(
       ~type, ~country, ~"year_1999", ~"year_2000", ~"year_2001",
                  "population", "Afghanistan",19987071, 20595360, 21295360,
                  "cases", "Afghanistan",        745 ,      2666, NA,
                  "population", "Brazil", 172006362,  174594898, 178594898,
                  "cases",      "Brazil", 37737,      80488, 12452,
                  "population", "China",  1272915272, 1280428583, 1310428583,
                  "cases",    "China",   212258,     213766, 313766,
       )

untidy_table
```

```
# A tibble: 6 x 5
  type       country      year_1999  year_2000  year_2001
  <chr>      <chr>            <dbl>      <dbl>      <dbl>
1 population Afghanistan   19987071   20595360   21295360
```

```
2 cases      Afghanistan        745       2666         NA
3 population Brazil       172006362  174594898  178594898
4 cases      Brazil           37737      80488      12452
5 population China       1272915272 1280428583 1310428583
6 cases      China          212258     213766     313766
```

> Answer
>
> The data frame exhibits several issues related to its organization. Instead of utilizing separate columns for individual variables, the "type" column specifies the nature of the values within each row. Columns 3 and 4, which both contain doubles, indicate the year of the values in those columns.

Let's use `pivot_longer` from `dplyr` package to take care of this. Recap:

```
pivot_longer(data, cols, names_to, values_to))
```

Arguments:

- `data` data frame to pivot
- `cols` columns to pivot into longer format.
- `names_to` name of new column to create from the information stored in the *column names* of = the pivoted columns
- `values_to` name of the new column to create from the data stored in *cell values* of the pivoted columns

Edit the following code to produce the output below:

```
table1 <- pivot_longer(data = .. ,
                       cols = ... ,
                        names_to = ... ,
                        values_to = ...)
```

```
table1 <- pivot_longer(data = untidy_table,
                       cols = c('year_1999','year_2000' ,'year_2001'),
                        names_to = "year",
                        values_to = "value")
```

```
table1
```

```
# A tibble: 18 x 4
   type       country     year           value
```

```
   <chr>      <chr>       <chr>          <dbl>
 1 population Afghanistan year_1999   19987071
 2 population Afghanistan year_2000   20595360
 3 population Afghanistan year_2001   21295360
 4 cases      Afghanistan year_1999        745
 5 cases      Afghanistan year_2000       2666
 6 cases      Afghanistan year_2001         NA
 7 population Brazil      year_1999  172006362
 8 population Brazil      year_2000  174594898
 9 population Brazil      year_2001  178594898
10 cases      Brazil      year_1999      37737
11 cases      Brazil      year_2000      80488
12 cases      Brazil      year_2001      12452
13 population China       year_1999 1272915272
14 population China       year_2000 1280428583
15 population China       year_2001 1310428583
16 cases      China       year_1999     212258
17 cases      China       year_2000     213766
18 cases      China       year_2001     313766
```

However, `year_2000` isn't really a value of `year`. It would be better for the values to contains the numeric value only. We can remove the "recorded_" prefix with pivot_longer()'s `names_prefix` argument. As stated in the help file `?pivot_longer`, this argument accepts a regular expression and is used to remove matching text from the start of each variable name. Note that I used the negative indexing to exclude the first two columns in pivoting, rather than passing the names of the columns.

```
(table1 <- pivot_longer(data = untidy_table, cols = -c(1:2),
                        names_to = "year",
                        names_prefix = "year_",
                        values_to = "value"))
```

```
# A tibble: 18 x 4
   type       country     year       value
   <chr>      <chr>       <chr>      <dbl>
 1 population Afghanistan 1999    19987071
 2 population Afghanistan 2000    20595360
 3 population Afghanistan 2001    21295360
 4 cases      Afghanistan 1999         745
 5 cases      Afghanistan 2000        2666
 6 cases      Afghanistan 2001          NA
 7 population Brazil      1999   172006362
```

```
 8 population Brazil      2000   174594898
 9 population Brazil      2001   178594898
10 cases      Brazil      1999       37737
11 cases      Brazil      2000       80488
12 cases      Brazil      2001       12452
13 population China       1999  1272915272
14 population China       2000  1280428583
15 population China       2001  1310428583
16 cases      China       1999      212258
17 cases      China       2000      213766
18 cases      China       2001      313766
```

Let's look at structure of the new table;

```
  str(table1)
```

```
tibble [18 x 4] (S3: tbl_df/tbl/data.frame)
 $ type   : chr [1:18] "population" "population" "population" "cases" ...
 $ country: chr [1:18] "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
 $ year   : chr [1:18] "1999" "2000" "2001" "1999" ...
 $ value  : num [1:18] 19987071 20595360 21295360 745 2666 ...
```

It's probably more useful to store year as a integer. We can tell `pivot_longer()` our desired type for the `names_to` column by using the optional `names_transform` argument.

