R for Data Science - Solutions Manual

Jonathan Mendes de Almeida
jonathanalmd@gmail.com
jonathan@aluno.unb.br
@jonyddev (github)

Mai 06, 2018

Solutions to the exercises in **R for Data Science (Garrett Grolemund & Hadley Wickham)**. The individual R files are available in my github - **@jonyddev** - repository called **R4DataScience-Solutions**. *If you have any questions about my answers to these exercises do not hesitate to enter in contact with me.*

Useful links

- **RStudio Cheat Sheets**
- **ggplot2 Documentation**
- **ggplot2 Cheat Sheet**

Prerequisites

```
library(tidyverse) # Prerequisite for your life as a (R) Data Scientist student
```

```
## -- Attaching packages ---------------------------------------------------------------------------
## √ ggplot2 2.2.1     √ purrr   0.2.4
## √ tibble  1.4.2     √ dplyr   0.7.4
## √ tidyr   0.8.0     √ stringr 1.3.0
## √ readr   1.1.1     √ forcats 0.3.0
## -- Conflicts ------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(nycflights13) # Chapter 5
```

To install tidyverse library:

```
install.packages('tidyverse')
```

mpg data frame

```
mpg = ggplot2::mpg
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model    displ  year   cyl trans   drv     cty   hwy fl
##    <chr>        <chr>    <dbl> <int> <int> <chr>   <chr> <int> <int> <chr>
##  1 audi         a4         1.8  1999     4 auto(l~ f        18    29 p
```

```
##  2 audi         a4         1.8  1999     4 manual~ f        21    29 p
##  3 audi         a4         2    2008     4 manual~ f        20    31 p
##  4 audi         a4         2    2008     4 auto(a~ f        21    30 p
##  5 audi         a4         2.8  1999     6 auto(l~ f        16    26 p
##  6 audi         a4         2.8  1999     6 manual~ f        18    26 p
##  7 audi         a4         3.1  2008     6 auto(a~ f        18    27 p
##  8 audi         a4 quat~   1.8  1999     4 manual~ 4        18    26 p
##  9 audi         a4 quat~   1.8  1999     4 auto(l~ 4        16    25 p
## 10 audi         a4 quat~   2    2008     4 manual~ 4        20    28 p
## # ... with 224 more rows, and 1 more variable: class <chr>
```

Chapter 1 - Introduction

No exercises in this chapter.

Chapter 2 - Introduction 2

No exercises in this chapter.

Chapter 3 - Visualize

3.2.4 Exercises

Exercise 1

Run `ggplot(data = mpg)`. What do you see?

```
ggplot(data = mpg)
```

(Answer) An empty plot. To see some nice plots we should add some geom_function to map some points. Add a good caption for each axis is great to make your plot easier to read and understand!

Exercise 2

How many rows are in `mpg`? How many columns?

```
nrow(mpg)
```

```
## [1] 234
```

```
ncol(mpg)
```

```
## [1] 11
```

(Answer) 234 rows and 11 columns

Alternative method to check the number of rows and columns of a data frame:

```
glimpse(mpg)
```

```
## Observations: 234
## Variables: 11
## $ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "...
## $ model        <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 qua...
## $ displ        <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0,...
## $ year         <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1...
## $ cyl          <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6...
## $ trans        <chr> "auto(l5)", "manual(m5)", "manual(m6)", "auto(av)...
```

```
## $ drv          <chr> "f", "f", "f", "f", "f", "f", "f", "4", "4", "4",...
## $ cty          <int> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 1...
## $ hwy          <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 2...
## $ fl           <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p",...
## $ class        <chr> "compact", "compact", "compact", "compact", "comp...
```

Exercise 3

What does the `drv` variable describe? Read the help for ?mpg to find out. *run ?mpg in RStudio console and check the 'help' tab)*

```
mpg['drv']
```

```
## # A tibble: 234 x 1
##    drv
##    <chr>
##  1 f
##  2 f
##  3 f
##  4 f
##  5 f
##  6 f
##  7 f
##  8 4
##  9 4
## 10 4
## # ... with 224 more rows
```

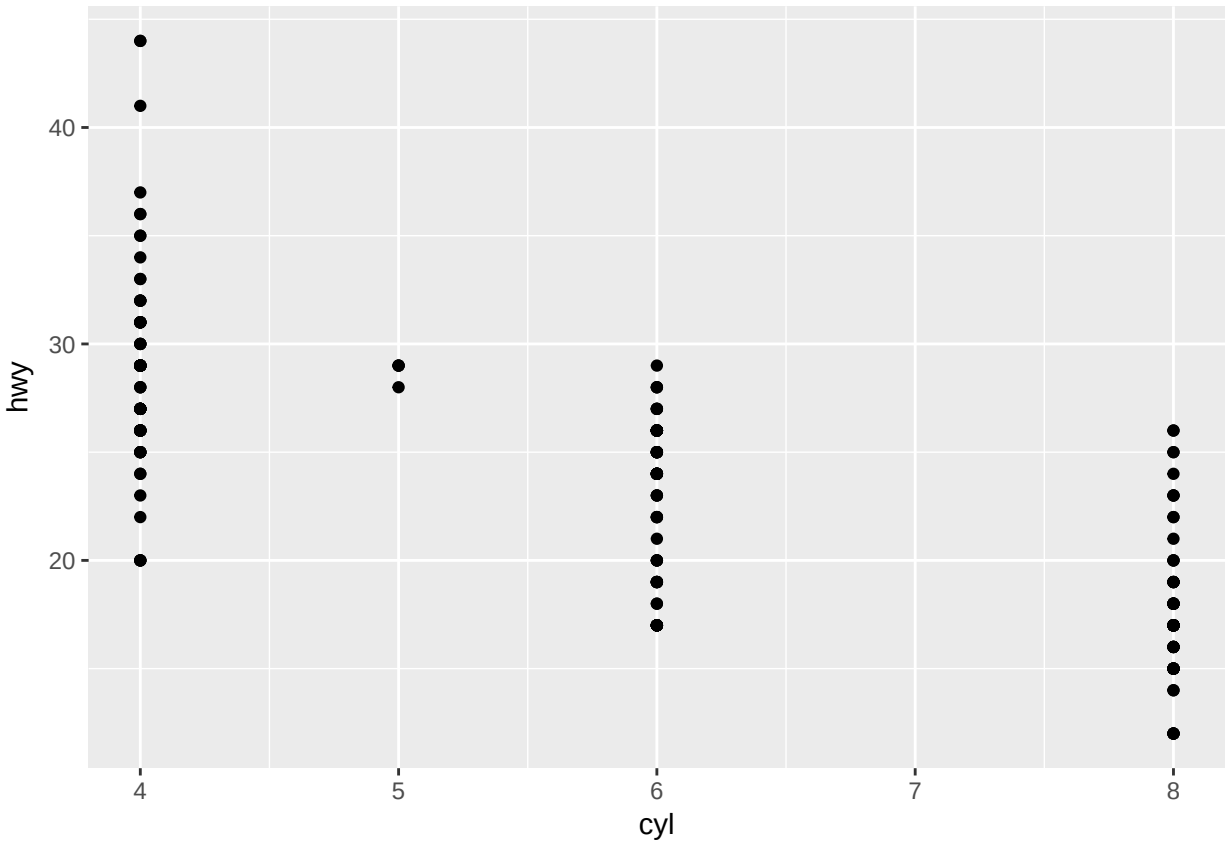
(Answer) The **drv** variable describes the traction control system. There are 3 possible values for `drv` variable (**variable** : *description*):

- **f**: *front-wheel drive*
- **r**: *rear wheel drive*
- **4**: *4wd*

Exercise 4

Make a scatterplot of `hwy` vs `cyl`

```
ggplot(data = mpg) + geom_point(mapping = aes(x = cyl, y = hwy))
```

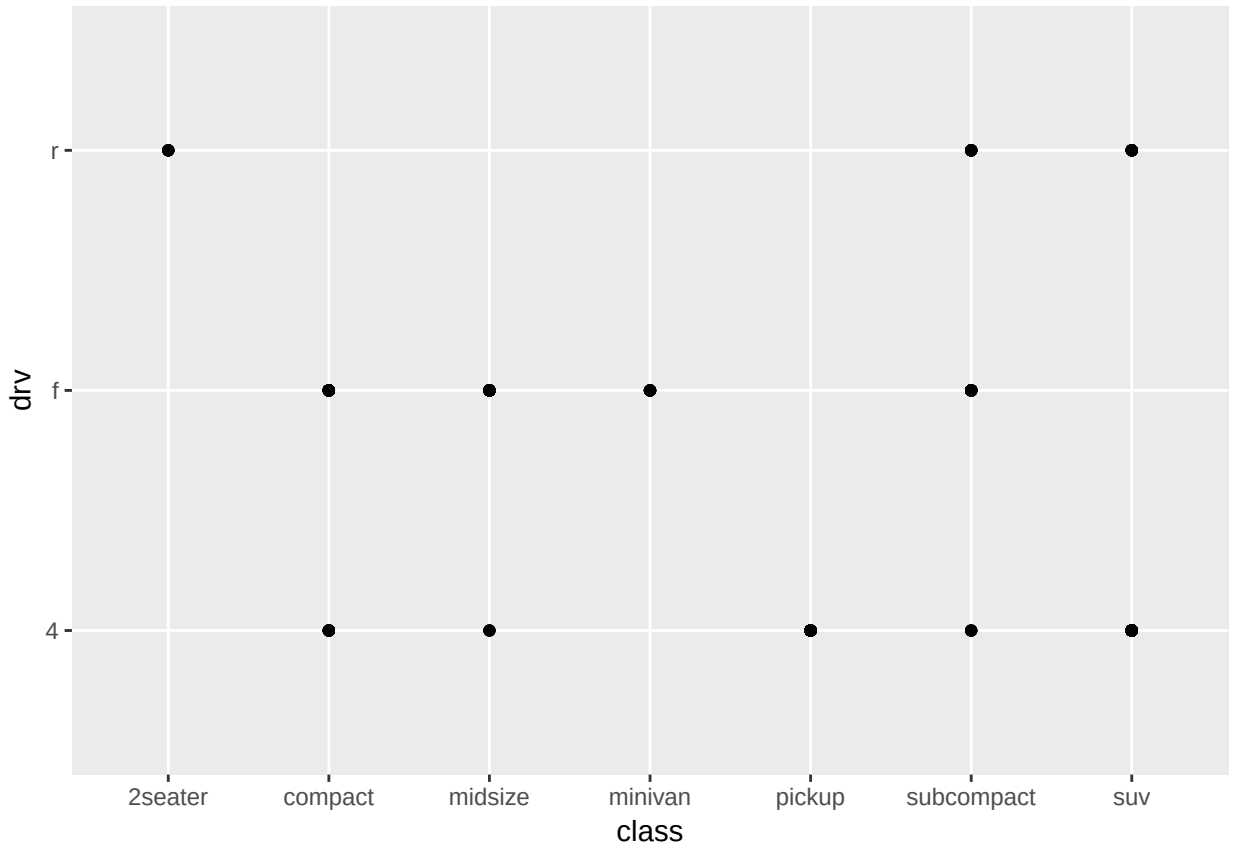


(Answer) We add a definition of the data used in x,y axis and add geom_point function mapping these points. Here we are using aes(colour = class) (to associate the name of the aesthetic with a variable to display) to plot using a different colour for each class present in our data frame (car type: 2seater, compact, midsize, minivan, pickup, subcompact, suv).

Exercise 5

What happens if you make a scatterplot of *class* **vs** *drv*? Why is the plot not useful?

```
ggplot(data = mpg, aes(x = class, y = drv)) + geom_point()
```



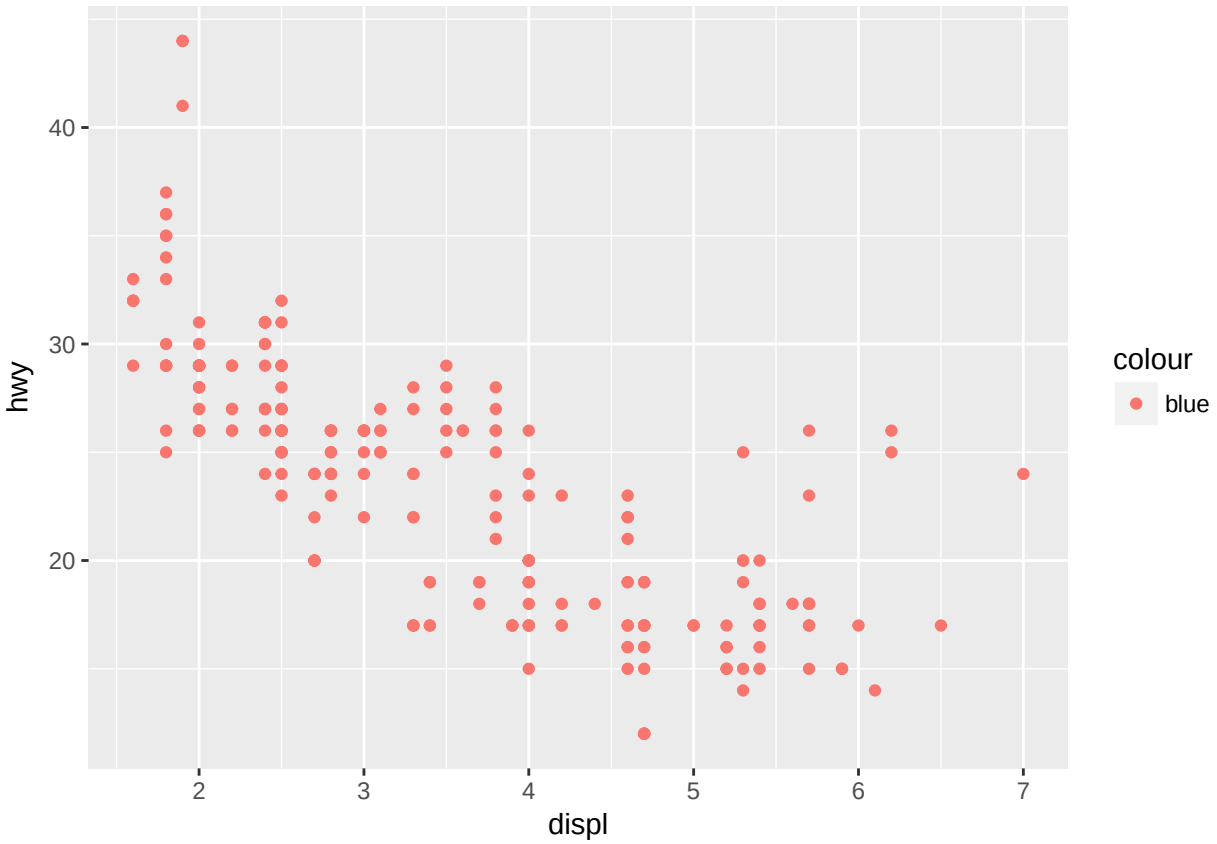
(Answer) This plot is not useful because `class` and `drv` are factor variables. Each possible value of these two variables is limited by a set (r, f and 4 are the possible values for `drv` and 2seater, compact, midsize, minivan, pickup, subcompact and suv are the possible values for `class`). This plot is pretty useless to perform a data analysis.

3.3.1 Exercises

Exercise 1

What's gone wrong with this code? Why are the points not blue?

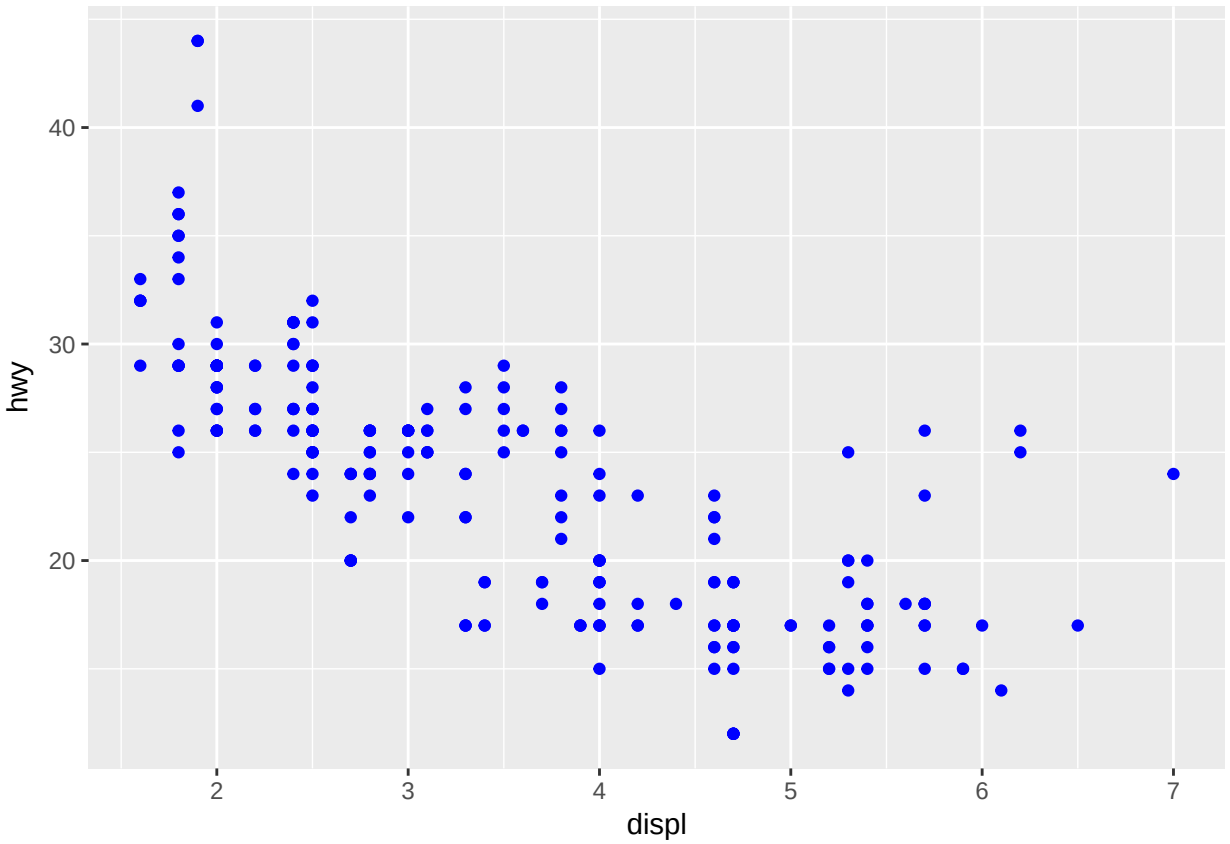
```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, color = 'blue'))
```



(Answer) The `color` argument is not in the correct place. The `color` argument is included inside the `mapping` argument so it is treated as an aesthetic, which receives a variable (like we used `class` as argument in previous exercise). In this case, the `color` argument is interpreted as a variable with only one value (which is 'blue' in this case).

If the goal is to plot all these points using blue, the correct code is:

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



Exercise 2

> Which variables in `mpg` are categorical? Which variables are continuous? (**Hint:** *type* ***?mpg*** *(using RStudio console) to read the documentation for the dataset*). How can you see this information when you run `mpg`?

If you are not able to classify each variable as categorical or continuous by checking the description of each variable (by typing `?mpg`) you can print the data frame and R will answer this for you (another way to check this information is using the `glimpse()` function).

```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model    displ  year   cyl trans   drv     cty   hwy fl
##    <chr>        <chr>    <dbl> <int> <int> <chr>   <chr> <int> <int> <chr>
##  1 audi         a4         1.8  1999     4 auto(l~ f        18    29 p
##  2 audi         a4         1.8  1999     4 manual~ f        21    29 p
##  3 audi         a4         2    2008     4 manual~ f        20    31 p
##  4 audi         a4         2    2008     4 auto(a~ f        21    30 p
##  5 audi         a4         2.8  1999     6 auto(l~ f        16    26 p
##  6 audi         a4         2.8  1999     6 manual~ f        18    26 p
##  7 audi         a4         3.1  2008     6 auto(a~ f        18    27 p
##  8 audi         a4 quat~   1.8  1999     4 manual~ 4        18    26 p
##  9 audi         a4 quat~   1.8  1999     4 auto(l~ 4        16    25 p
## 10 audi         a4 quat~   2    2008     4 manual~ 4        20    28 p
## # ... with 224 more rows, and 1 more variable: class <chr>
```

As you can see, the information is given at top of each column within '<>'. If the variable is categorical, it

will have a class of 'character' (represented as `<chr>`). So, once you know where to find this is information is easy to anwser which variable is categorical and which is continuous.

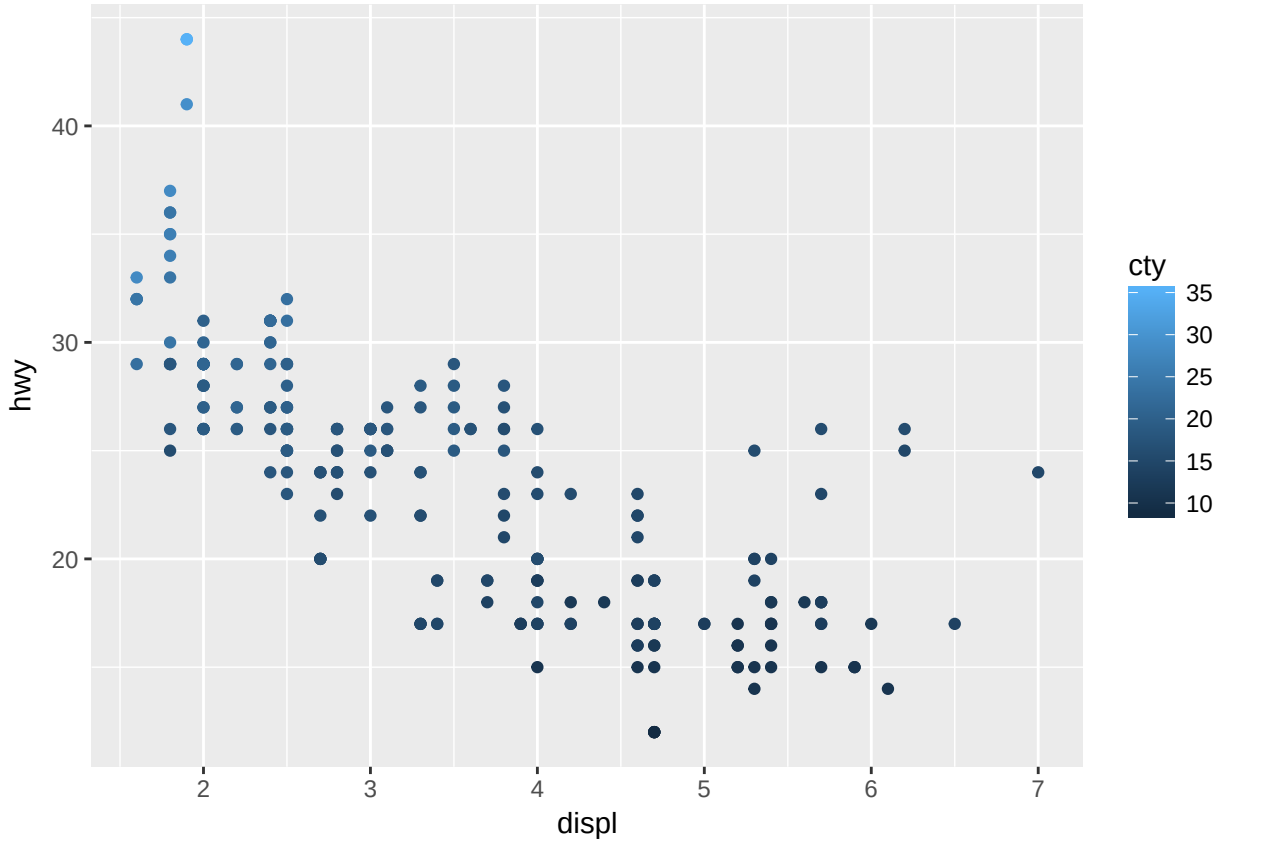
- **model**: categorical
- **displ**: continuous
- **year**: continuous
- **cyl**: continuous
- **trans**: categorical
- **drv**: categorical
- **cty**: continuous
- **hwy**: continuous
- **fl**: categorical
- **class**: categorical

Exercise 3

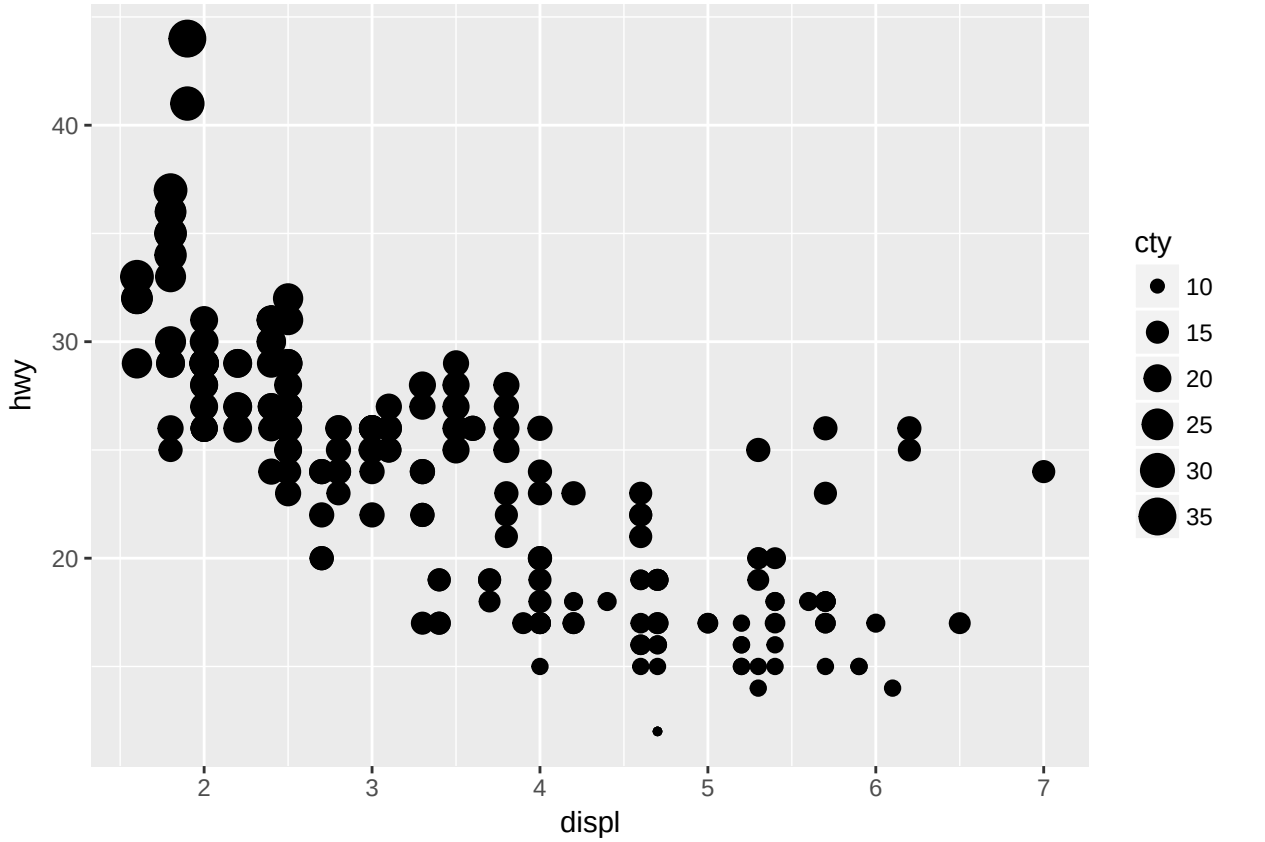
> Map a continuous variable to `color`, `size` and `shape`. How do these aesthetics behave differently for **categorical** *vs* **continuous** variables?

(Answer) *Using the varible* ***cty*** *(city miles per gallon) - which is a continuous variable.*

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, colour = cty))
```



```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, size = cty))
```




```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, shape = cty))
```

When **mapped to colour**: the continuous variable uses a scale that varies using tons of blue (light to dark).

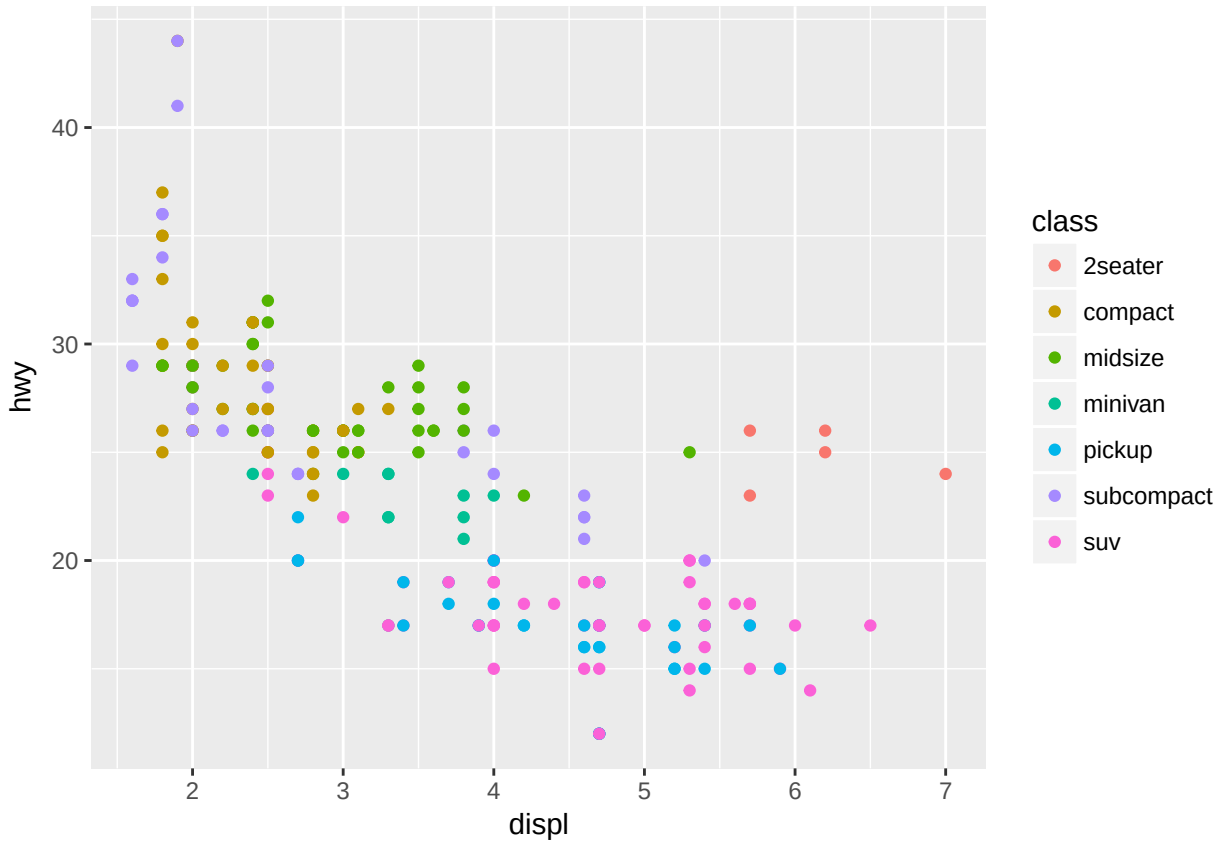
When **mapped to size**: the continuous variable uses a scale that varies using different sizes.

These information is easy to verify by checking these two previous plots.

However, when **mapped to shape**, R will give an error (a continuous variable can not be mapped to shape). This is because shapes does not have a natural order.

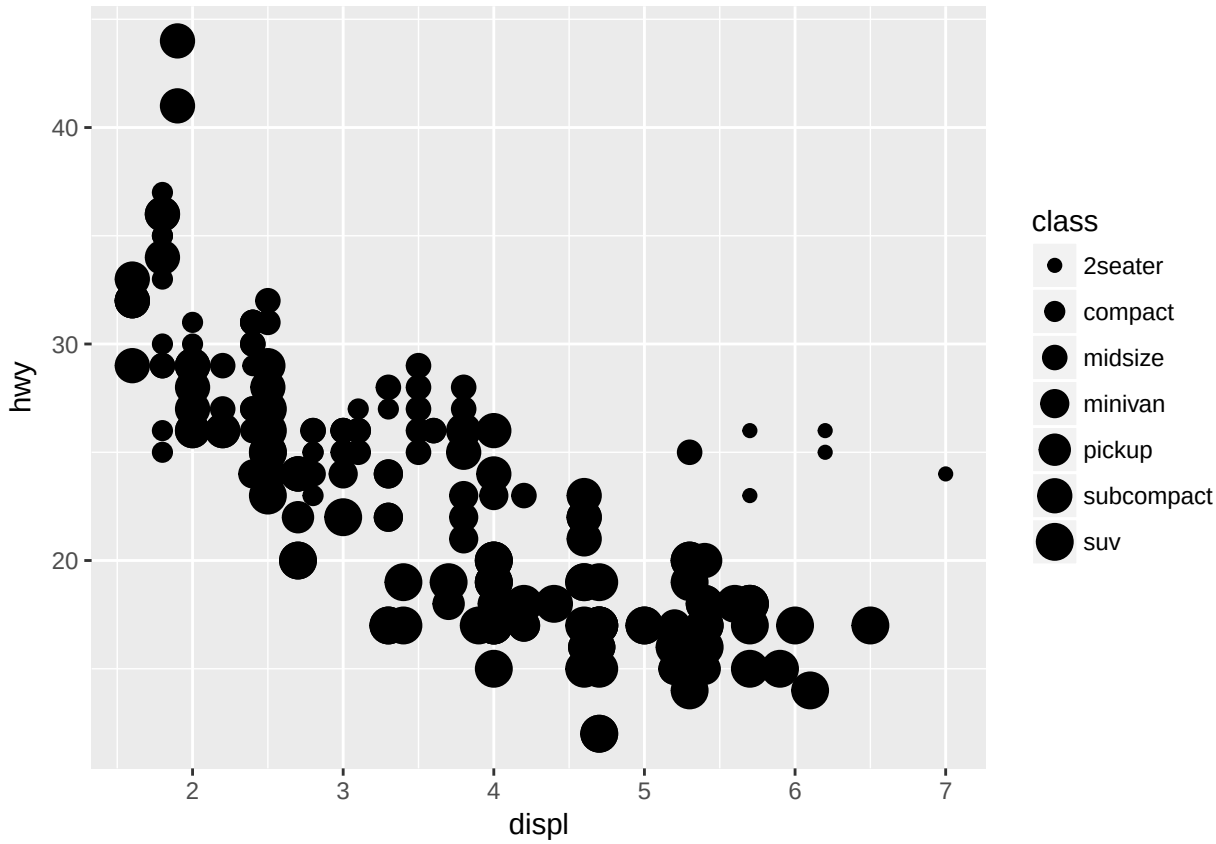
Now, let's plot using `class` *- which is a categorical variable.*

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, colour = class))
```




```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, size = class))
```

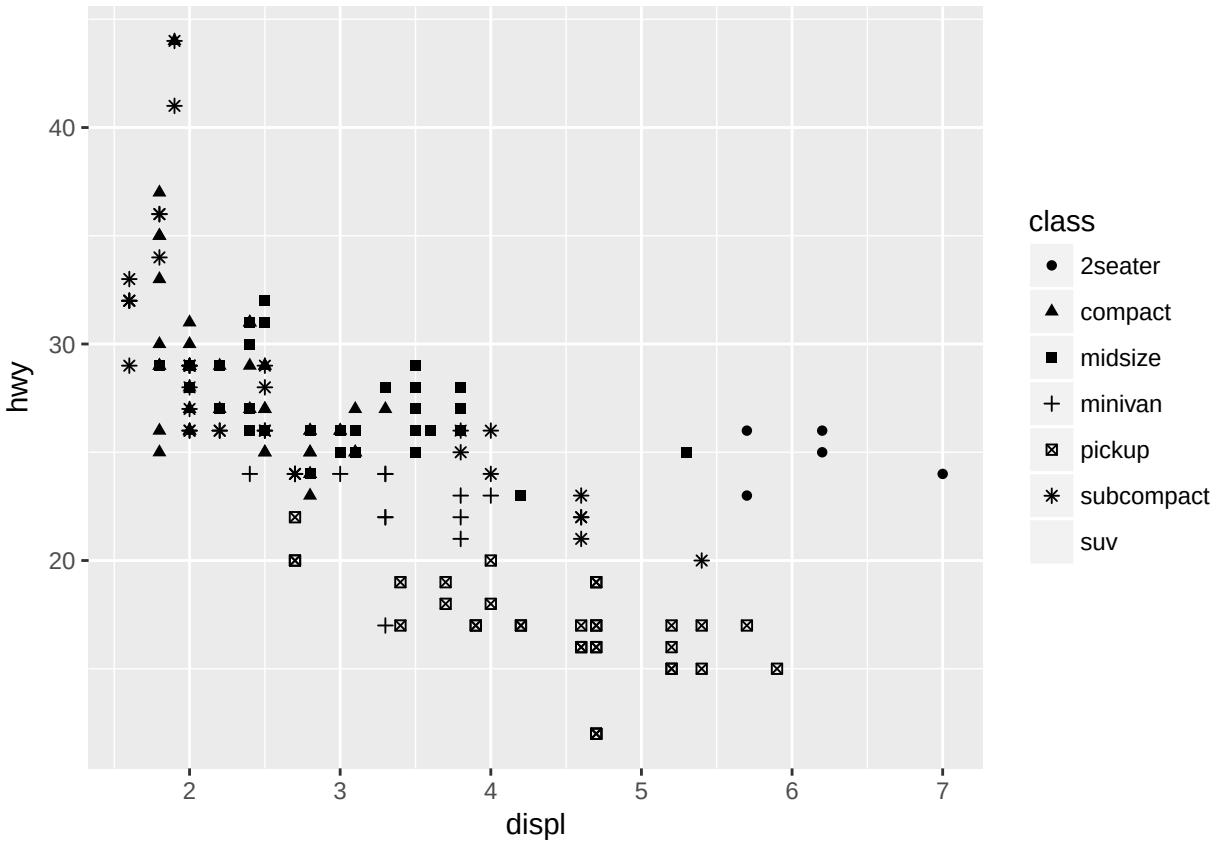
```
## Warning: Using size for a discrete variable is not advised.
```




```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```

```
## Warning: The shape palette can deal with a maximum of 6 discrete values
## because more than 6 becomes difficult to discriminate; you have 7.
## Consider specifying shapes manually if you must have them.
```

```
## Warning: Removed 62 rows containing missing values (geom_point).
```



When **maped to colour**: each category from `class` variable is representedy by a different colour.

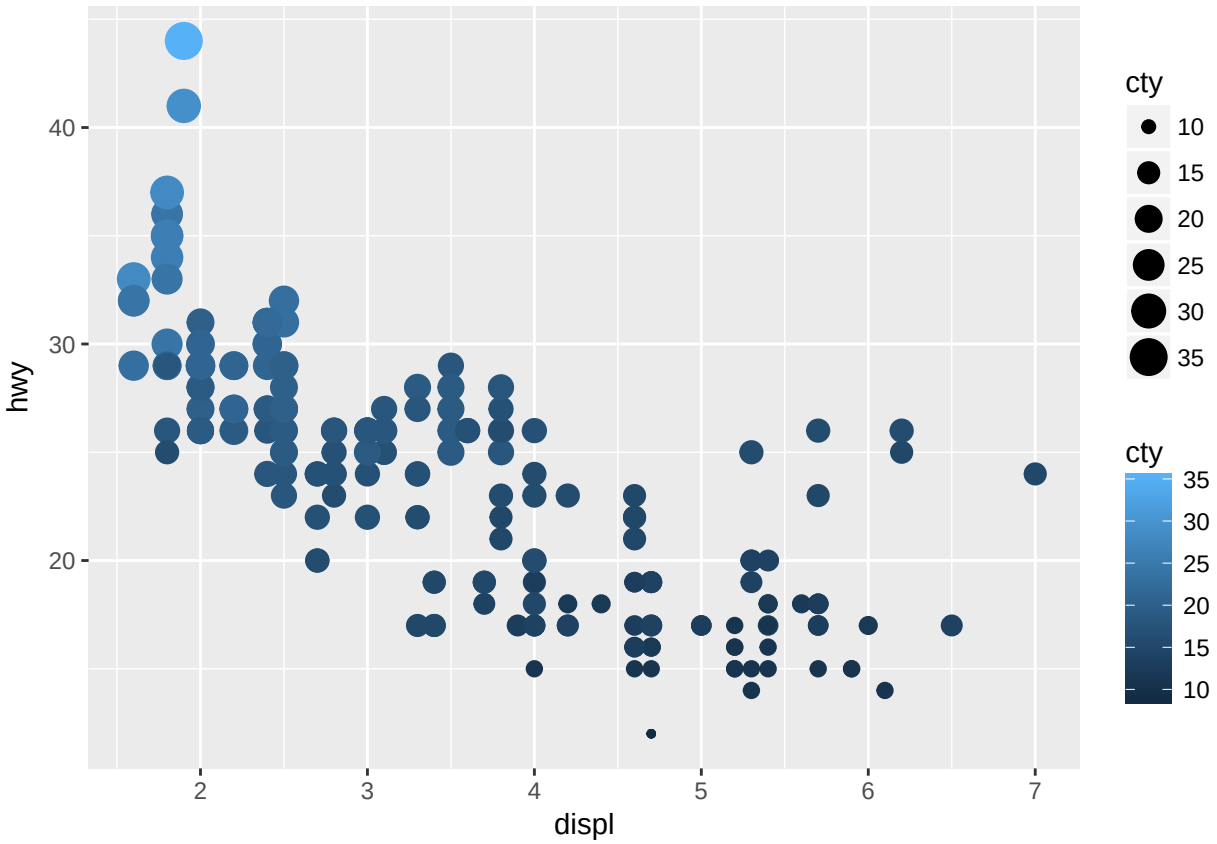
When **mapped to size**: each category from `class` variable is represented by a diffrenet size. This aesthetic with a categorical variable is a **bad idea**.

When **mapped to shape**: each category from `class` variable is represented by a different shape. For categorical variables with more than 6 categories, this aesthetics is not a good idea, since there is only 6 different shapes.

Exercise 4

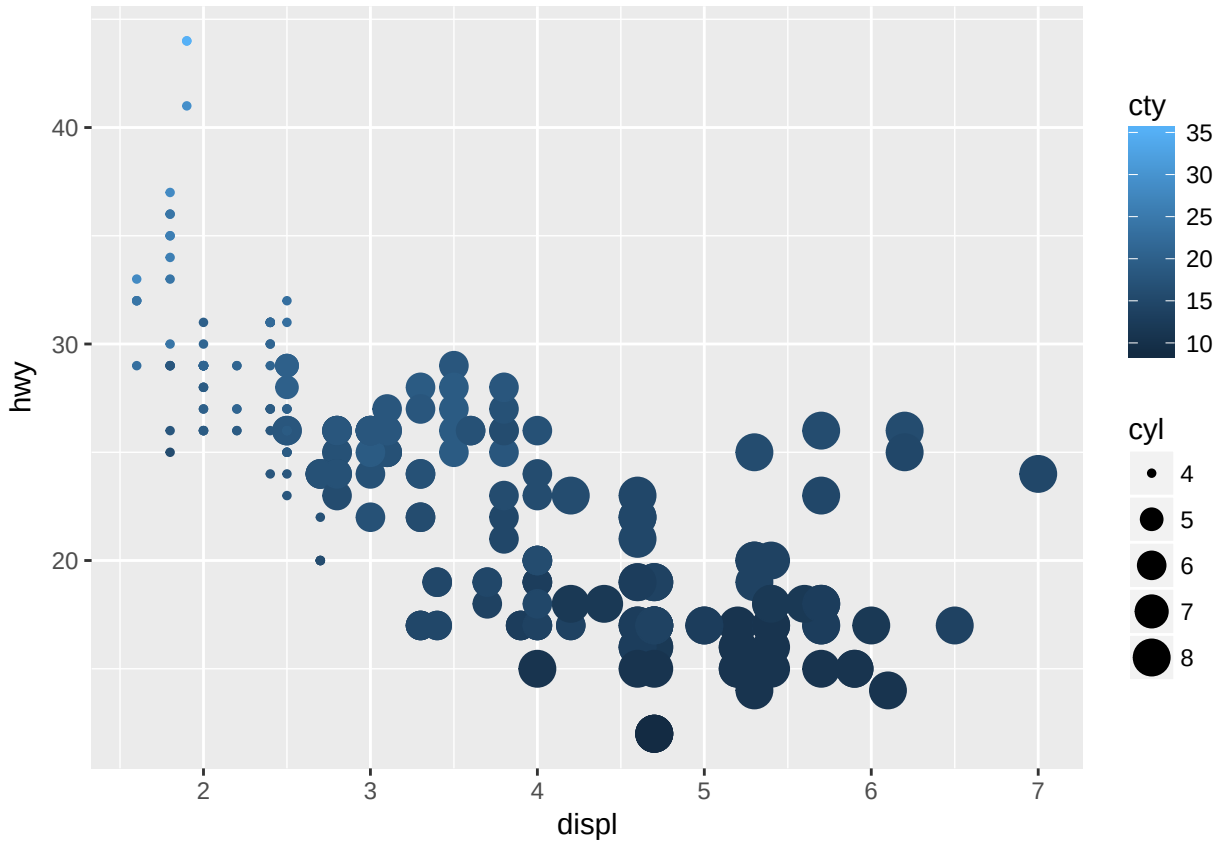
What happens if you map the same variable to multiple aesthetics?

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, colour = cty, size = cty))
```



Mapping one variable to multiple aesthetics is not a good idea because is redundant (however, the plot looks pretty cool). Using different variables and the plot will show more information about your dataset. The next plot uses four different variables in aesthetics, which gives useful aditional information when compared to all the previous plots.