```
  (table1 <- pivot_longer(data = untidy_table, cols = -c(1:2),
                          names_to = "year",
                          names_prefix = "year_",
                          names_transform = list(year = as.integer),
                          values_to = "value"))
```

```
# A tibble: 18 x 4
   type       country      year      value
   <chr>      <chr>       <int>      <dbl>
 1 population Afghanistan  1999   19987071
 2 population Afghanistan  2000   20595360
 3 population Afghanistan  2001   21295360
 4 cases      Afghanistan  1999        745
 5 cases      Afghanistan  2000       2666
 6 cases      Afghanistan  2001         NA
 7 population Brazil       1999  172006362
```

```
 8 population Brazil       2000  174594898
 9 population Brazil       2001  178594898
10 cases      Brazil       1999      37737
11 cases      Brazil       2000      80488
12 cases      Brazil       2001      12452
13 population China        1999 1272915272
14 population China        2000 1280428583
15 population China        2001 1310428583
16 cases      China        1999     212258
17 cases      China        2000     213766
18 cases      China        2001     313766
```

We can also utilize a similar conversion for the `values_to` column as well. Edit the following code to convert the `value` to integer (to save memory da!). See here to learn about the parameter.

```
table1 <- pivot_longer(data = untidy_table,
                       cols = c('year_1999','year_2000' ,'year_2001'),
                        names_to = "year",
                        names_prefix = "year_",
                        names_transform = list(year = as.integer),
                        values_to = "value"
                        # your code here
                        # ??? = ....
                        )
```

```
table1 <- pivot_longer(data = untidy_table,
                       cols = c('year_1999','year_2000' ,'year_2001'),
                        names_to = "year",
                        names_prefix = "year_",
                        names_transform = list(year = as.integer),
                        values_to = "value",
                        values_transform = list(value = as.integer)
                        )
```

For downstream analysis we should make our table wider, For this we use `pivot_wider()` which adds columns making the data frame wider.

This corrects the issue that each variable does not have its own column. For example column `type` can be two different columns; seperate column for `population` and another column for `case`.

Recap:

```
pivot_wider(data, names from = "", values_from = "")
```

- `data` data frame to pivot
- `names_from` column whose values are the names of the new columns
- `values_from` column whose values are the values of the new columns

Now write your code to create a wider table describe above.

Hint

Step 1: Identify the column whose values will supply the column names (`name_from`).

Step 2: Identify the column whose values will supply the column values. (`values_from`)

```
wide_table1 <- pivot_wider(table1,
                           names_from = ... ,  # step 1
                           values_from = ... ) # step 2
```

```
wide_table1 <- pivot_wider(table1,
                           names_from = type,
                           values_from = value )
```

```
wide_table1
```

```
# A tibble: 9 x 4
  country      year population  cases
  <chr>       <int>      <int>  <int>
1 Afghanistan  1999   19987071    745
2 Afghanistan  2000   20595360   2666
3 Afghanistan  2001   21295360     NA
4 Brazil       1999  172006362  37737
5 Brazil       2000  174594898  80488
6 Brazil       2001  178594898  12452
7 China        1999 1272915272 212258
8 China        2000 1280428583 213766
9 China        2001 1310428583 313766
```

Now create a new column, `cases_per_cap`, that is the number of cases divided by the total population that year. saved it to a new data frame `table2`.

```
# ??? recall which function we use to create new cols
table2 <- ???(wide_table1,
```

```
              cases_per_cap = ... )
# ... should be a vector of length n
# where n is the number of rows in wide_table1
```

```
table2 <- mutate(wide_table1, cases_per_cap = cases / population)
```

This next example demonstrates how to use the `case_when()` function. `case_when()` is very useful when creating a new categorical variable based on already existing numeric variables.

This function does not have any explicit parameters. Instead, there can be any number of logical statements (e.g. cases / population < 0.0001) with a corresponding `~`.

`~` creates a formula in R. On the other side of the `~` is the value of the variable if the logical statement evaluates to `TRUE`.

Make sure only one formula evaluates as TRUE for each observation. The code chunk below demonstrates the syntax for `case_when()`.

```
(new_table <- mutate(table2, cases_per_cap = case_when(
  cases / population < 0.0001 ~ "low",
  cases / population  >= 0.0001 & cases / population < 0.0003 ~ "medium",
  cases / population >= 0.0003 ~ "high")))
```

```
# A tibble: 9 x 5
  country      year population  cases cases_per_cap
  <chr>       <int>      <int>  <int> <chr>
1 Afghanistan  1999   19987071    745 low
2 Afghanistan  2000   20595360   2666 medium
3 Afghanistan  2001   21295360     NA <NA>
4 Brazil       1999  172006362  37737 medium
5 Brazil       2000  174594898  80488 high
6 Brazil       2001  178594898  12452 low
7 China        1999 1272915272 212258 medium
8 China        2000 1280428583 213766 medium
9 China        2001 1310428583 313766 medium
```

This example creates a new variable that designates the level of cases per capita as low, medium, or high. The new variable is a string instead of numeric. This variable is perfect for using the factor data type, as it has inherent ordered levels.

Write code below to turn this variable into a factor. Remember that when using mutate, if you set something equal to a variable name already in use, it will overwrite that variable.