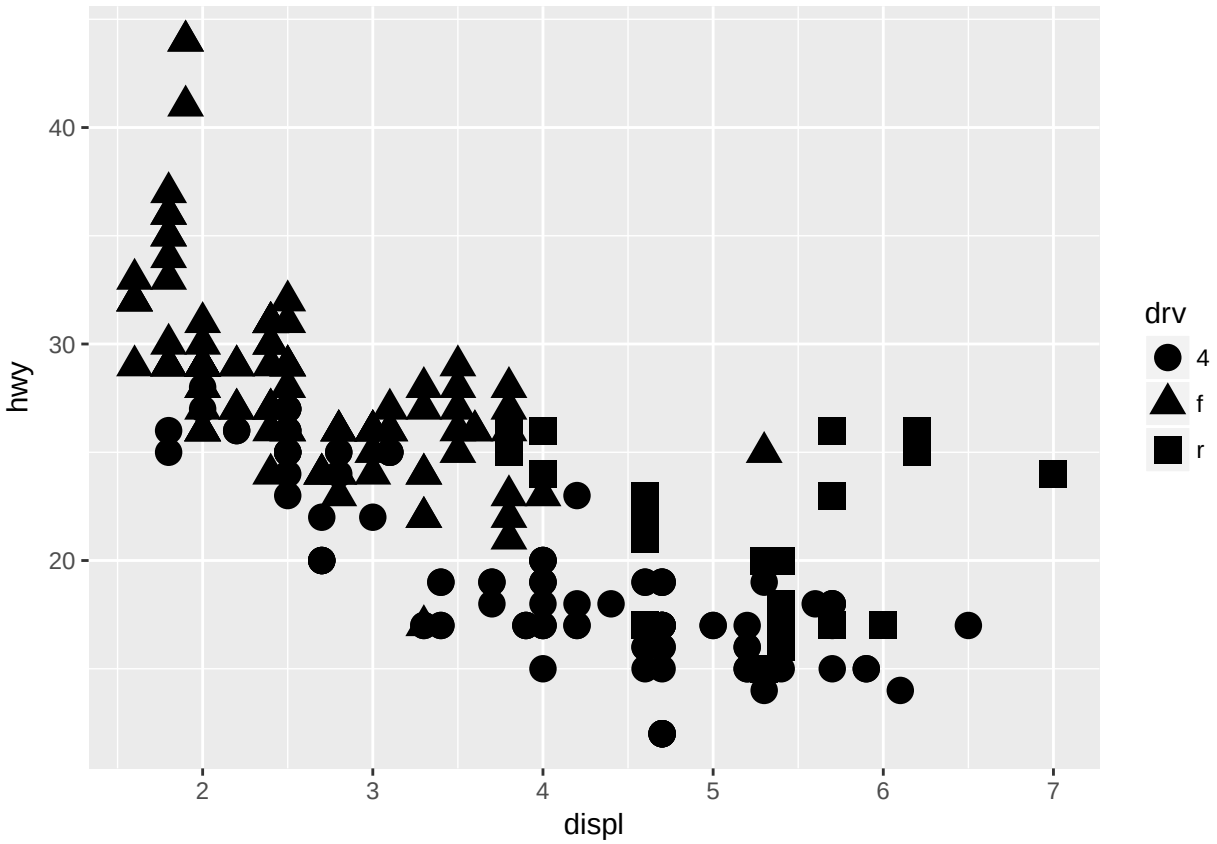
```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, colour = cty, size = cyl))
```



Exercise 5

What does the `stroke` aesthetic do? What shapes does it work with? (Hint: use `?geom_point` and check the 'help' tab)

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, shape = drv), stroke = 5)
```



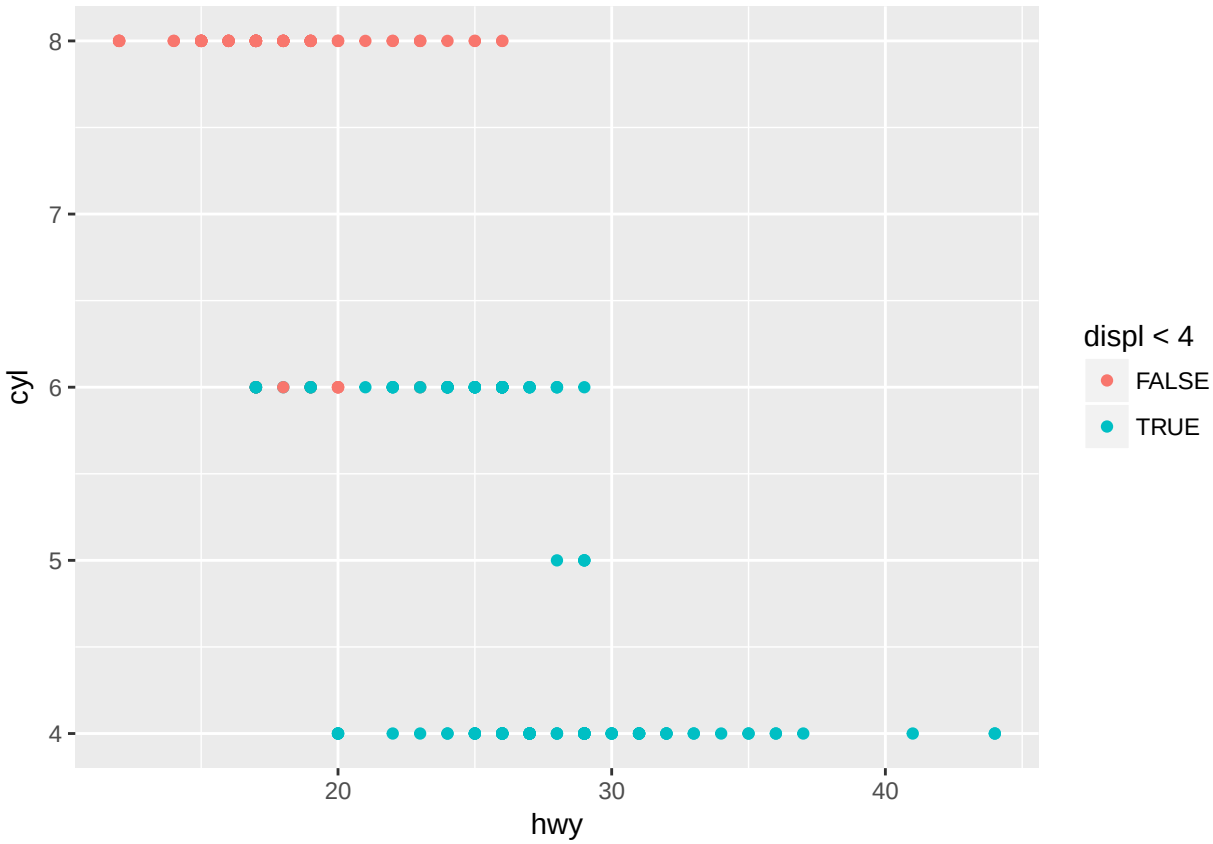
(Answer) The `stroke` aesthetic is used to modify the width of the border.

Exercise 6

> What happens if you map an aesthetic to something other than a variable name, like `aes(colour = displ < 5)`?

(Answer) The colour indicates if each `displ` value is less than 4 or not. The ggpplot function will assign the result of this expression (displ < 5, which is going to be true or false) to a temporary variable and then will assign a colour for values > 5 and a different colour for values < 5). This is easy visualize by checking the results of this code:

```
ggplot(data = mpg) + geom_point(mapping = aes(x = hwy, y = cyl, colour = displ < 4))
```



3.5.1 Exercises

Exercise 1

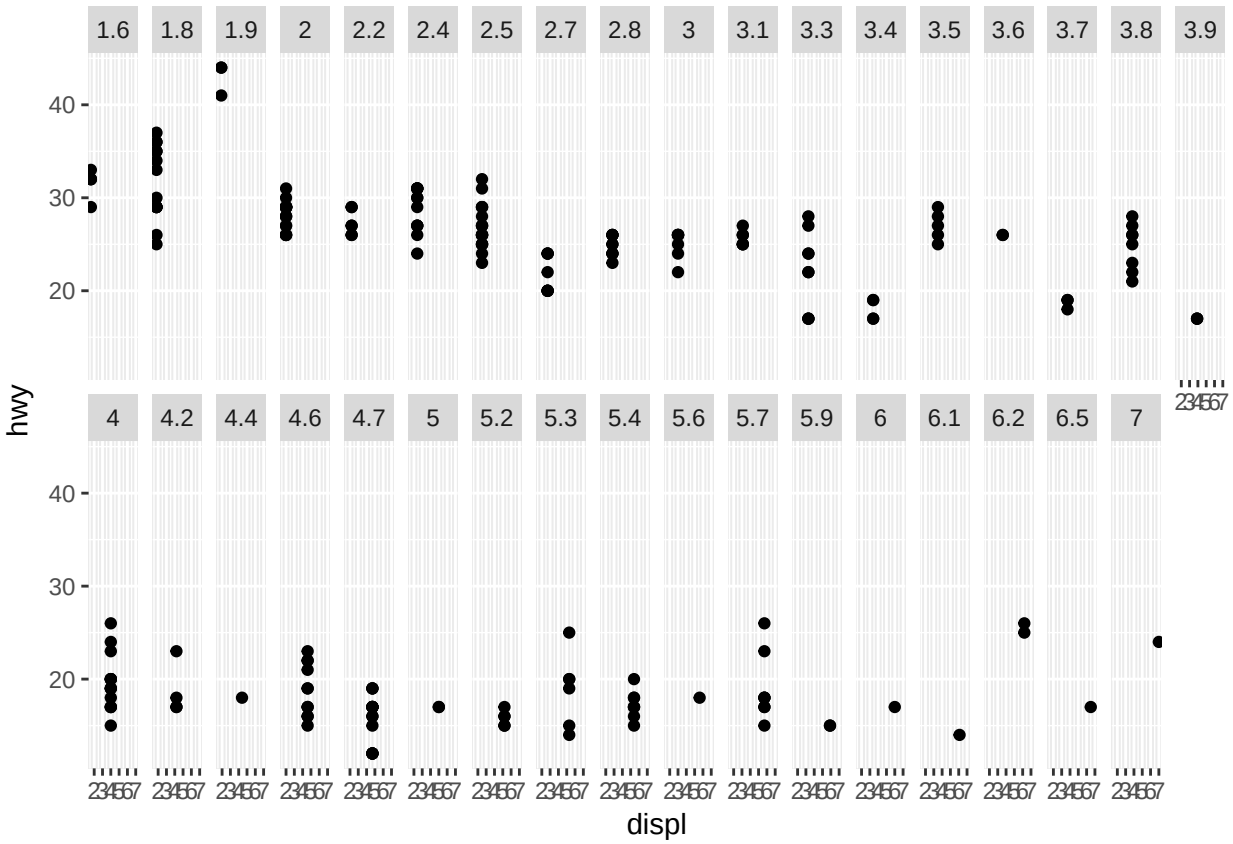
What happens if you facet on a continuous variable?

(Answer) To remember the variables classification:

Continuous	Categorical
displ	model
year	trans
cyl	drv
cty	fl
hwy	class

Let's plot and see what happens!

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ displ, nrow = 2)
```

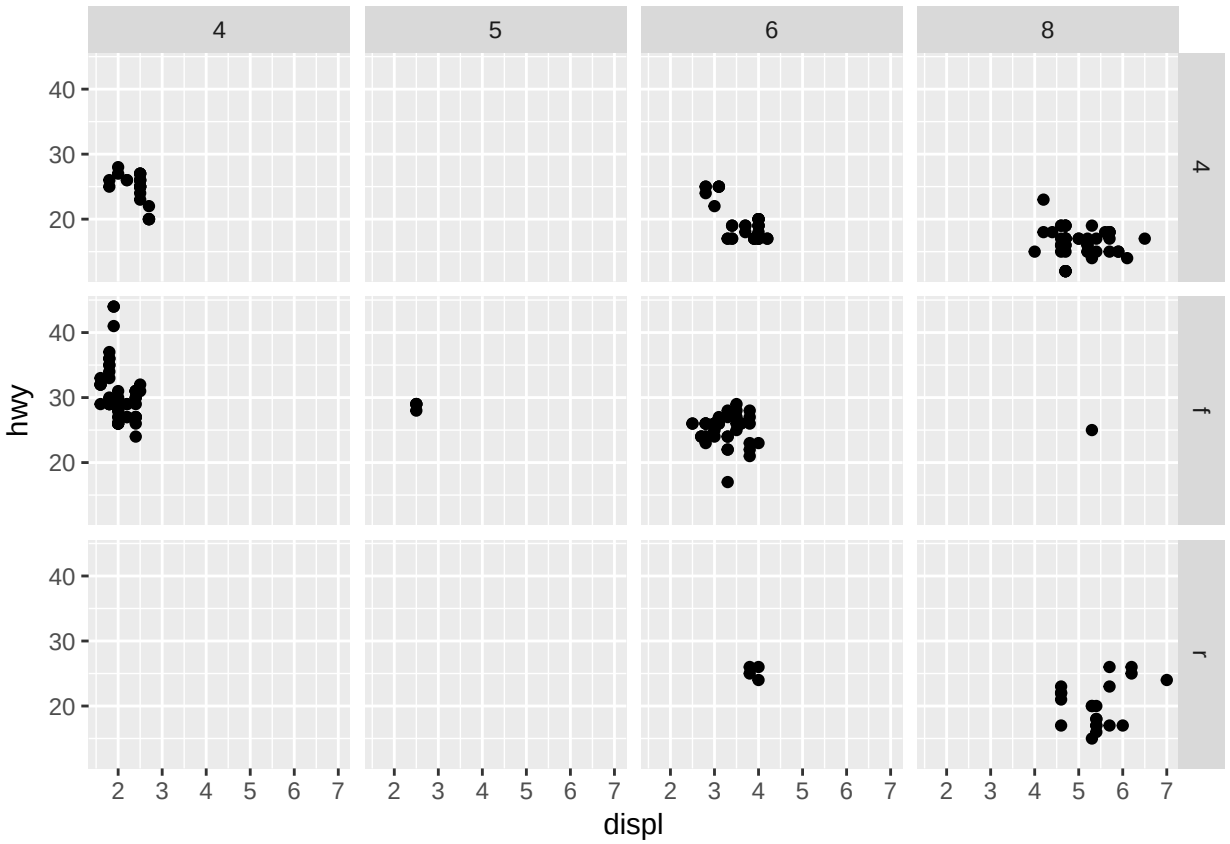


As you can see, it converts the continuous variable to a factor and then creates facets for all unique values of it. Facets is particularly useful for **categorical** variables.

Exercise 2

What do the empty cells in plot with **facet_grid(drv ~ cyl)** mean? How do they relate to this plot?

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ cyl)
```



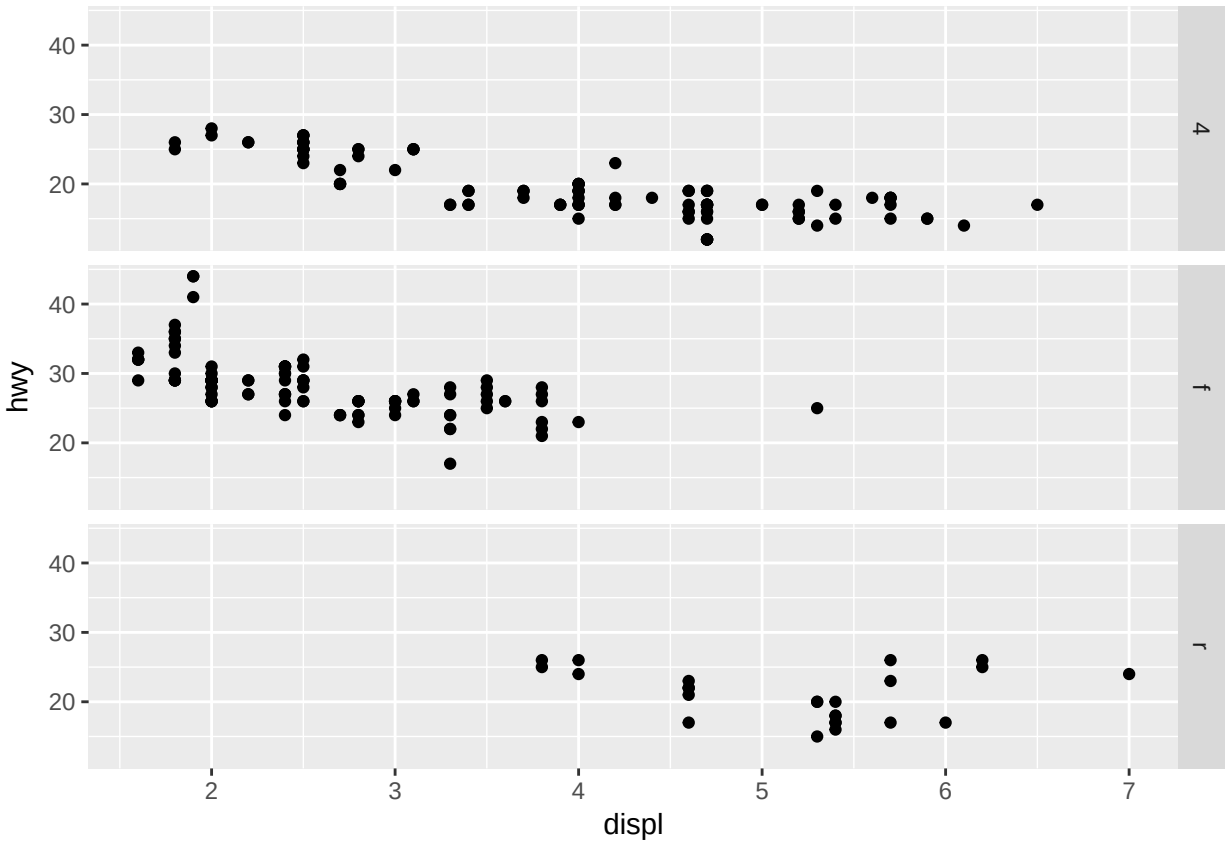
(Answer) The empty cells means that there are no values for the combination of `drv` and `cyl`. In this case, there are no cars which the traction control system is 4wd and the number of cylinders is 5, for example (you can check the same for the two others empty cells).

Exercise 3

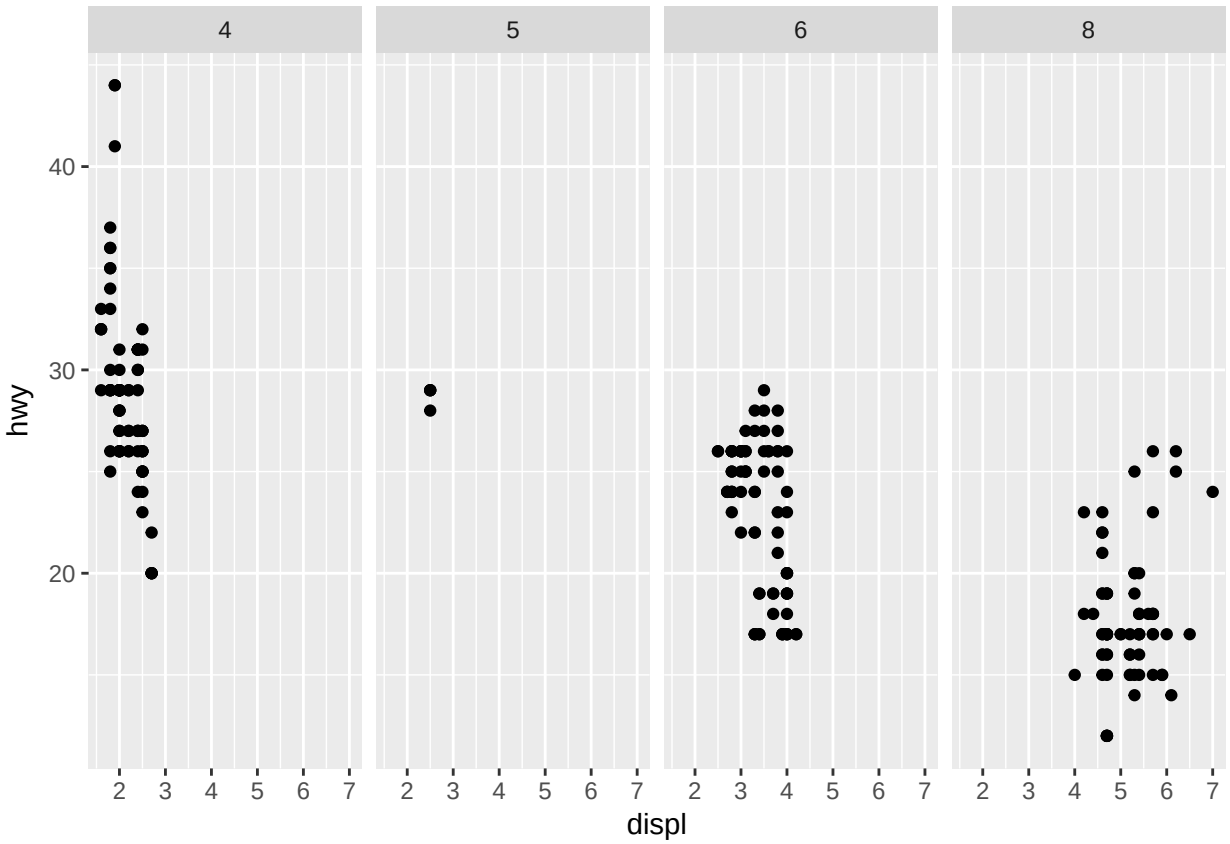
What plots does the following code make? What does `.` do?

Let's see!

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ .)
```




```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(. ~ cyl)
```