```
new_table <- mutate(new_table, cases_per_cap =
          factor(cases_per_cap, levels = c("low", "medium", "high")))
```

Visualization

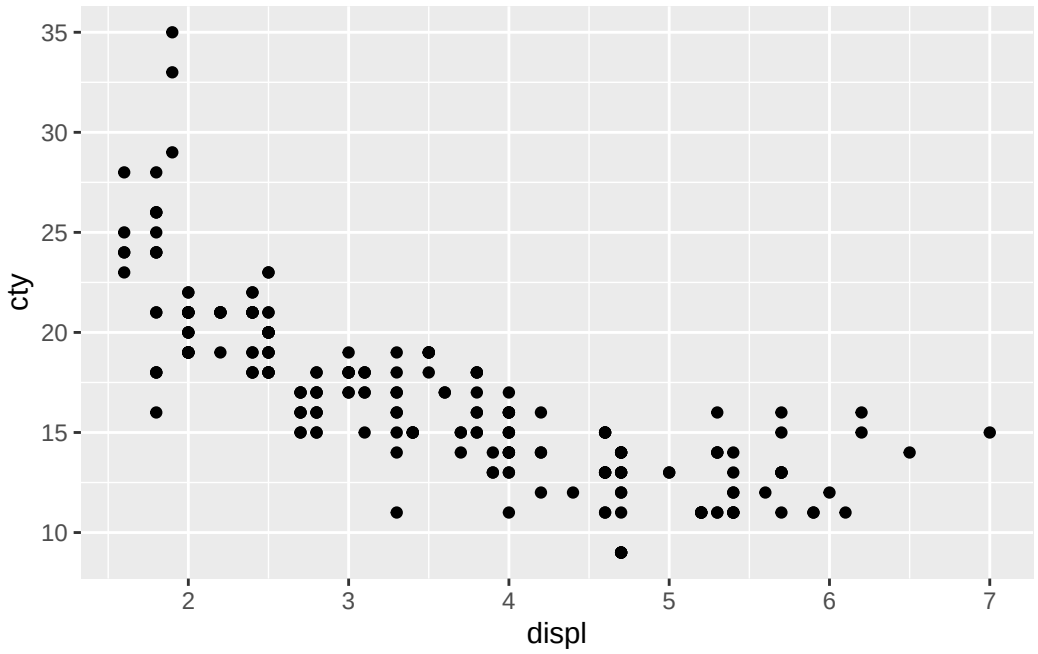
Recap: `ggplot2` uses the theoretical framework called the grammar of graphics as the basis for its organization. There are three essential components to a graphic

1. `data`: the clean data containing the variable(s) of interest
2. `geom`: the type of geometric object in the plot e.g. line, point, bar, box
3. `aes`: aesthetic attributes of the geometric object e.g. x/y variables, color, shape, size can be mapped (coordinated) to variables

Example 1: Create simple scatter plot that shows the relationship between engine displacement and city miles per gallon for the cars whithin `mpg` dataset. You may find the data visualization cheatsheet helpful here.

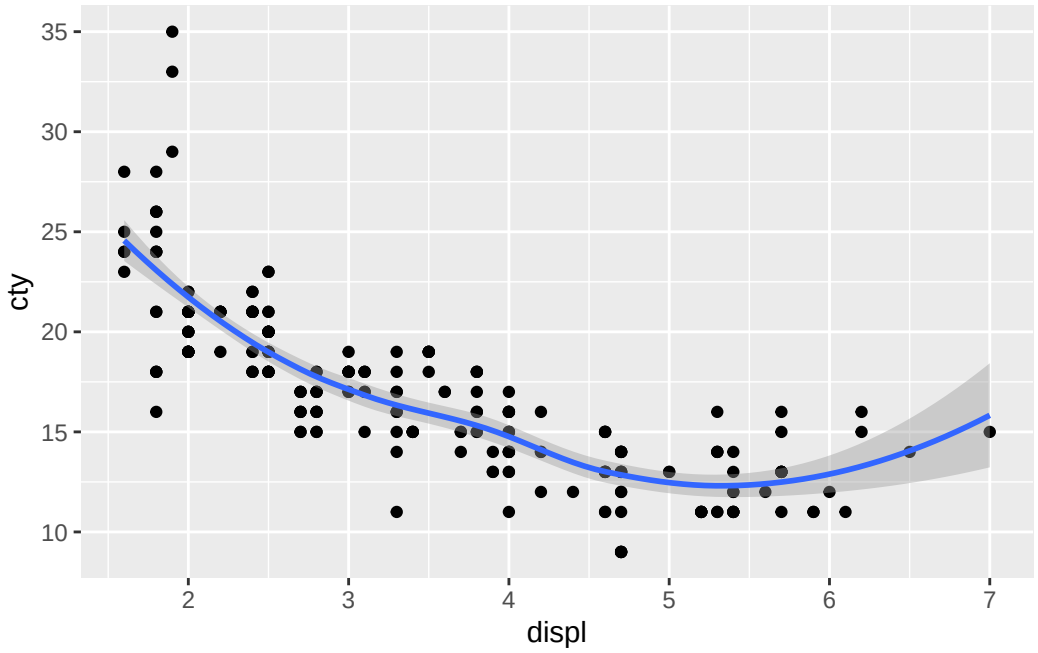
```
plot1 <- ggplot(data = ...,
       aes(...)) + geom_ ....
```