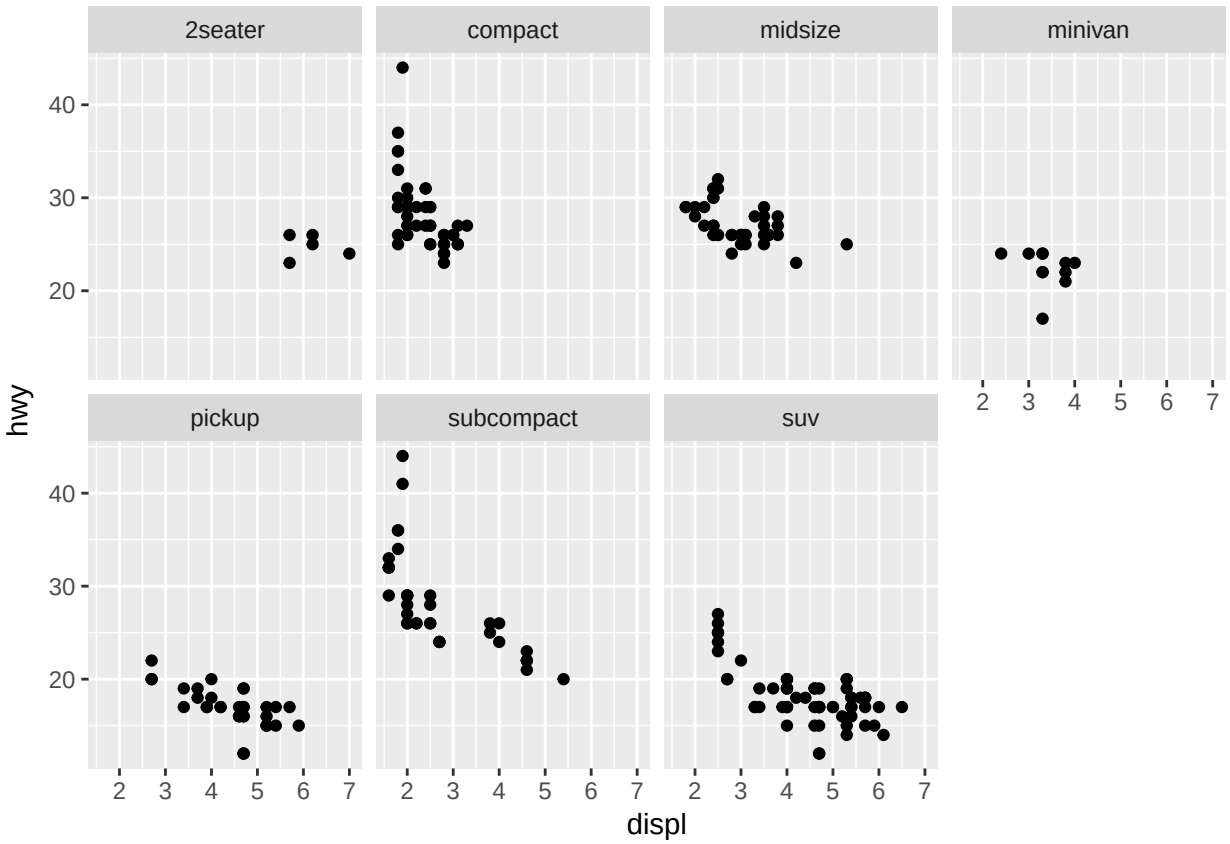


(Answer) As you can see in these two plots, `.` ignores a dimension for faceting (x or y axis).

Exercise 4

Take the first faceted plot in this section:

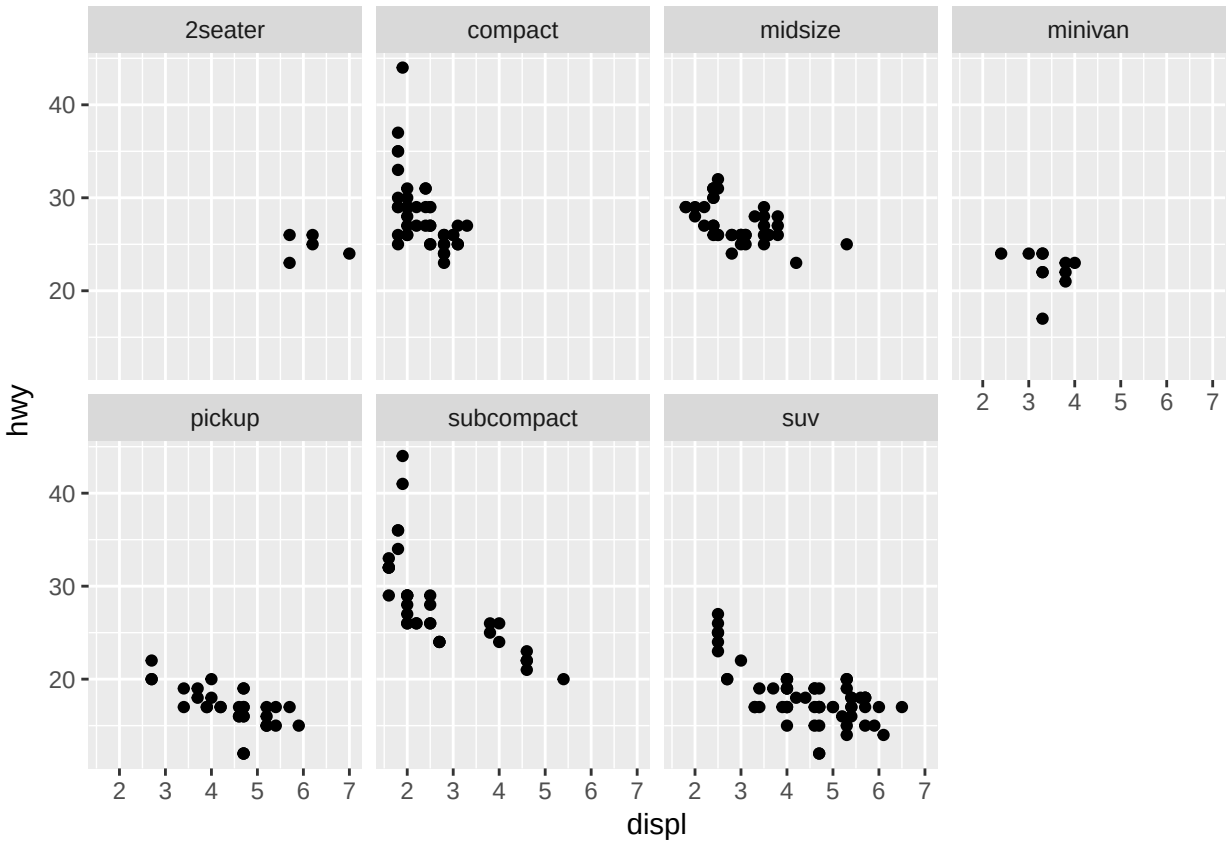
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```



What are the advantages to using faceting instead of the colour aesthetic? What are the disadvantages? How might the balance change if you had a larger dataset?

Let's run this code (again):

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```



(Answer) Using faceting could be a great option to visualize you data if the number of classes (categorical variable) is not so large because faceting permits to visualize each category separated and maybe this can show the information about your dataset better than when using colour aesthetic (if you want to see the results for each category or a set of categories, for example). However, for larger datasets we might face with a categorical variable with many possible results and for this situation is better to visualize the data using colour aesthetic. The function you use depends on your dataset.

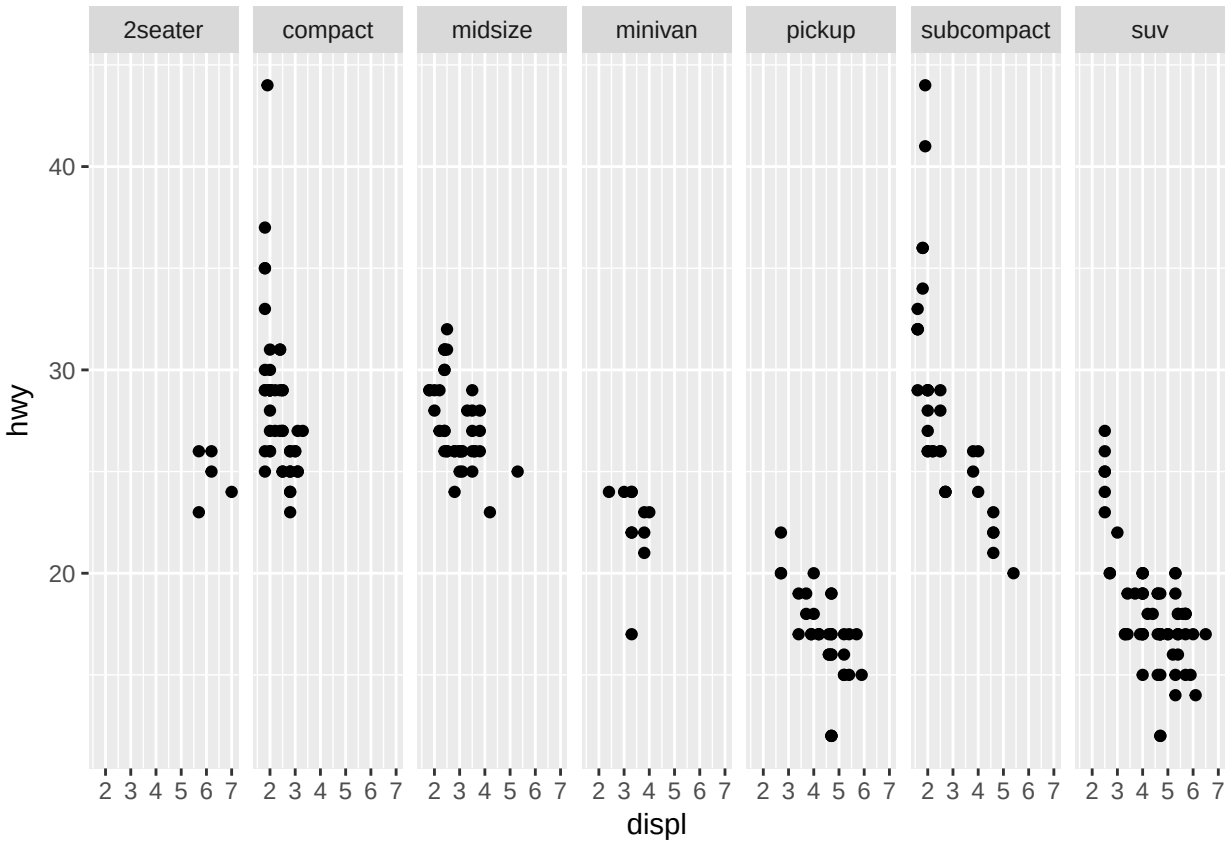
Exercise 5

> Read `?facet_wrap`. What does `nrow` do? What does `ncol` do? What other options control the layout of the individual panels? Why doesn't `facet_grid()` have `nrow` and `ncol` argument?

(Answer) `nrow` and `ncol` define the number of rows and columns and this is necessary since facet_wrap only facets on one variable. You also can change the layout of the individual panels with `scales`, `switch`, `as.table` or `dir`, for example.

Let's see what happens when we set `ncol = 7`, which is the number of different car classes (`class`).

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, ncol = 7)
```



Really cool, right?

On the other hand, `nrow` and `ncol` are not necessary in `face_grid()` because the number of rows and columns are already determined depending only on the variables that were chosen (number of unique values of the variables used).

Exercise 6

> When using `facet_grid()` you should usually put the variable with more unique levels in the columns. Why?

(Answer) This is usually used in this way just to be easier to visualize. Is better to see the plot larger horizontally than vertically. So, using the variable with more unique levels in the columns the plot will grow horizontally. On the other hand, if this variable is used in the rows, the plot will grow vertically and for humans, usually this is worse to visualize.

3.6.1 Exercises

Exercise 1

> What geom would you use to draw a line chart? A boxplot? A histogram? An area chart?

(Answer) Plot type - Geom you should use:

Plot type	Geom
Line chart	`geom_line()`

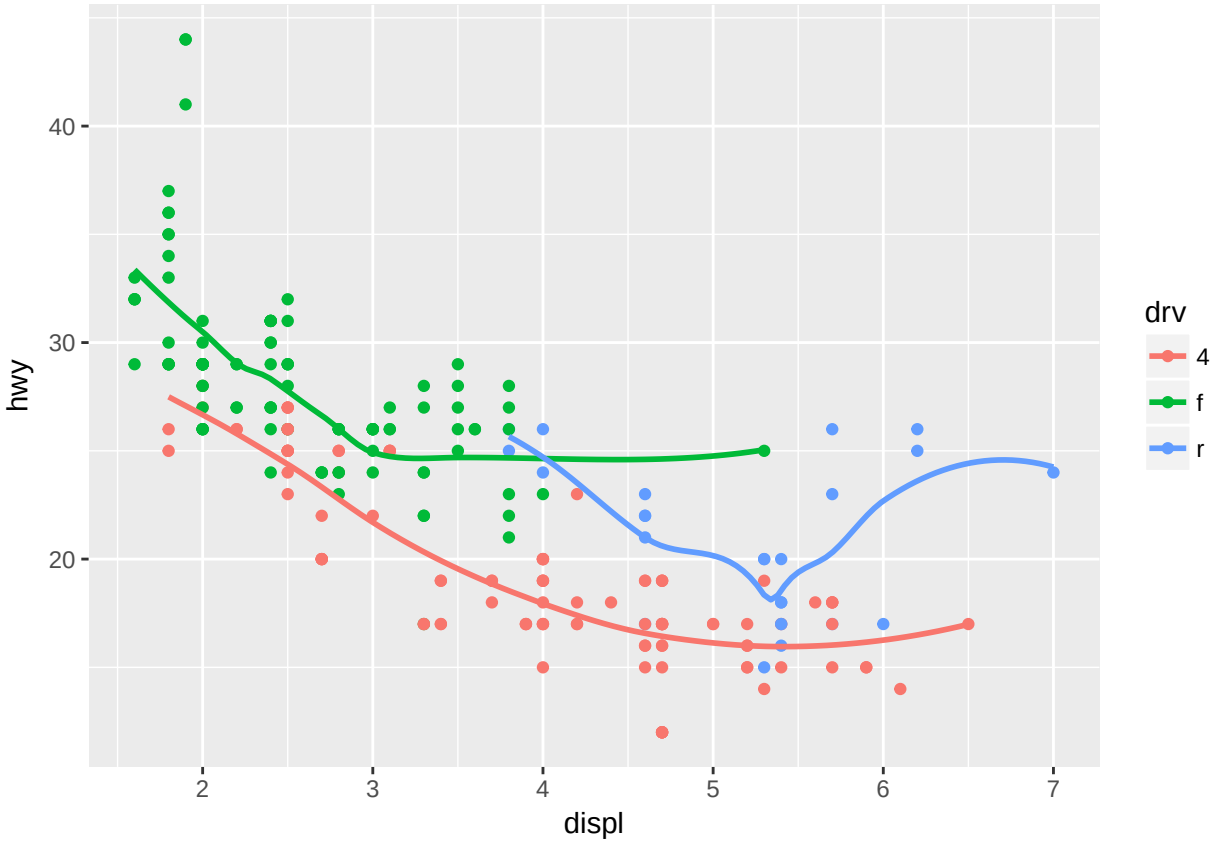
Plot type	Geom
Boxplot	`geom_boxplot()`
Histogram	`geom_hist()`
Area chart	`geom_area()`

Exercise 2

Run this code in your head and predict what the output will look like. Then, run the code in R and check your predictions.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  geom_smooth(se = FALSE)
```

```
## `geom_smooth()` using method = 'loess'
```



(Answer) As you can see in the previous plot, this code produces a scatter plot with `displ` on the x axis and `hwy` on the y axis and the points are coloured according to the `drv` variable. Also, there is a smooth line created with `geom_smooth` with the standard errors setted to false (`se = FALSE`) and fitted according to `drv`.

Exercise 3

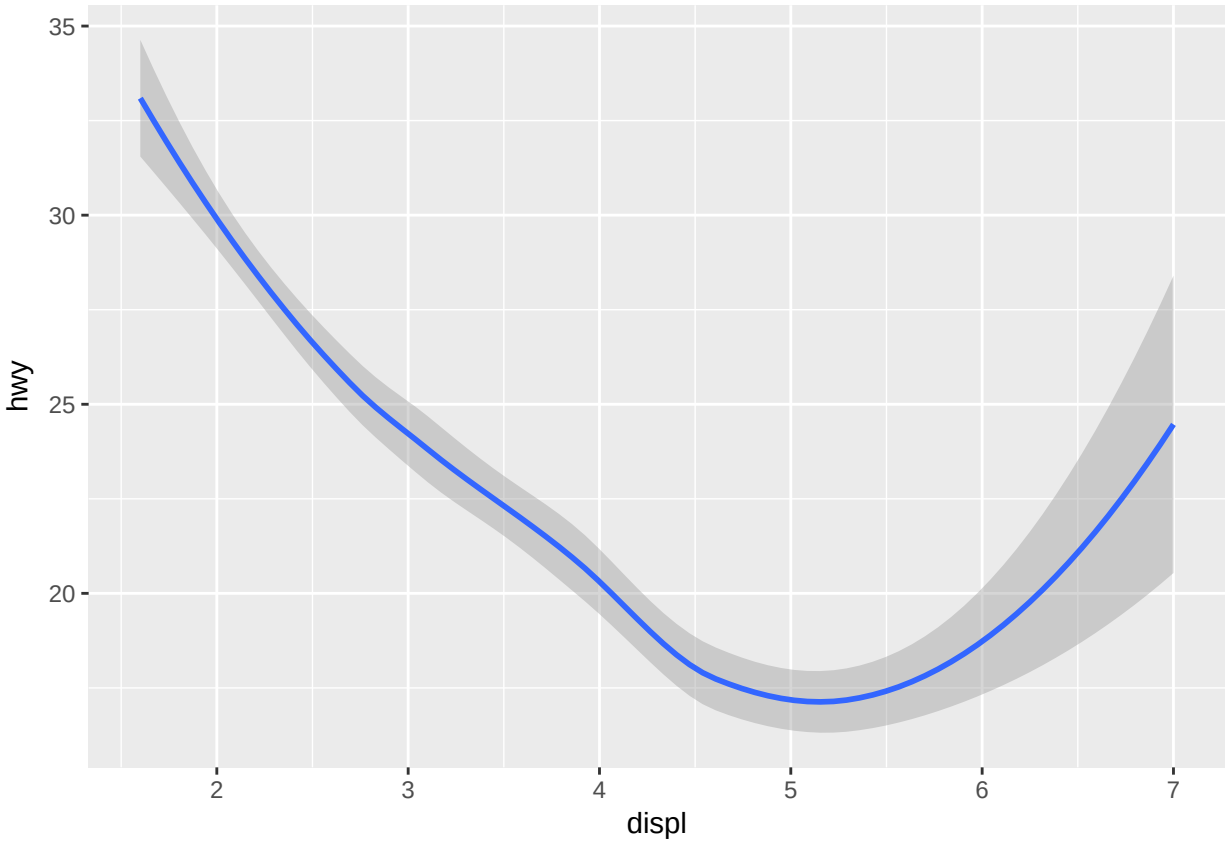
What does show.legend = FALSE do? What happens if you remove it? Why do you think I used it earlier in the chapter?

I am no sure if my answer is one hundred percent correct for the last question of this exercise.