```
plot1 <- ggplot(data = mpg, aes(x = displ, y = cty)) +
  geom_point()
plot1
```



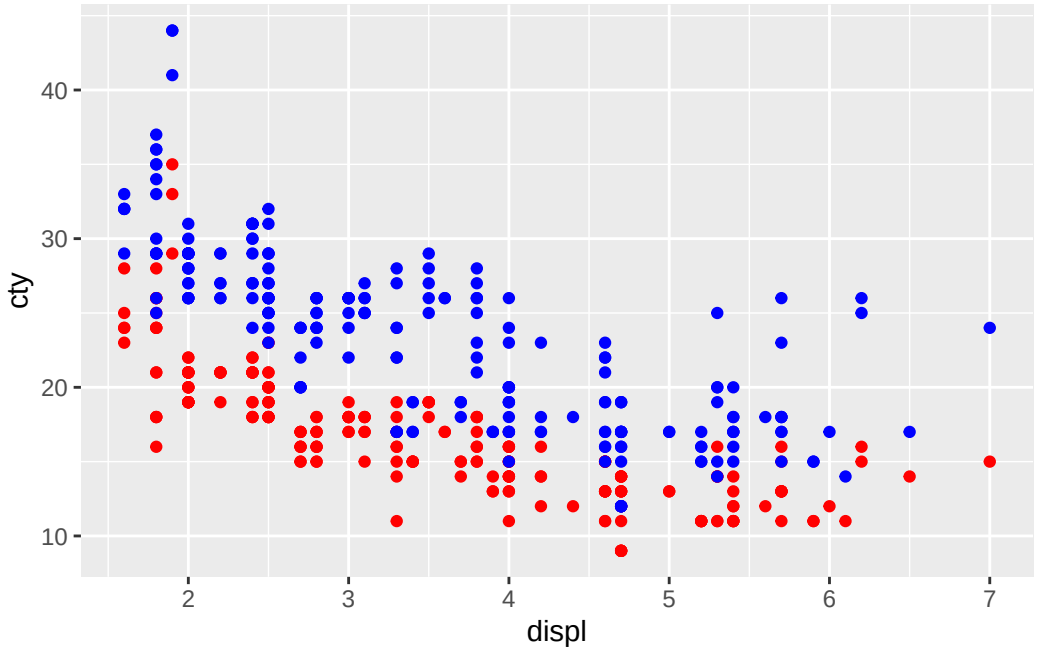
In order to capture the relationship better we can fit a line to this points. This can be done by adding `geom_smooth`.