(Answer) `show.legend = FALSE` hides the legend for the plot. If you do not specify this, the default value is going to be true (plot will show the legend box, if there is more than one category). The book used it earlier in this chapter to create these 3 plots:

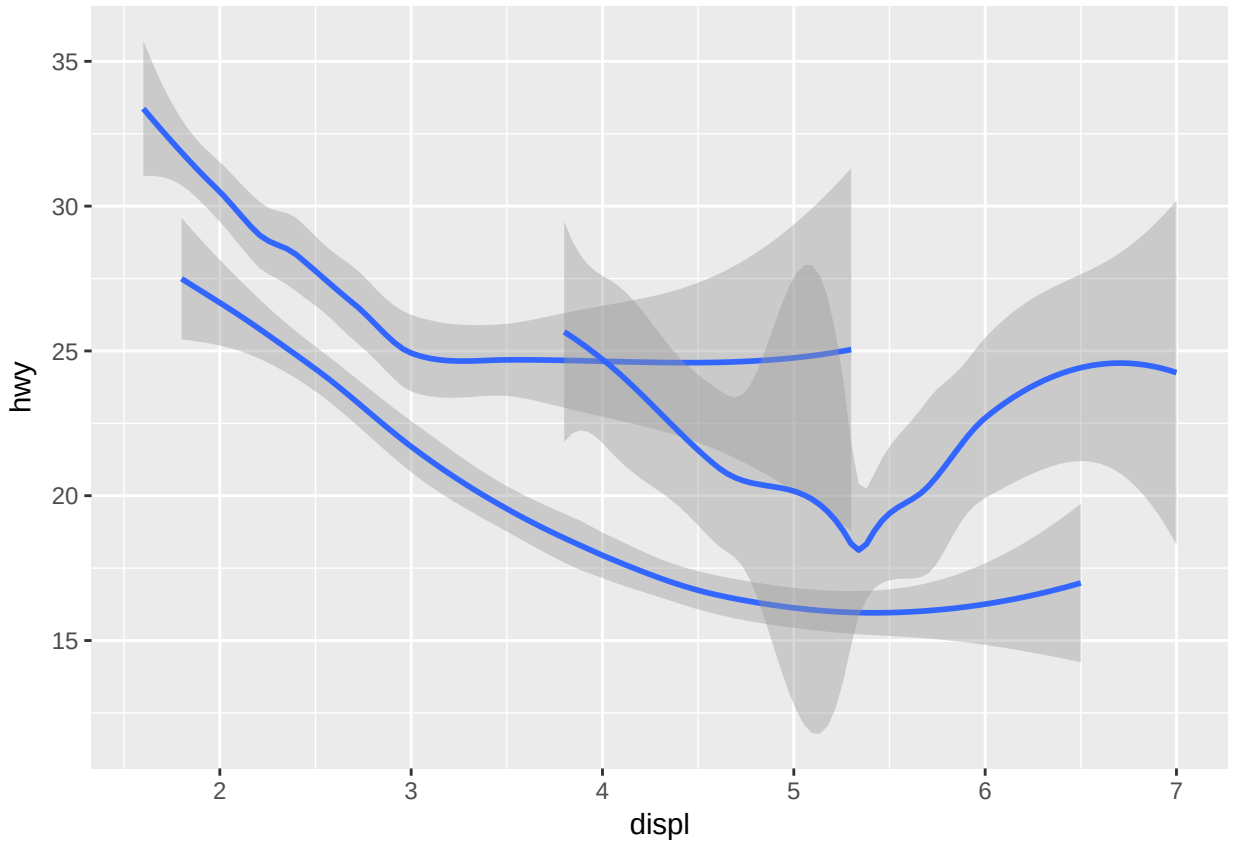
```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

```
## `geom_smooth()` using method = 'loess'
```



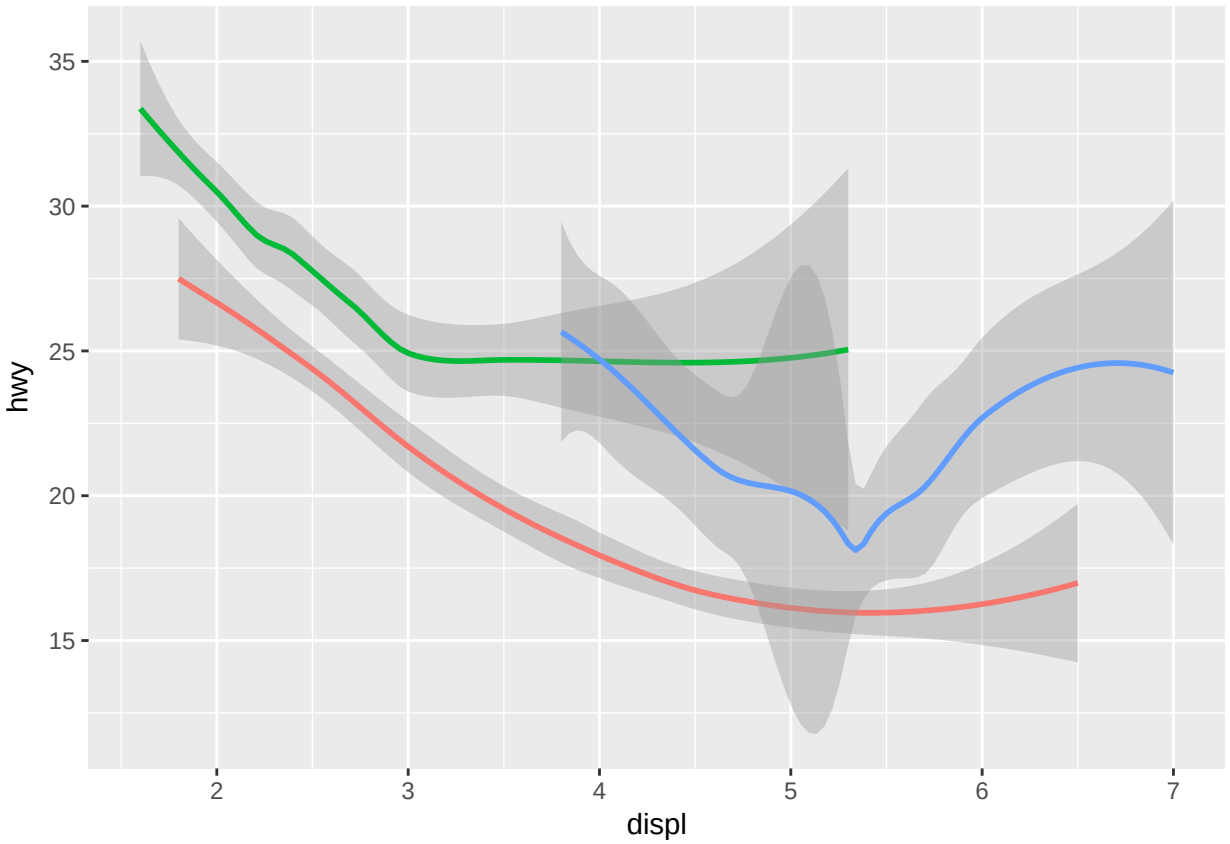

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, group = drv))
```

```
## `geom_smooth()` using method = 'loess'
```




```
ggplot(data = mpg) +
  geom_smooth(
    mapping = aes(x = displ, y = hwy, color = drv),
    show.legend = FALSE
  )
```

```
## `geom_smooth()` using method = 'loess'
```



In this case, a legend just in the last plot is not a good idea because in the two first plots there is no legend for the plot. The legend would make a irregular presentation and would show a irrelevant information (out of the scope of the goal that these 3 plots have).

Exercise 4

What does the `se` argument to `geom_smooth()` do?

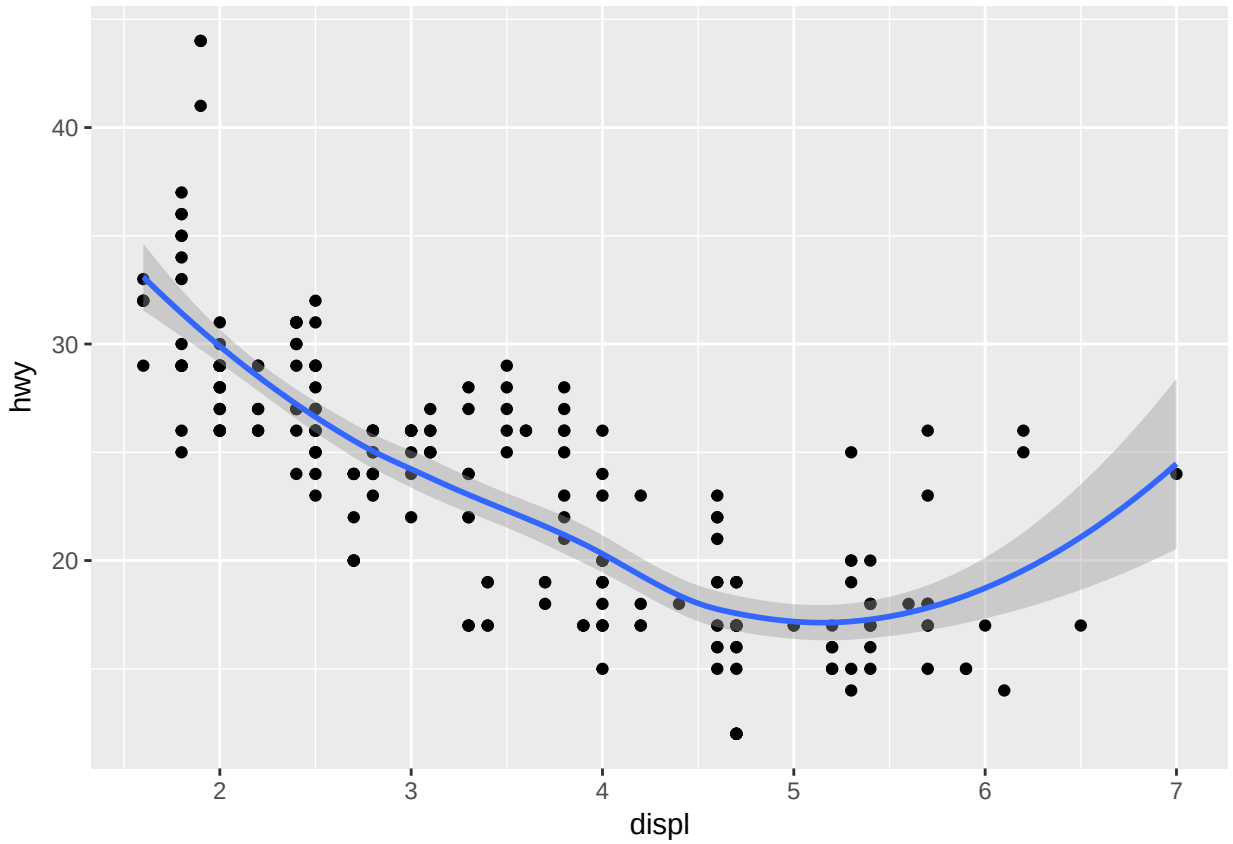
The answer for this question is inside the exercise 2 answer. The `se` argument used in `geom_smooth()` is used to specify if you want to plot with the standard errors (default or set `se = TRUE`) or not (`se = FALSE`). In the plot, the standard error is the 'grey shadow'.

Exercise 5

Will these two graphs look different? Why/why not?

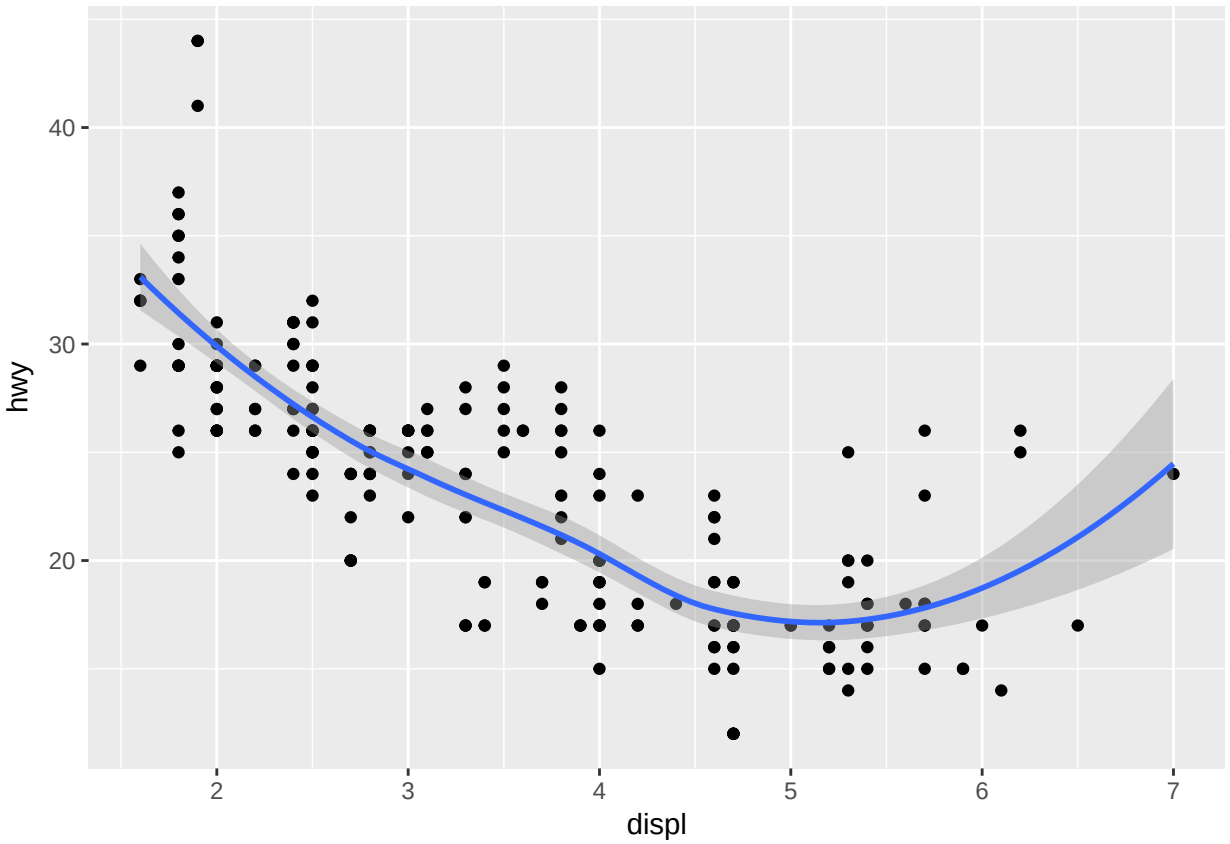
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'loess'
```




```
ggplot() +
  geom_point(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_smooth(data = mpg, mapping = aes(x = displ, y = hwy))
```

```
## `geom_smooth()` using method = 'loess'
```



(Answer) As you can see, these two codes produce idendical plots. The first code specifies the data and mapping inside `ggplot()` function, which will automatically be used by geoms functions (in this case, `geom_point()` and `geom_smooth()`). In the second code, the data and mapping definition are specified in both geoms (duplicated code, which is bad even if works).

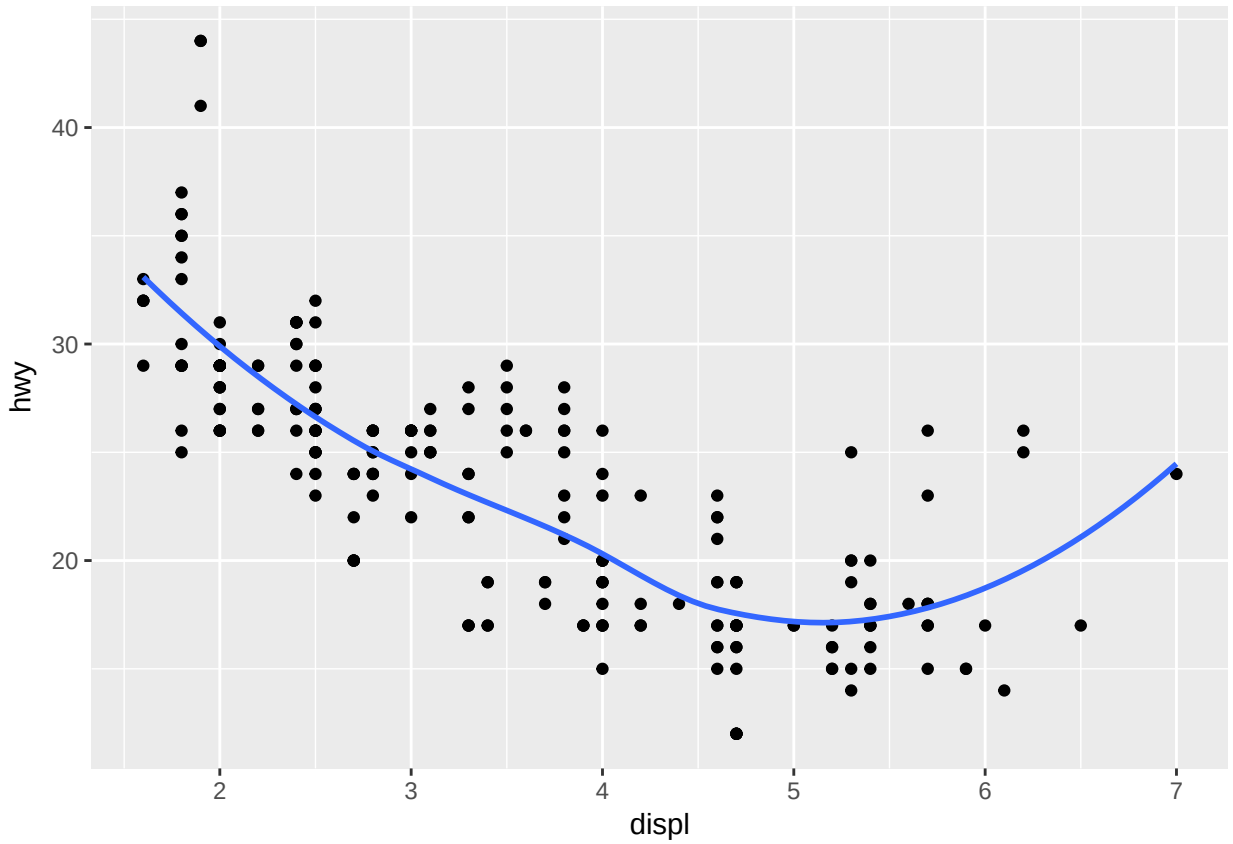
Exercise 6

Recreate the R code necessary to generate the following (6) graphs.

(Answer)

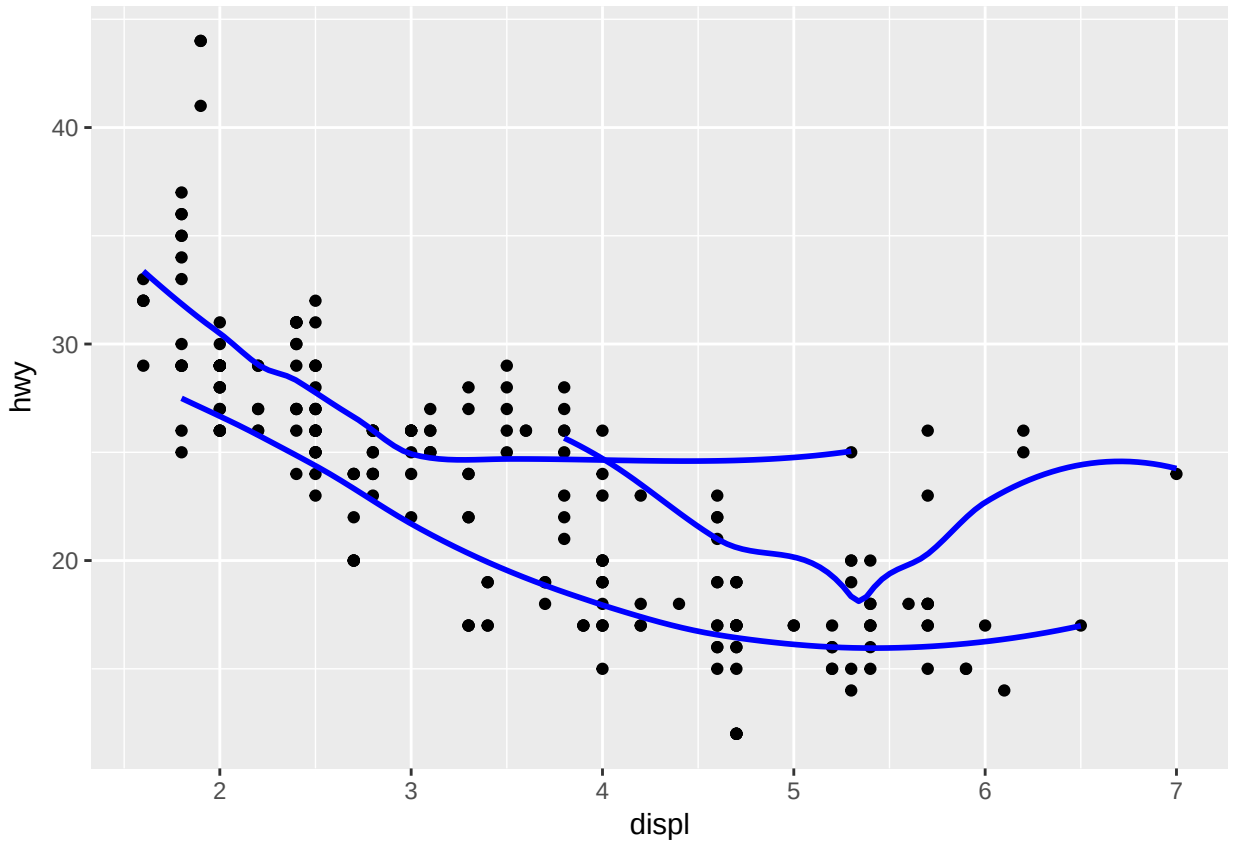
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
```

```
## `geom_smooth()` using method = 'loess'
```



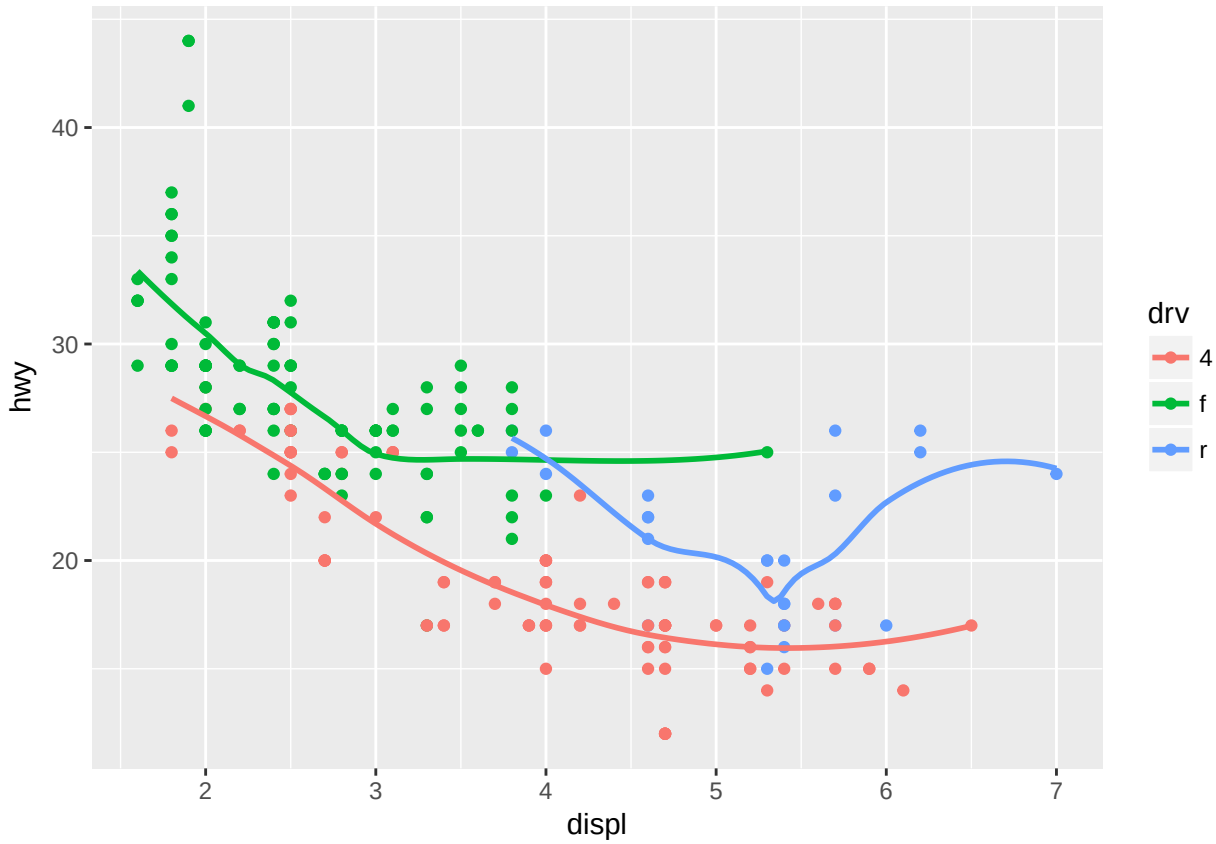

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(mapping = aes(group = drv), color = 'blue', se = FALSE)
```