```
plot1 + geom_smooth()
```



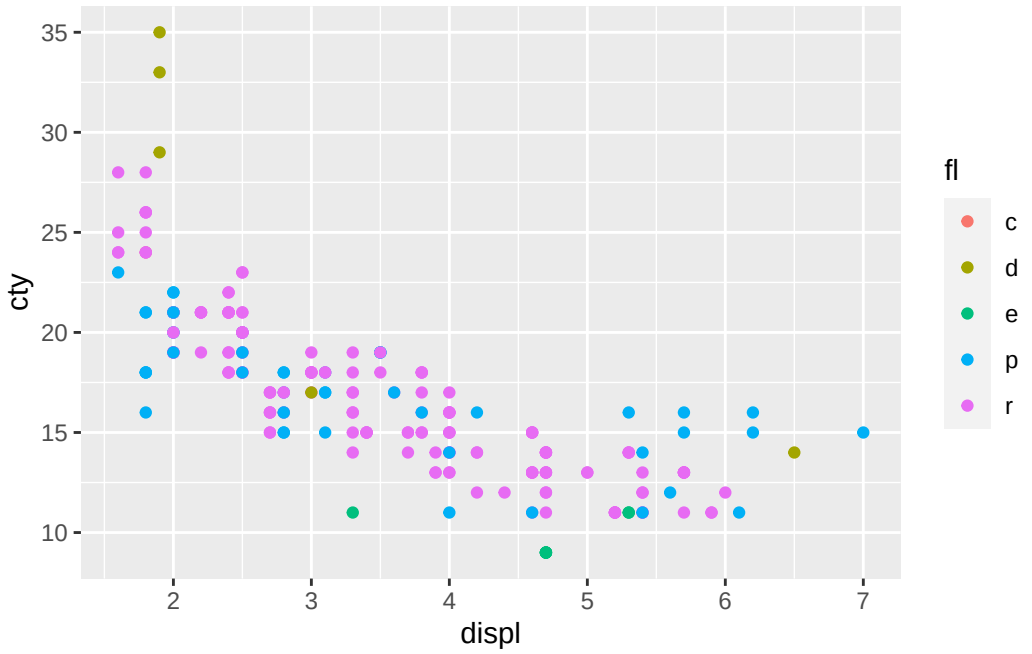
Example 2: Let's plot two different variables on the same axis. Note that we need to make sure that both variables have the same unit.

```
ggplot(data = mpg)+
  geom_point(aes(x = displ, y = cty), color = 'red') +
  geom_point(aes(x = displ, y = hwy), color = 'blue')
```



Example 3: We want to differentiate between different fuel type. we can use this categorical variable as the color! This will automatically add a legend to our plot.

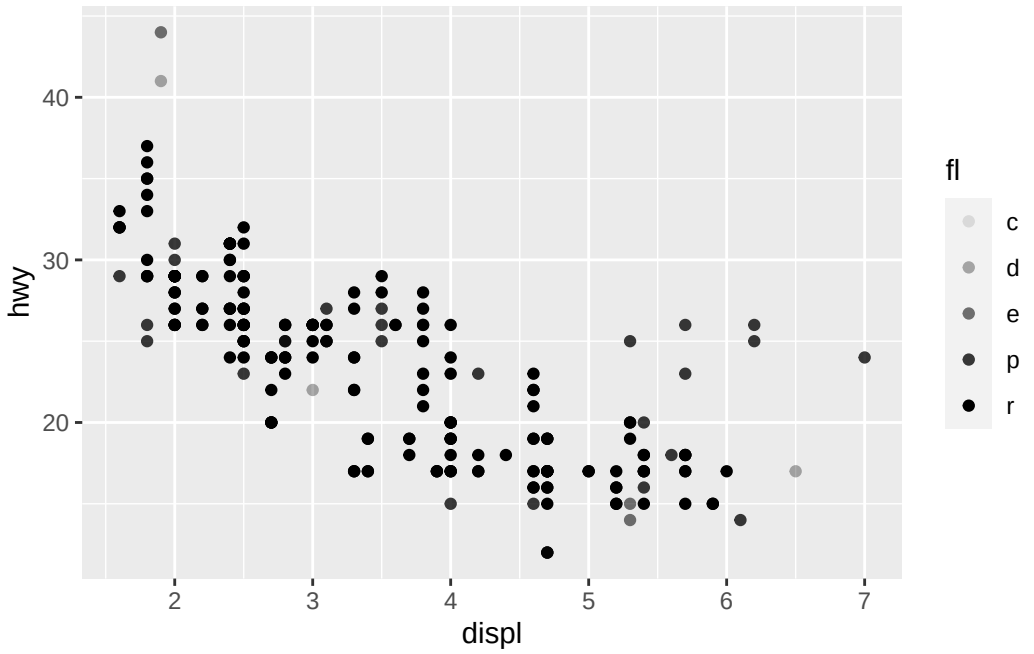
```
ggplot(data = mpg) +
  geom_point(aes(x = displ, y = cty, color = fl))
```



Alternatively, we can also use `alpha` to differentiate between different fuel type. It provides different shades and is better suited for black/white plots.

```
ggplot(data = mpg) +
  geom_point(aes(x = displ, y = hwy, alpha = fl))
```

```
Warning: Using alpha for a discrete variable is not advised.
```



Properties of `geom`

This section will explore other visual properties of the geom (aesthetics) like `size`, `shape`.

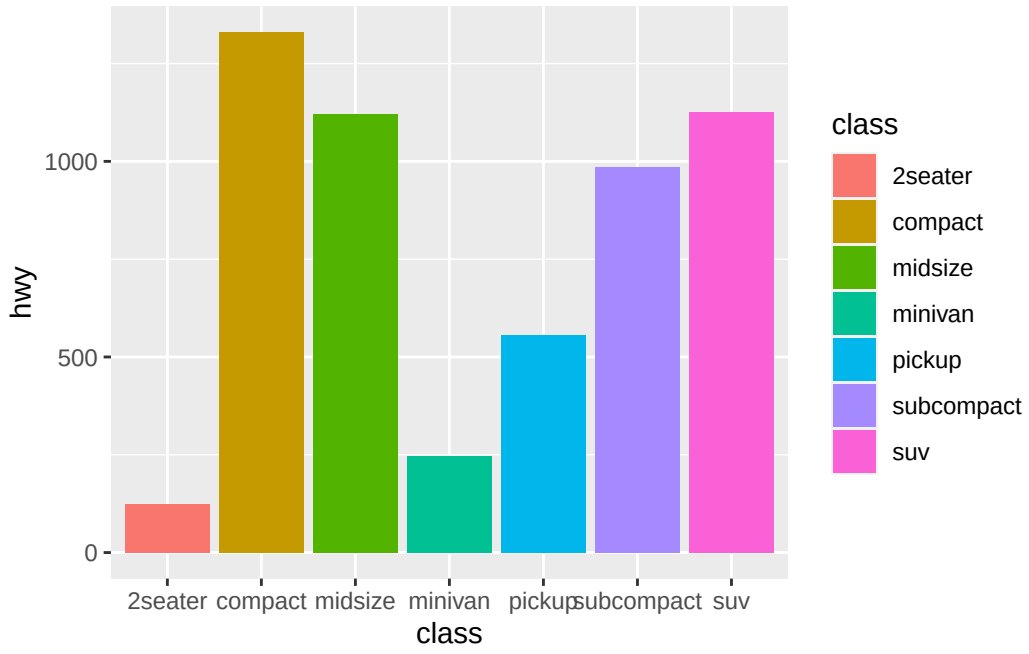
In general, we should know what type of the graphs are suitable for our purpose. One factor that we need to know is the type of the data that we're plotting in addition of the number of the variables that we're using.For example:

When having one categorical (discrete) variable one continues (numeric) variable: bar plot, violin plot, boxplot.

The following graph shows a bar plot of `hwy` for different `class` of cars. Note that we have a colored plot by using `fill`. This automate coloring our graphs.

Which class has the max and min fuel consumption?

```
mpg |>
  ggplot(aes(x = class, y = hwy, fill= class)) +
  geom_col()
```

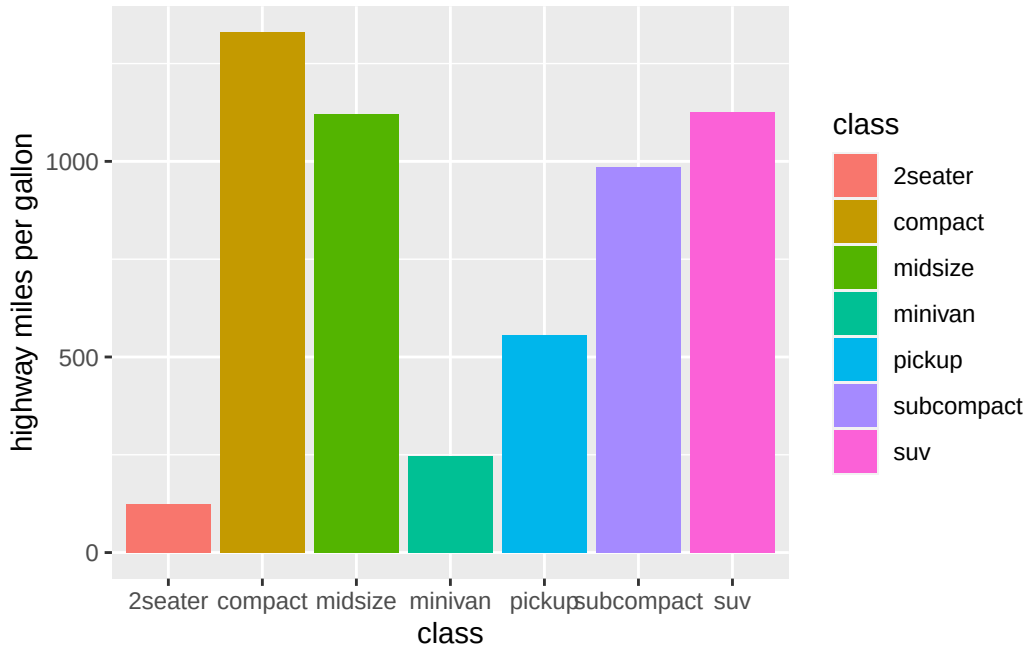


Replicate the above graph by using `color` instead of `fill`. what is the difference? which one do you prefer?

```
#your code
```

Now let's change the name of y axis to highway miles per gallon.