```
## `geom_smooth()` using method = 'loess'
```



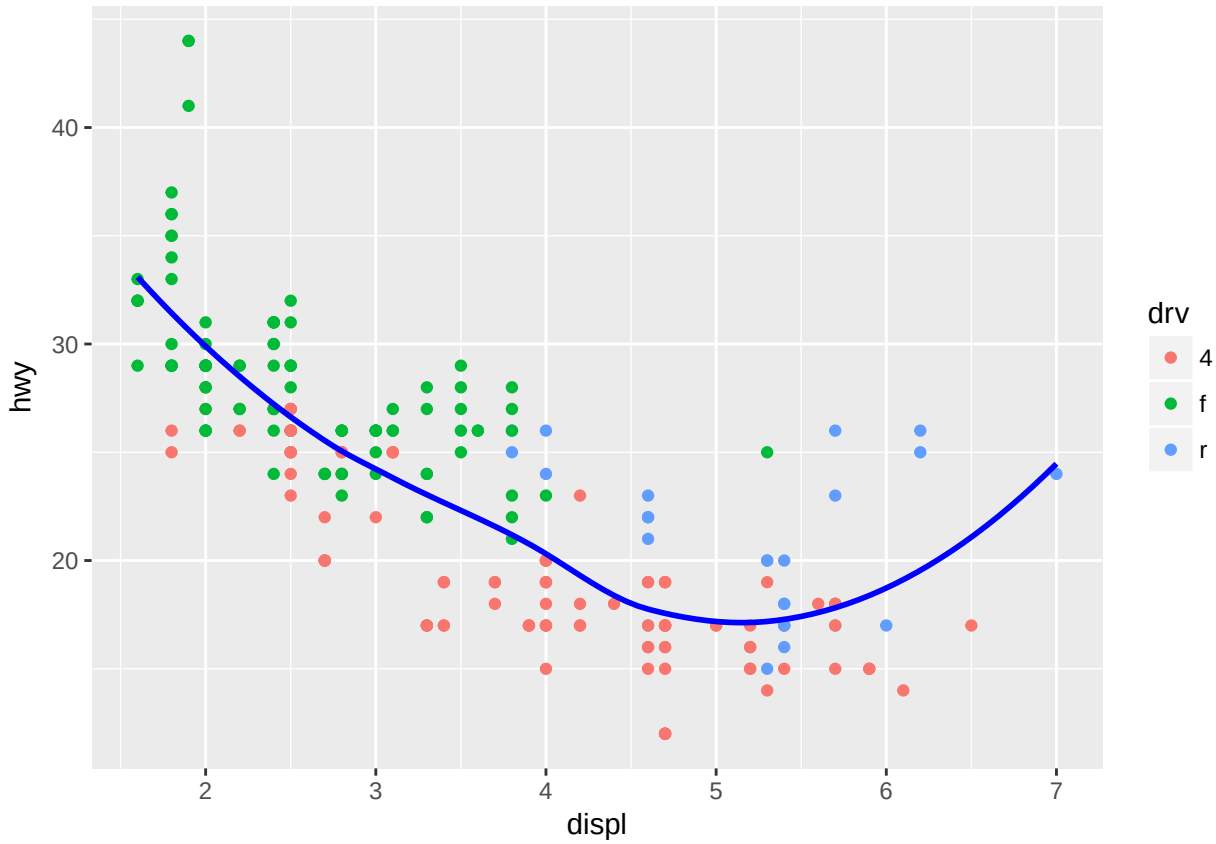

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(colour = drv)) +
  geom_smooth(mapping = aes(group = drv, colour = drv), se = FALSE)
```

```
## `geom_smooth()` using method = 'loess'
```



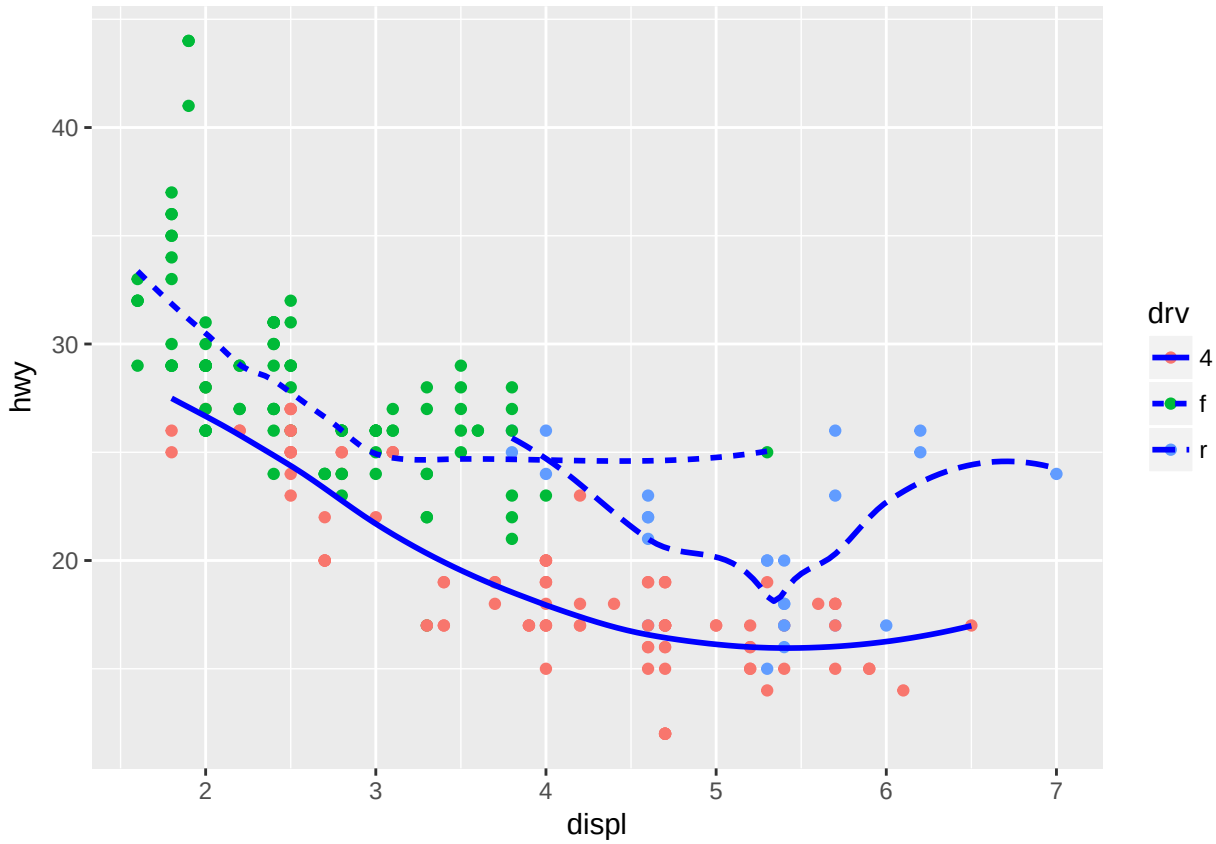

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(colour = drv)) +
  geom_smooth(colour = 'blue', se = FALSE)
```

```
## `geom_smooth()` using method = 'loess'
```

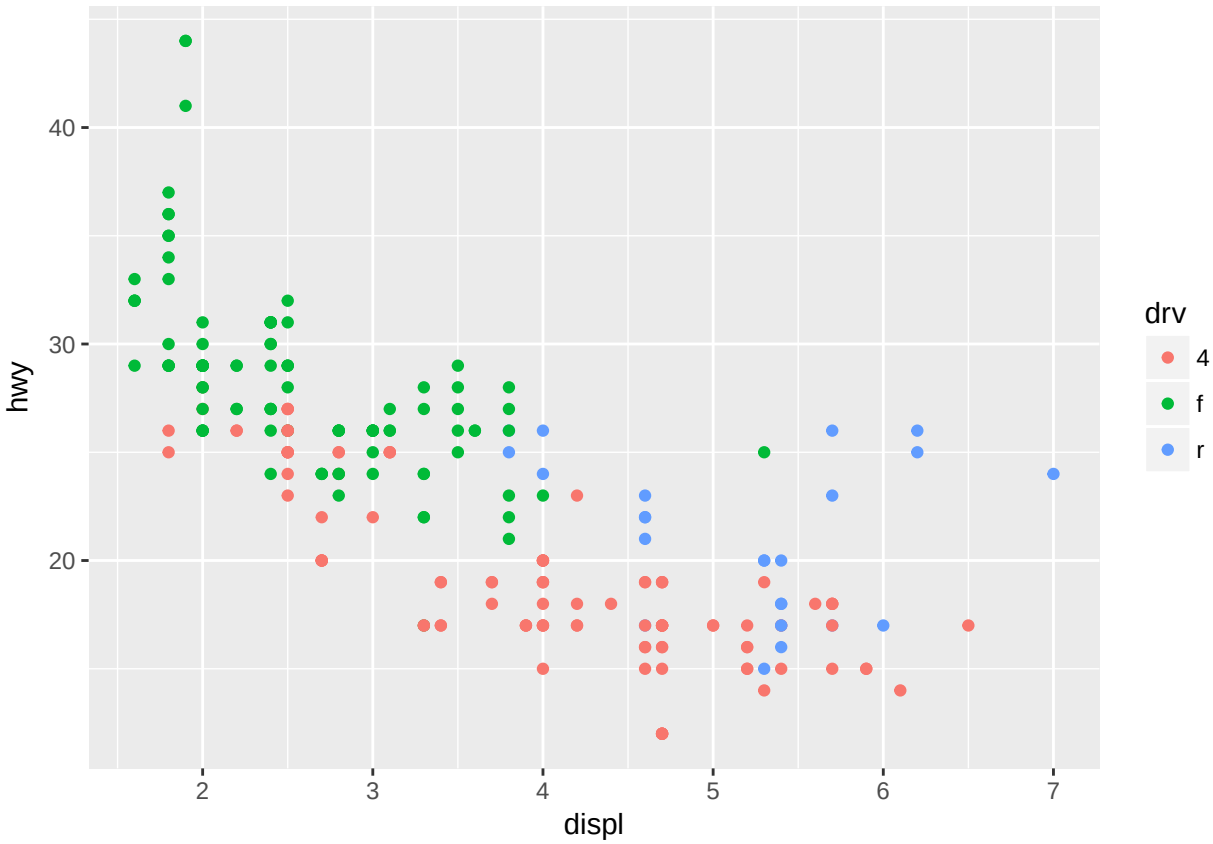



```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(colour = drv)) +
  geom_smooth(mapping = aes(linetype = drv), colour = 'blue', se = FALSE)
```

```
## `geom_smooth()` using method = 'loess'
```




```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(colour = drv))
```



Easy peasy lemon squeezy.

3.7.1 Exercises

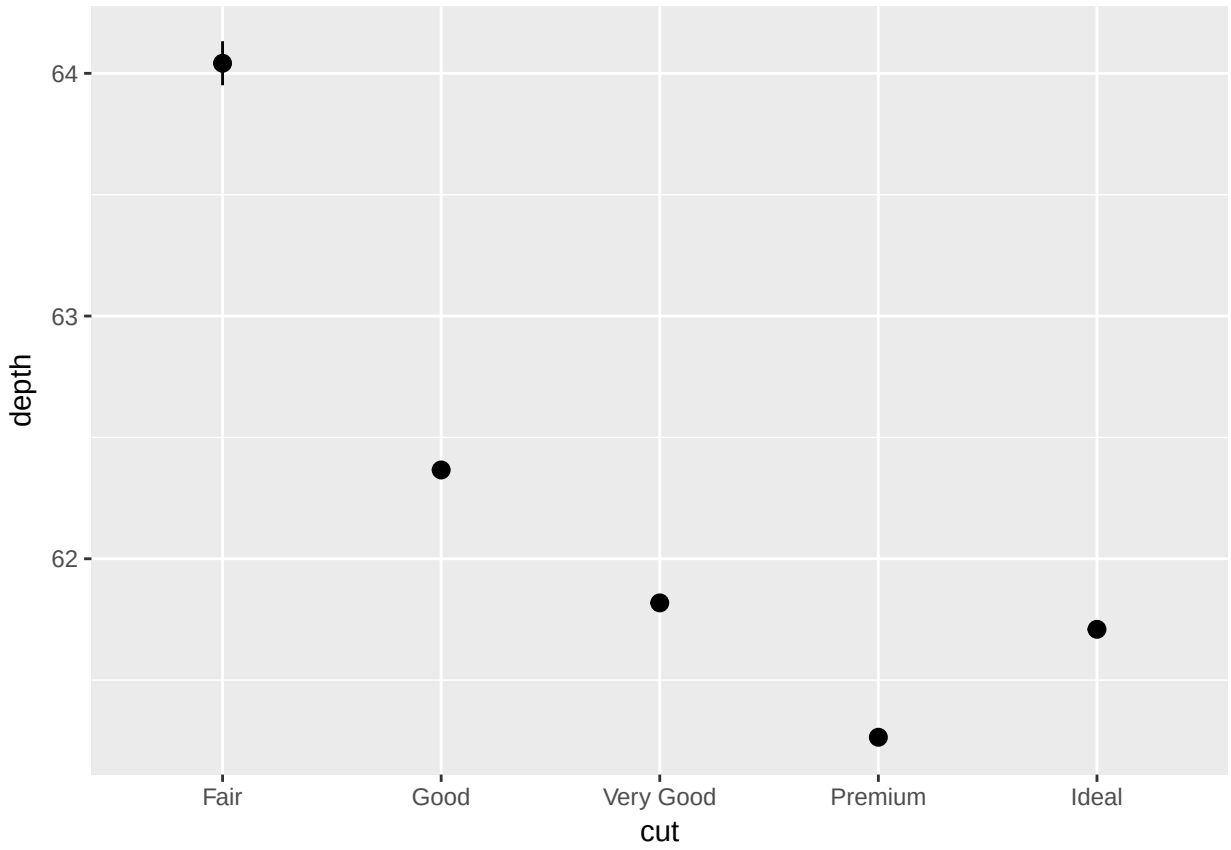
Exercise 1

> What is the default `geom` associated with `stat_summary()`? How could you rewrite the previous plot to use that `geom` function instead of the `stat` function?

(Answer) By typing `?stat_summary()` you are able to see the documentation for this function. So, is easy to notice that the default `geom` associated with `stat_summary()` is the `geom_pointrange()` geom, which uses `identity` as the dafault `stat`. To use this `geom` to plot a `summary`, just override the default stat by using `stat = 'summary'` as follows:

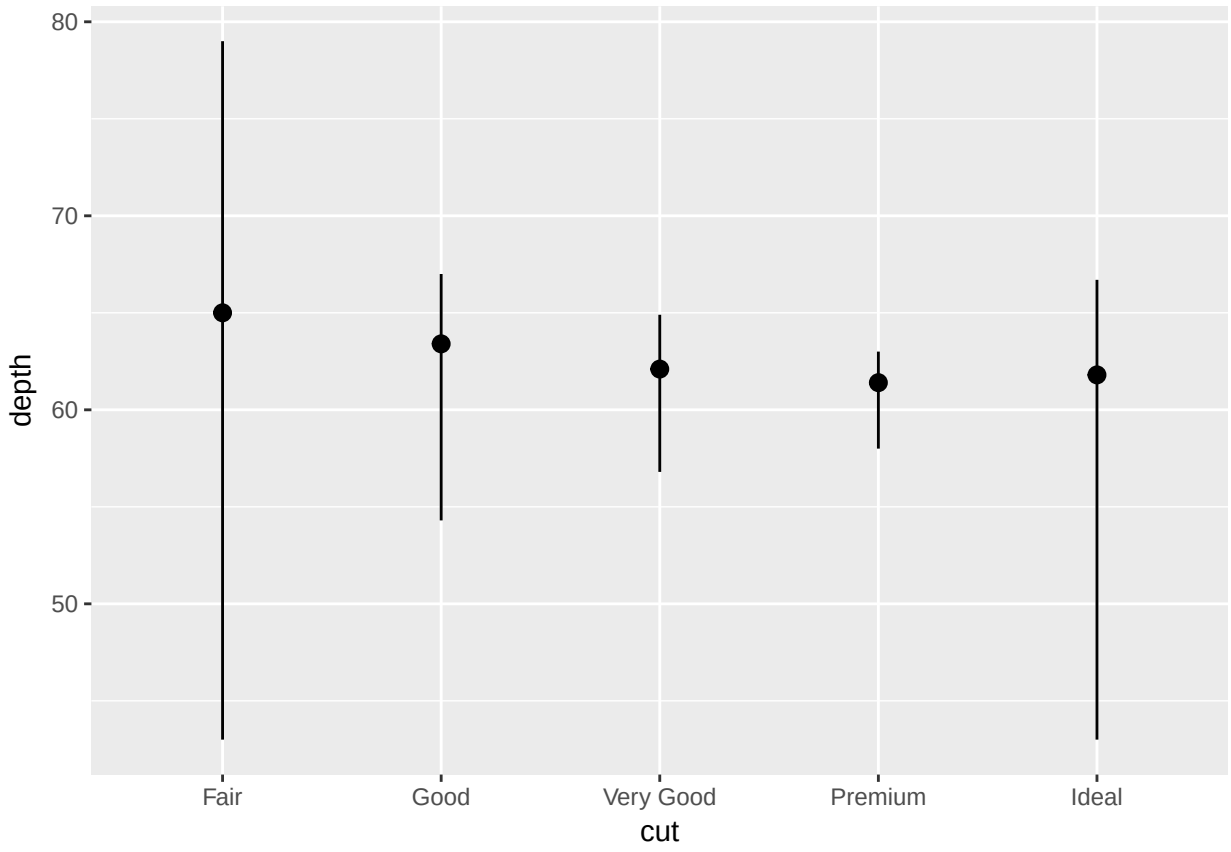
```
ggplot(data = diamonds) +
  geom_pointrange(
    mapping = aes(x = cut, y = depth),
    stat = 'summary'
  )
```

```
## No summary function supplied, defaulting to `mean_se()
```



Howerver, as you can notice that this last plot is a little bit different compared to the plot created with `stat_summary()`. It is because the default for `stat_summary()` is to use `mean` and `sd` to plt (the point and the range of the line). To fix this, just add the values used in the example (`fun.min = min`, `fun.max = max` and `fun.y = median`):