```
mpg |>
  ggplot(aes(x = class, y = hwy, fill = class)) +
  geom_col()+
  labs( y= "highway miles per gallon")
```



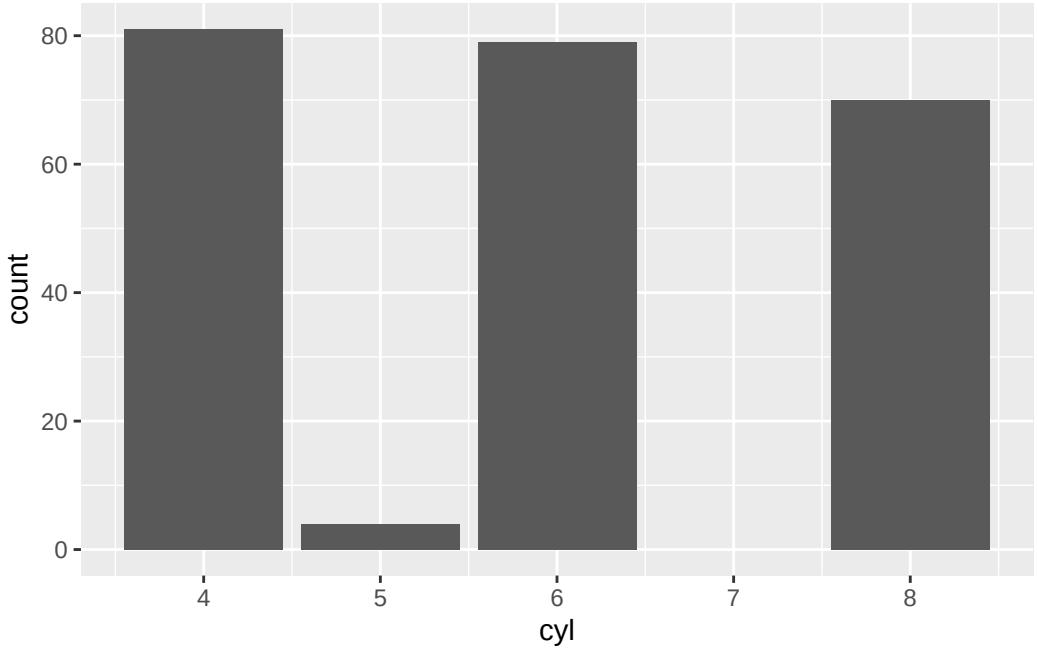
Add a title to the previous graph.

```
# your code
```

When creating a graph of only one numeric variable : histogram, density plot, dotplots.

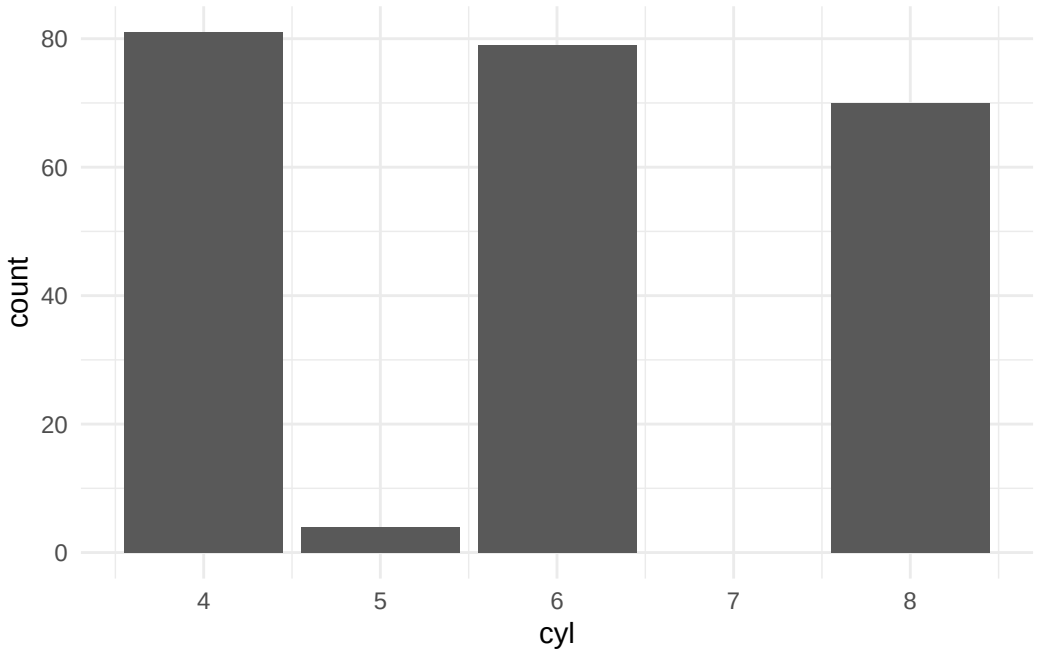
Now, let's create a bar plot of number of cylinders. The majority of the cars in our dataset has … cylinder?

```
mpg |>
  ggplot(aes(x=cyl)) +
  geom_bar()
```



We can also change the theme of our graphs, my favirote is `theme_minimal()`.

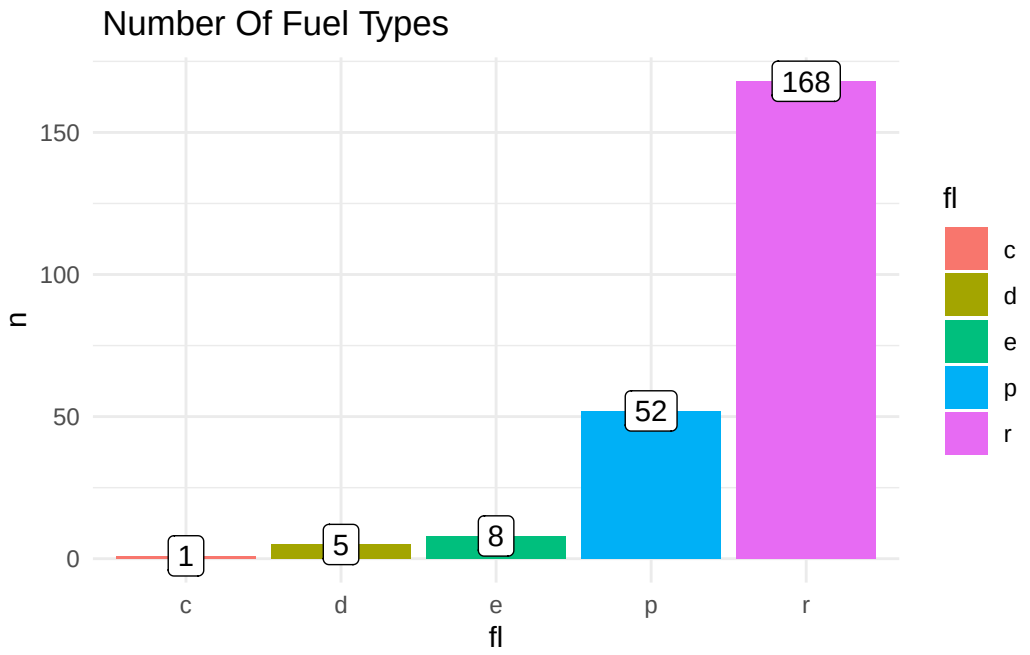
```
mpg |>
  ggplot(aes(x=cyl)) +
  geom_bar()+
  theme_minimal()
```



Explore other themes to see which one you like the best :)

Let's see how many cars we have for each fuel type. note that we're using `geom_col` since we have a categorical (discrete) data and a numeric variable. `n` is mapped to `count(fuel type)` .

```
mpg |>
 count(fl) |>
 ggplot() + geom_col(aes(x = fl, y = n, fill = fl)) +
 geom_label(aes(x = fl, y = n, label = n)) +
 labs(title = " Number Of Fuel Types ")+
 theme_minimal()
```



Resources

- Chapter 2 of the book Data Wrangling and Visualization Guide
- Chapter 6 (advanced pivoting) from Data Wrangling
- ggplot2 Cheat Sheet

Assignment 3

Deadline: Wednesday Oct 18th, 11:59 p.m.

The following questions are to be completed and submitted on Canvas.

1. We are using penguins dataset for the following questions. Download the `penguins.csv` dataset from the Datasets module on Canvas.

    a. Use the `read_csv` function from **readr**[1] package to read the csv file into your workspace name it to an object called `penguins`. (1 point)

    b. Provide a brief summary of the dataset. How many rows and columns are there? We've been using `str` from base R functions, do your research to find the alternative function from `tidyverse`, more specifically `dplyr` to provide the summary. (2 points)

[1] which is part of tidyverse

c. Write a pipe line that shows summary of the numeric columns only ? (2 points)

d. Remove the `NA`s from the data set.(1 point)

e. Add a new column called size that has categorize the penguins based on the following (4 points)

- if the body mass is greater than 4200, penguin size is `big`
- if body mass is lower than or equal than 4200, penguin size is `small`.

2. Create a violin plot for the depth of the penguins bill for each size.(2 points)

3. Create a simple scatter plot to show the relationship between `bill_length` and `bill_depth` for different species. (2 points)

a. Is there a consistent pattern of relationship among species? (1 point)

4. Create a box plot of the `mass_body` within each island for different species (3 points)

a. Do you see any outliers? (1 point)

b. Change the x and y labels to "Island" and " Body Mass (g)", respectively. (1 point)

c. Add an appropriate title to your graph. (1 point)

d. Change the theme to `theme_classic`. (1 point)

5. Create a colored bar plot that show number of species in Dream island. (5 points)

Total possible points: 27