```
ggplot(data = diamonds) +
  geom_pointrange(
    mapping = aes(x = cut, y = depth),
    stat = 'summary',
    fun.ymin = min,
    fun.ymax = max,
    fun.y = median
  )
```



Voilà!

Exercise 2

What does `geom_col()` do? How is it different to `geom_bar()`?

(Answer) The answer for this question is inside `geom_col()` documentation. By typing `?geom_col()` - *I encourage you to always read the documentation for the function you want to use* - is possible to see there are two types of bar charts: `geom_bar()` makes the height of the bar proportional to the number of classes in each group (or if the `weight` aesthetic is supplied, the sum of the `weights`). If you want the heights of the bars to represent values in the data, use `geom_col()` instead. `geom_bar()` uses `stat_count` by default (it counts the number of cases at each `x` position). In other hand, `geom_col()` uses `stat_identity`, which leaves the data as is.

Exercise 3

Most `geoms` and `stats` come in pairs that are almost always used in concert. Read through the documentation and make a list of all the pairs. What do they have in common?

(Answer) The answer to this question is inside `ggplot2` documentation. I highly recommend to read the `ggplot2` documentation available **here**.

Exercise 4

What variables does `stat_smooth()` compute? What parameters control its behaviour?

(Answer) *This is the last time I am going to recommend you to always read the documentation for the functions you use.* The answer for this question is easy to find by checking `stat_smooth()` documentation. The variables computed by `stat_smooth()` are:

- `y`: predicted value
- `ymin`: lower pointwise confidence interval around the mean
- `ymax`: upper pointwise confidence interval around the mean
- `se`: standard error

And the arguments used to control its behaviour are:

- `mapping`
- `data`
- `position`
- `...`
- `method`
- `formula`
- `se`
- `na.rm`
- `show.legend`
- `inherit.aes`
- `geom, stat`
- `n`
- `span`
- `fullrange`
- `level`
- `method.args`

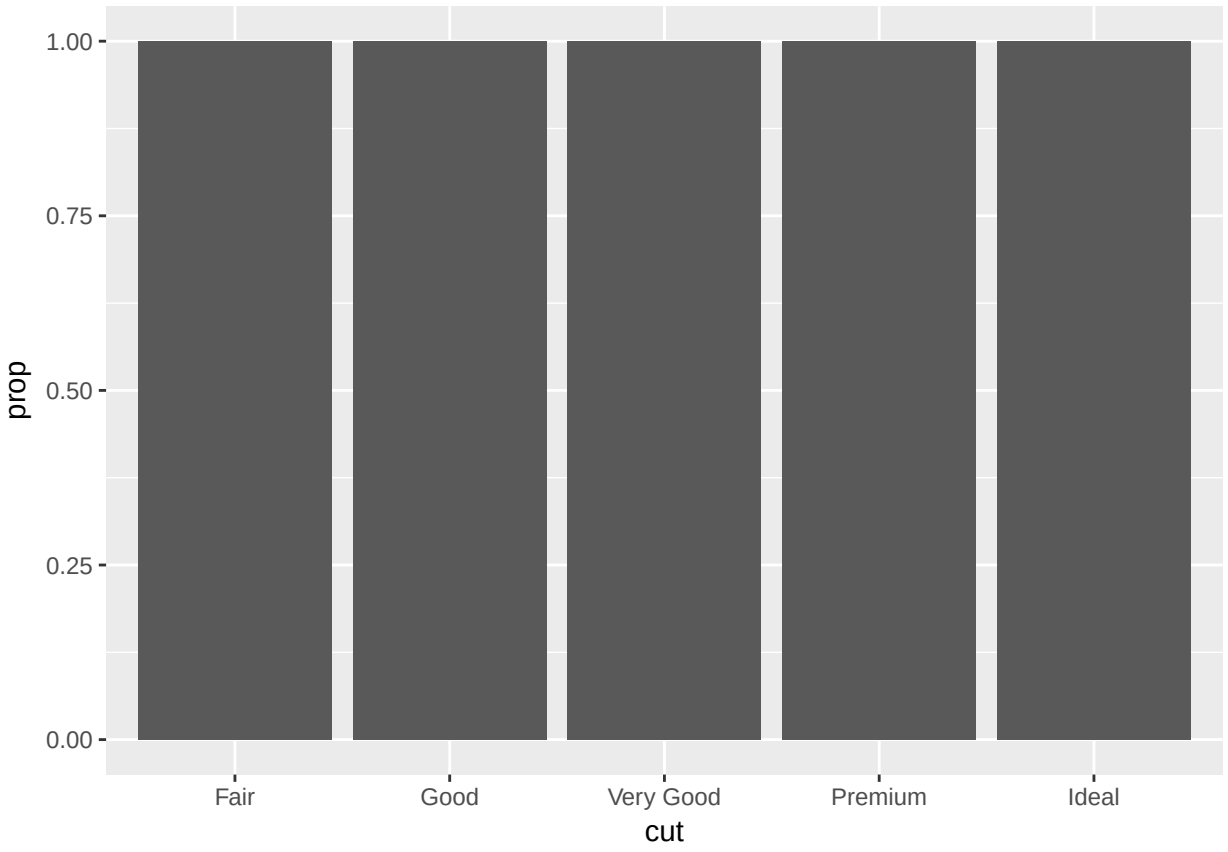
The most important argument is `mapping`, which determines which method will be used to calculate the predictions and confidence interval. To check the description for each argument, type `?stat_smooth()`.

Exercise 5

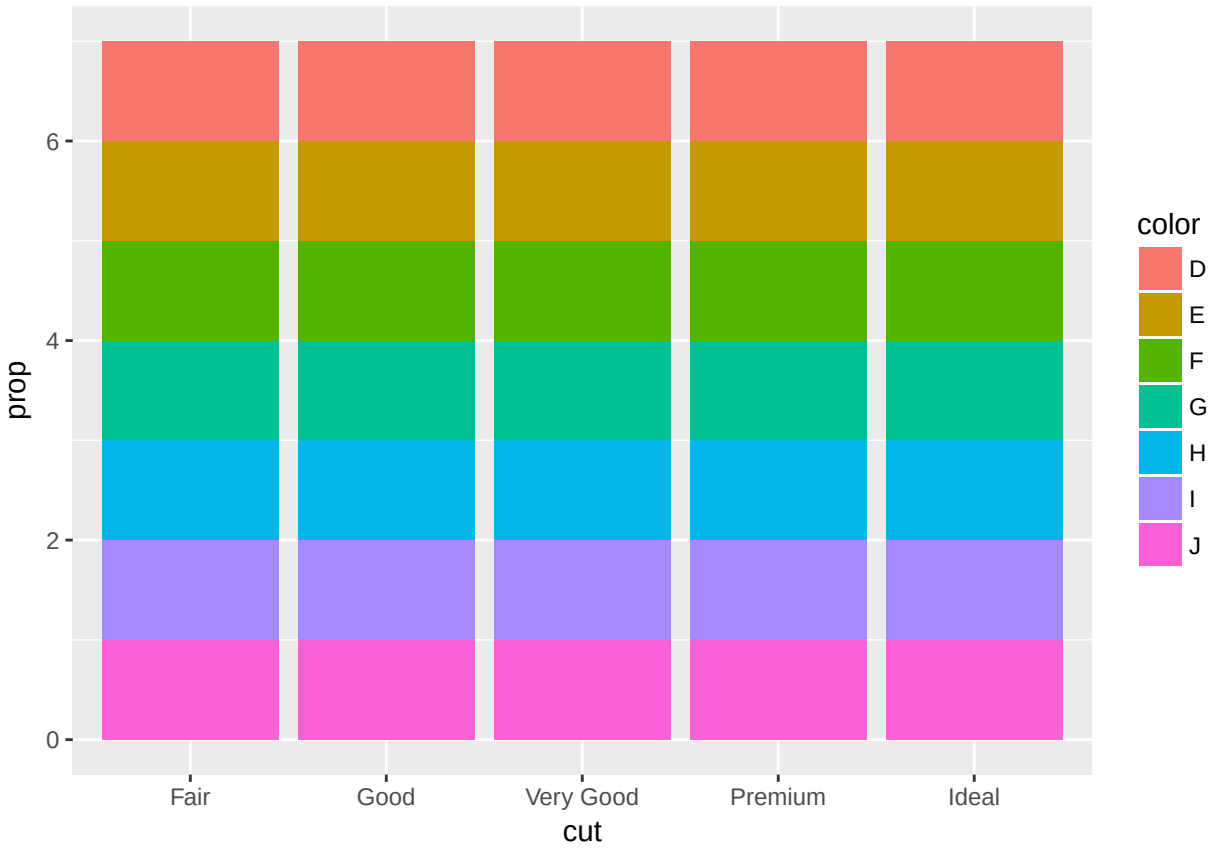
> In our proportion bar chart, we need to set `group = 1`. Why? In other words what is the problem with these two graphs?

I am no sure if my answer is one hundred percent correct for this exercise.

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop..))
```

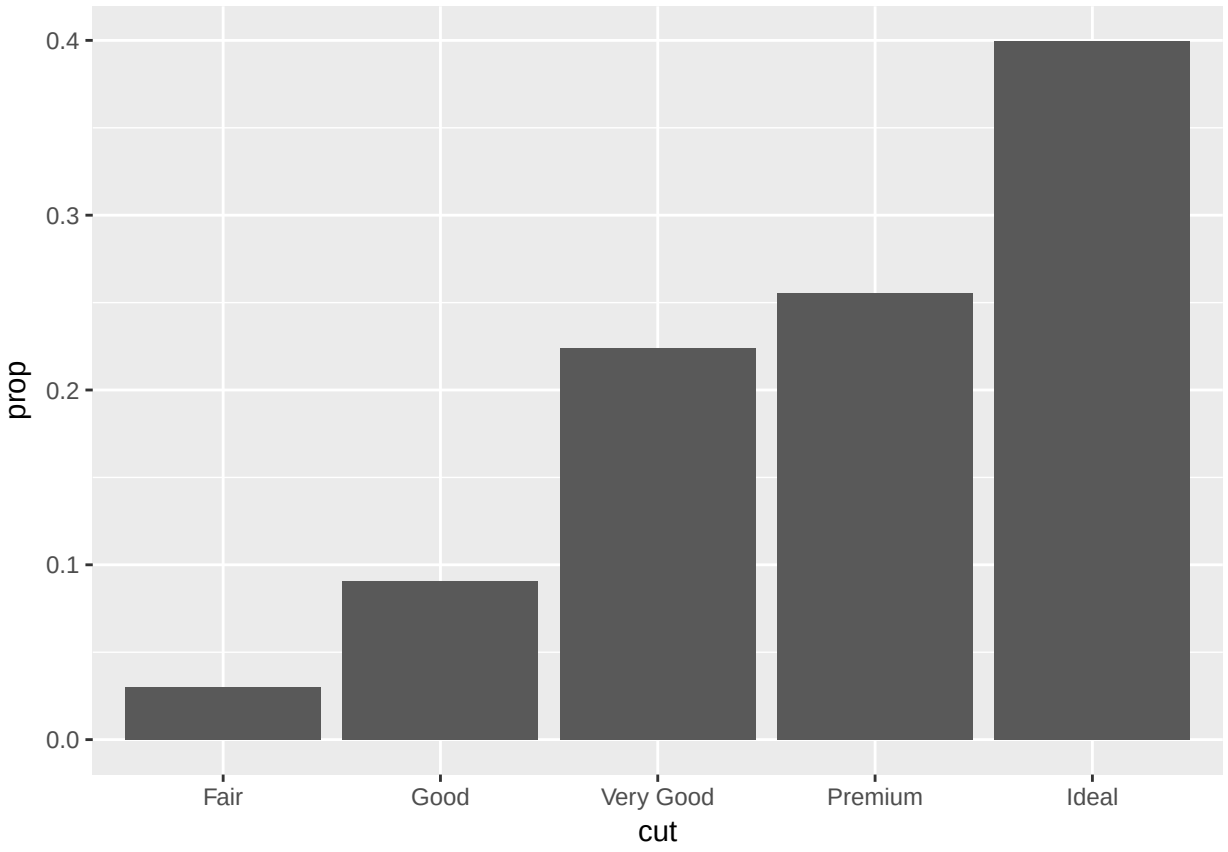



```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = color, y = ..prop..))
```

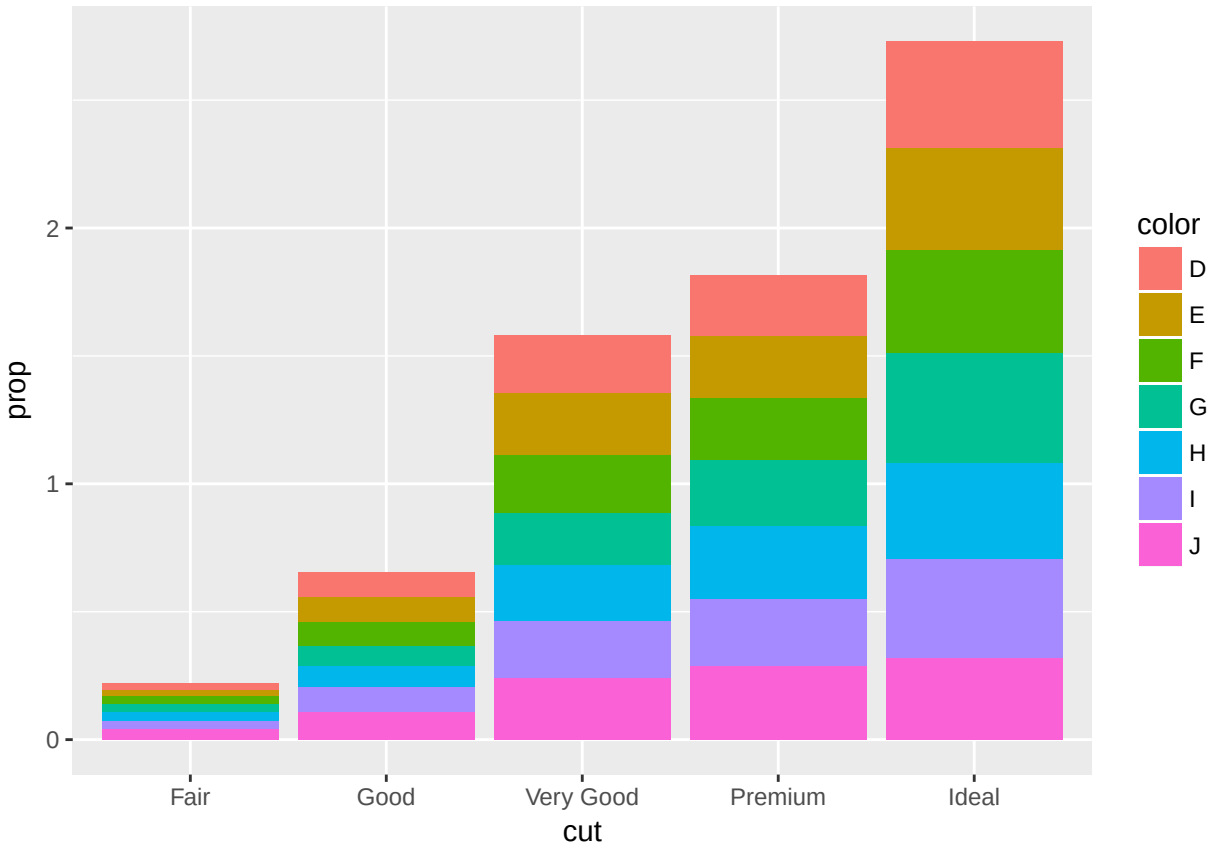


(Answer) `group = 1` is used to set the proportion (y axis) correctly. As you can see in these two plots above, the proportion for all diamonds are equals one (and this is not what we want). So the correct code would be something like this:

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop.., group = 1))
```




```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = color, y = ..prop.., group = color))
```

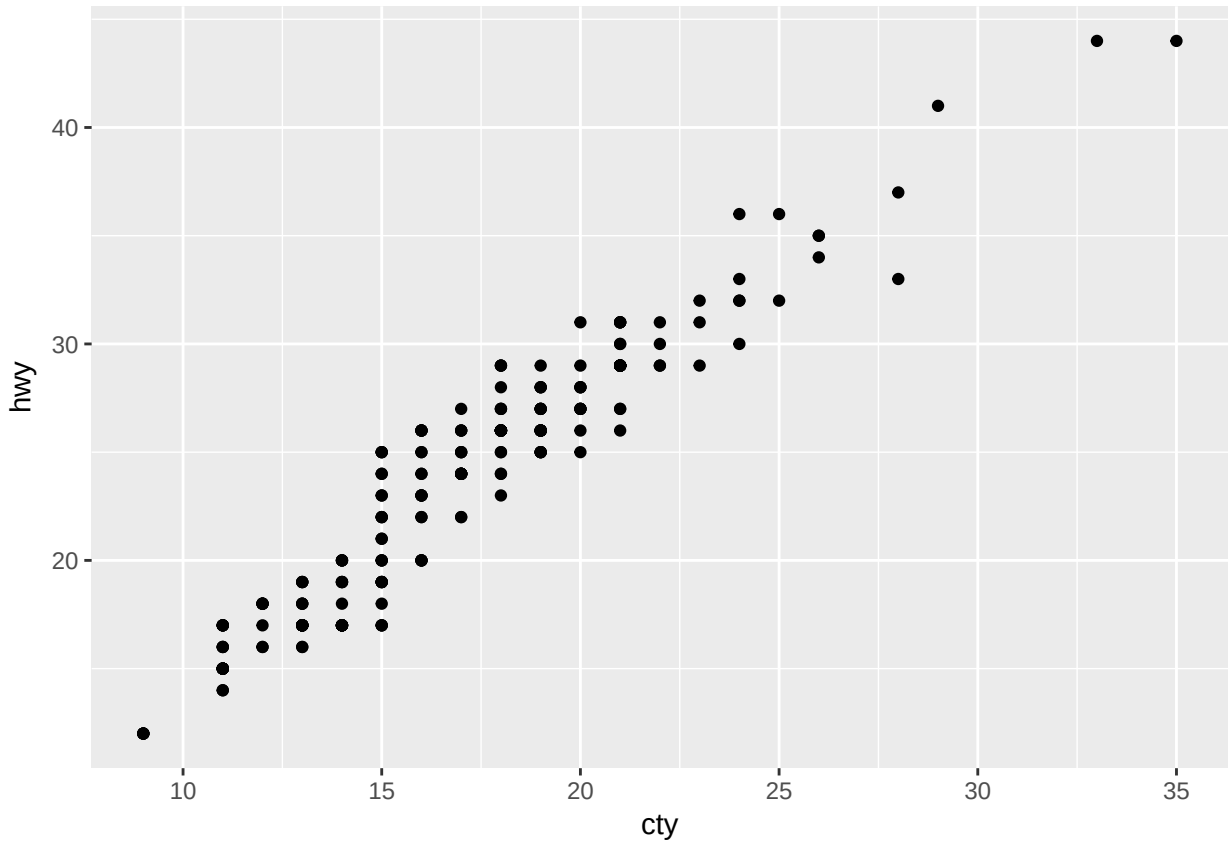


3.8.1 Exercises

Execise 1

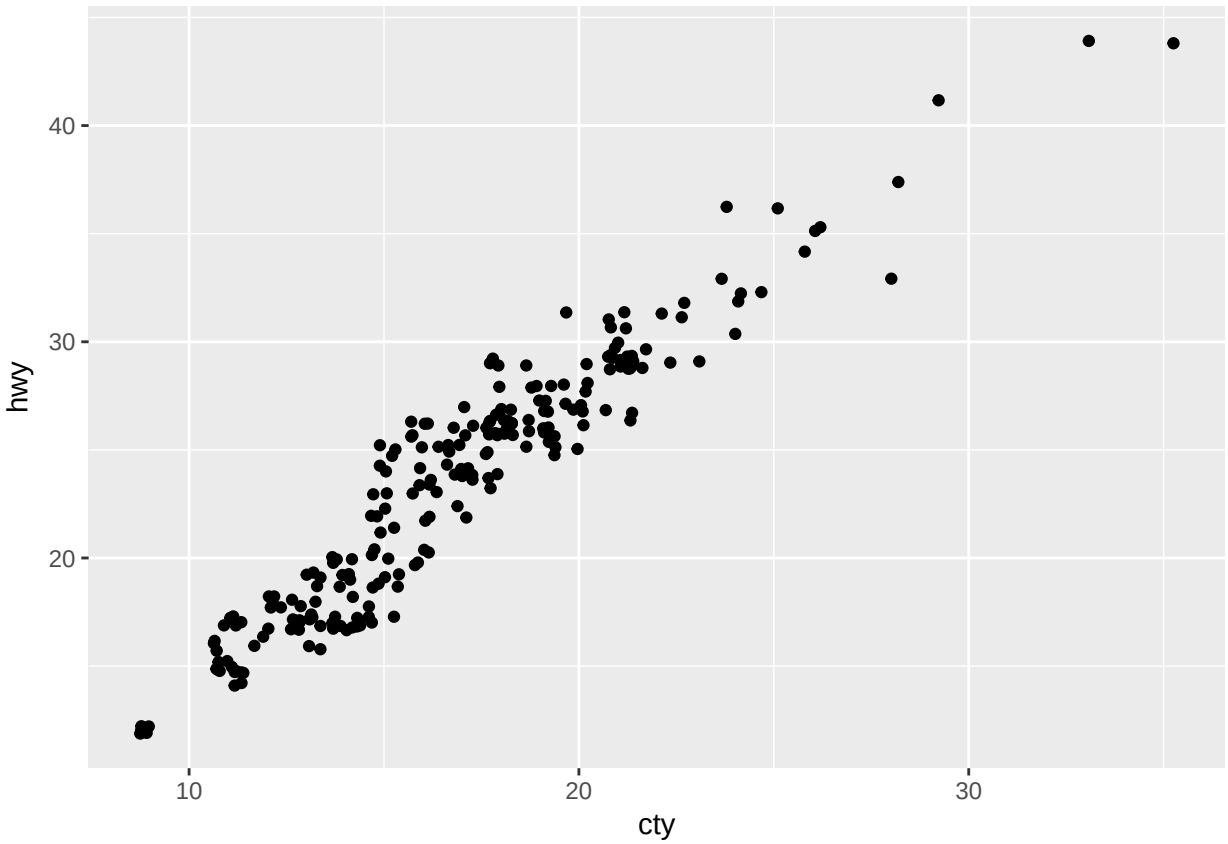
What is the problem with this plot? How could you improve it?

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point()
```



(Answer) Check this plot using same data:

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_jitter()
```



There is a relevant difference between them, right? It is because there are a lot of observations for each combination of `cty` and `hwy`. So, for this situation `geom_jitter()` is a great option, as you can see in our last plot above.

Exercise 2

What parameters to geom_jitter() control the amount of jittering?

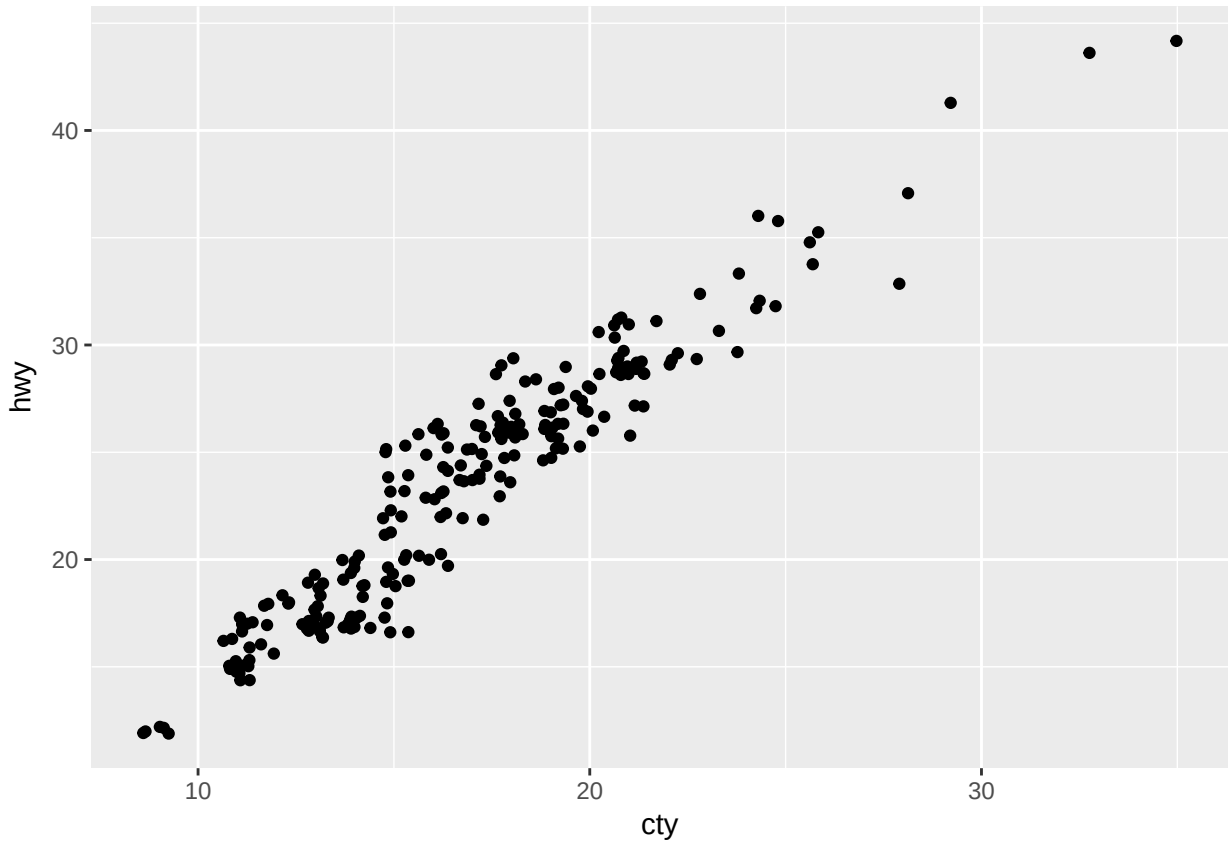
(Answer) As you can read in `position_jitter()` (or `geom_gitter()`) documentation, the parameters used to control the amount of jittering are: * **width:** amount of horizontal jitter * **height:** amount of vertical jitter The jitter is added in both positive and negative directions then the total spread is twice the value specified here. The default value is 40% of the resolution of the data. You can use with `geom_point(position = position_jitter(height, weight))` or with `geom_jitter(height, width)`.

Exercise 3

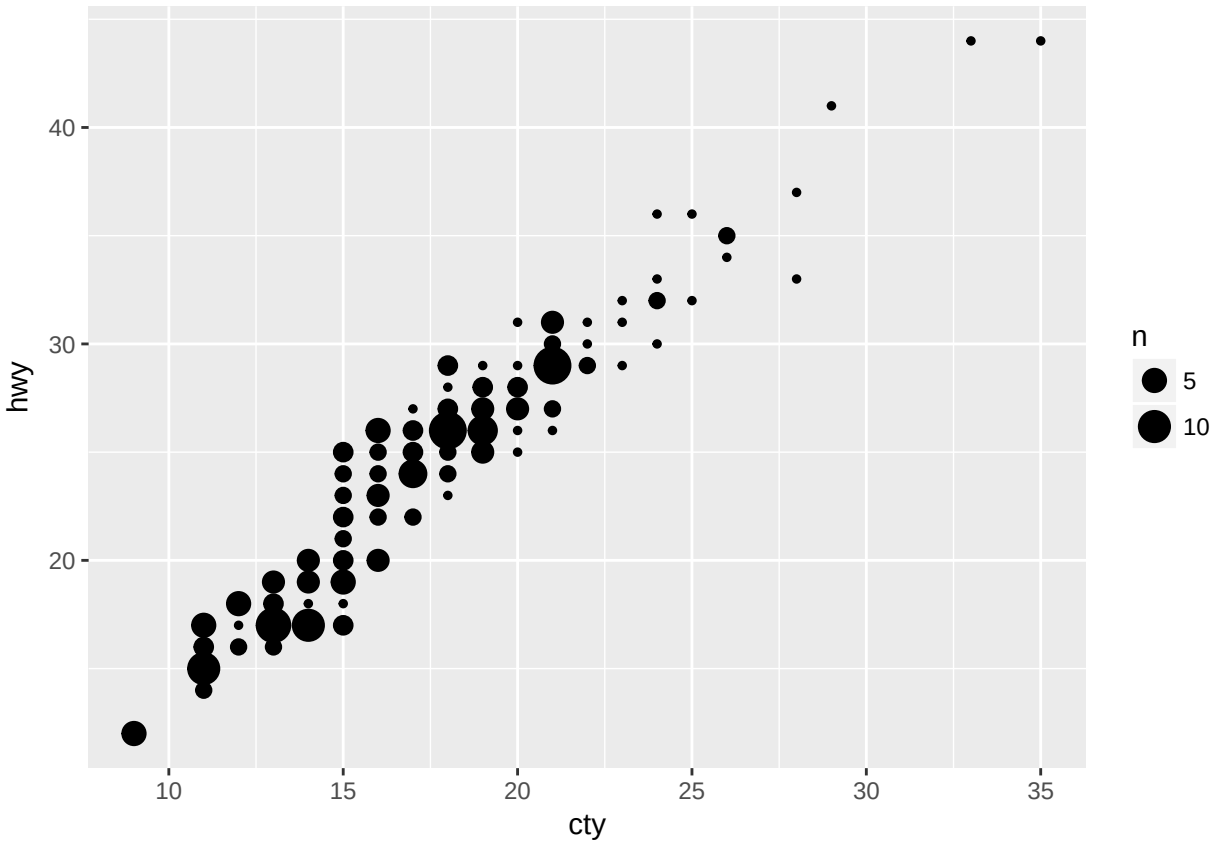
Compare and contrast geom_jitter() with geom_count().

(Answer) Let's plot two graphs, one using `geom_jitter()` and one using `geom_count()`:

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_jitter()
```




```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_count()
```



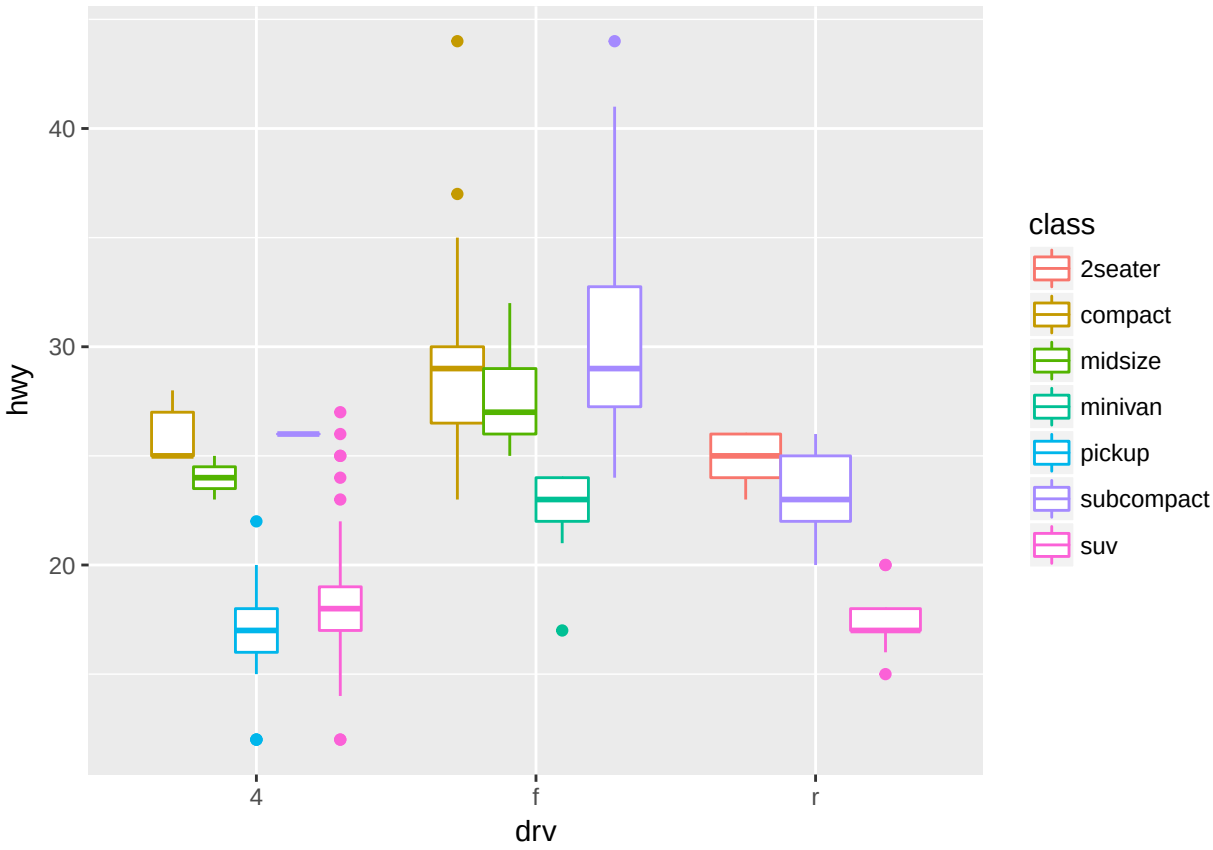
Is really easy to notice the difference between the two. In exercise 1 we verified that `geom_jitter()` adds 'noise' to our graph (both horizontally and vertically) and is easy to see this in the plot. As you can see in the last plot presented `geom_count()` makes agroupation of points and adds a legend to show the scale. In spite of the difference between the two functions, both are useful to understand better where are the concentrations of your dataset.

Exercise 4

> What's the default position adjustment for `geom_boxplot()`? Create a visualisation of the mpg dataset that demonstrates it.

(Answer) By checking the `geom_boxplot()` documentation you are able to verify that the default position for `geom_boxplot()` is `dodge`. Let's plot using `geom_boxplot()` without any custom argument:

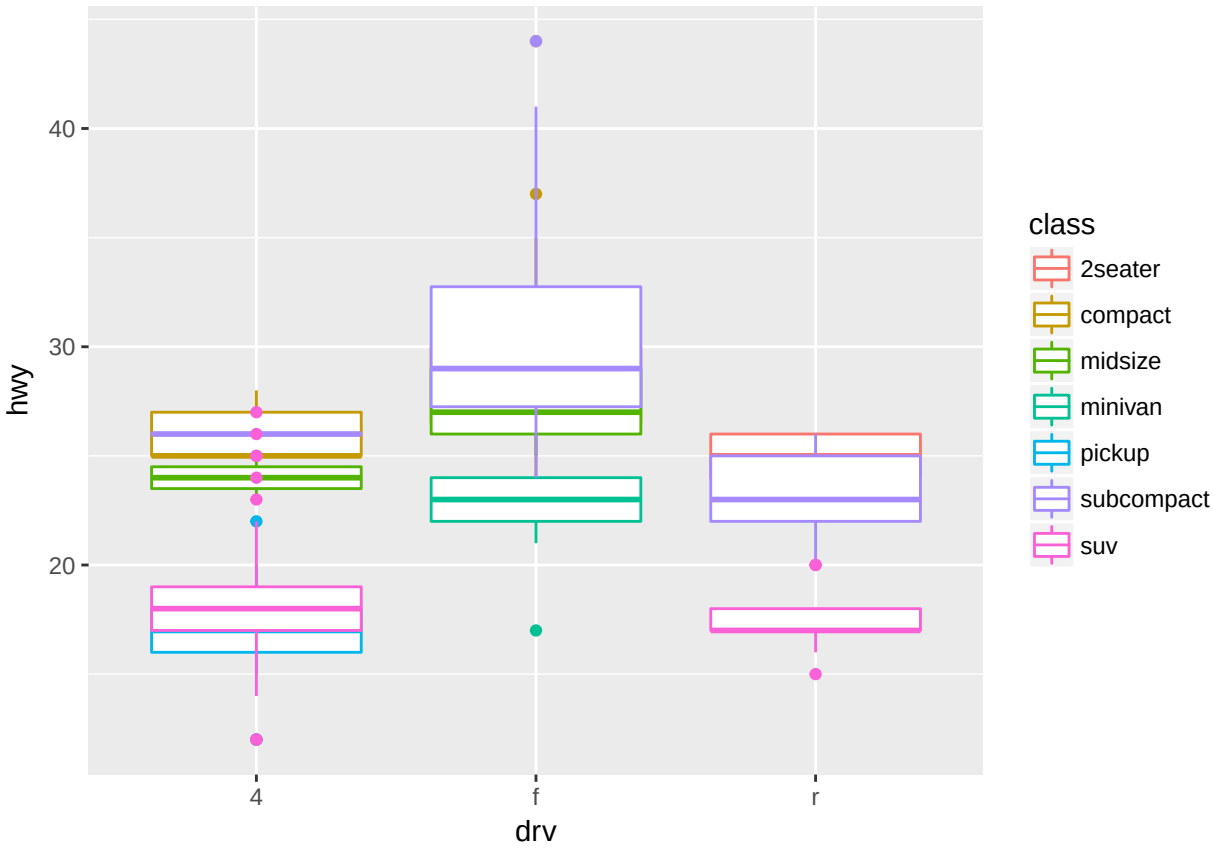
```
ggplot(data = mpg, aes(x = drv, y = hwy, color = class)) +
  geom_boxplot()
```



As you can see in this plot, the different classes from `drv` are side by side and it is because `geom_boxplot()` uses `dodge` as default position.

Now, let's plot overriding the default position adjustment:

```
ggplot(data = mpg, aes(x = drv, y = hwy, color = class)) +
  geom_boxplot(position = 'identity')
```



In this last plot, the different classes from `drv` are not side by side anymore, they are overlapped! This is because now the `geom_boxplot()` is using `identity` as position adjustment insted of `dodge`.

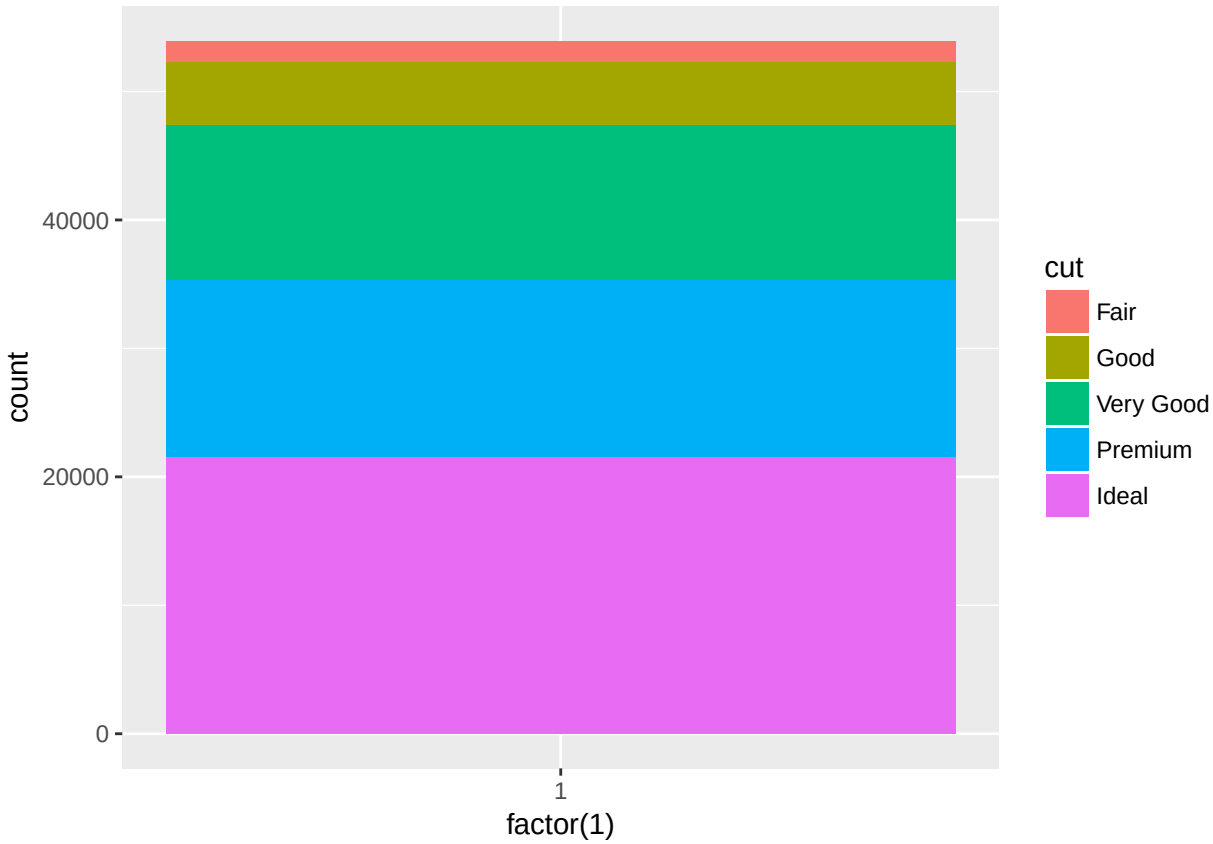
3.9.1 Exercises

Exercise 1

Turn a stacked bar chart into a pie chart using coord_polar().

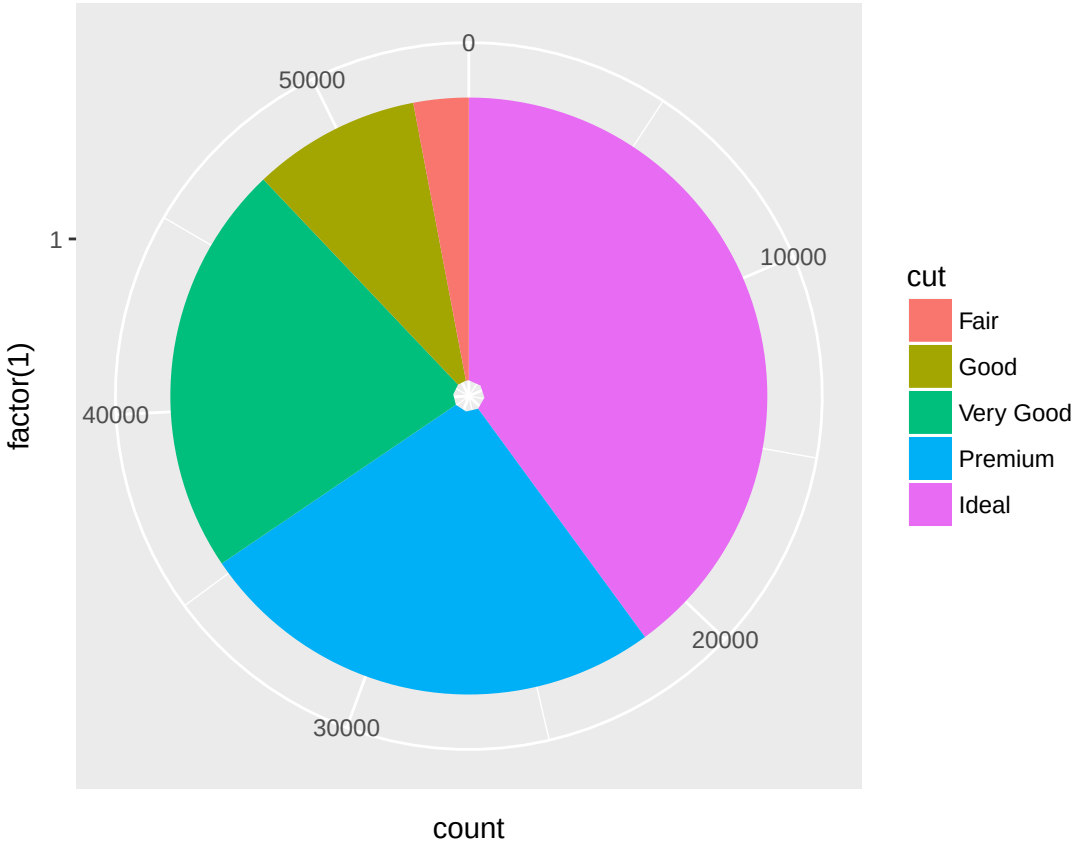
(Answer) Let's plot a stacked bar chart using diamonds data:

```
bar <- ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = factor(1), fill = cut))
print (bar)
```



As you can see in this last code, a variable called `bar` is receiving a plot and then this variable is printed using `print()`. Now, with `bar` variable storing a plot, is easier to transform the bar chart. Let's transform a bar chart into a pie chart (the `coord_polar()` documentation shows how to create a pie chart and many others cool graphs):

```
bar + coord_polar(theta = 'y')
```



Voilà! Pretty cool, right?

Exercise 2

What does `labs()` do? Read the documentation.

(Answer) `labs()` is used to modify axis, legend and plot labels. You can use `labs(y = 'labely', x = 'labelx', title = 'Awesome Plot Title')`

Exercise 3

What's the difference between `coord_quickmap()` and `coord_map()`?

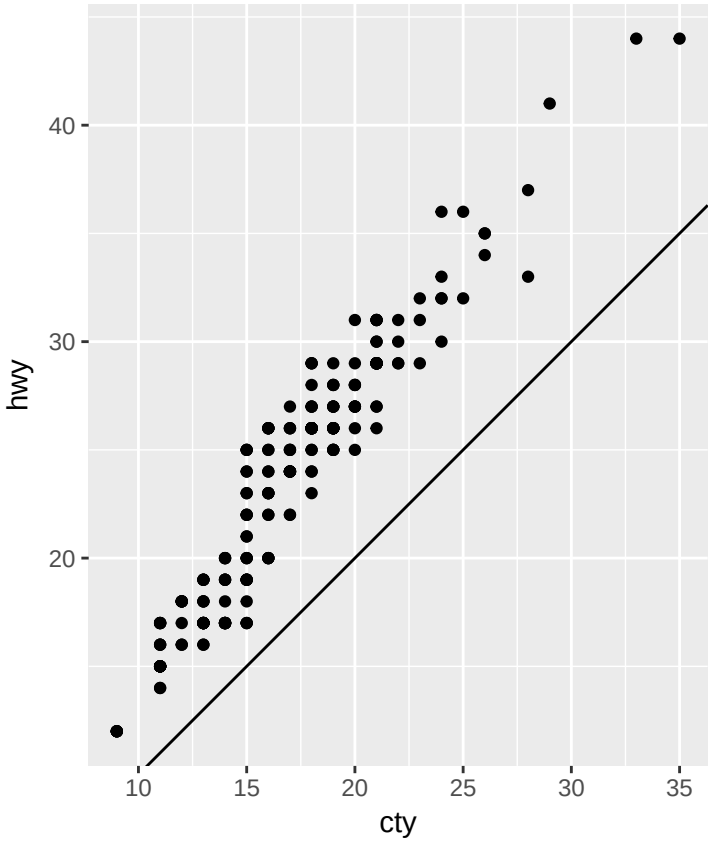
(Answer) Checking the documentation you will notice that `coord_quickmap()` function is a 'faster' option for `coord_map()`, which projects a portion of the earth (approximately spherical) onto a flat 2D plane using any projection defined by the `mapproj` package. So, the `coord_quickmap()` is a quick approximation that does preserve straight lines *and works best for smaller areas closer to the equator.*

Exercise 4

What does the plot below tell you about the relationship between city and highway mpg? Why is `coord_fixed()` important? What does `geom_abline()` do?

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point() +
```

```
  geom_abline() +
  coord_fixed()
```



(Answer) The plot tells me that the relationship between `cty` and `hwy` is linear. `coord_fixed()` is important to make sure that the line (created with `geom_abiline()`) is at 45 degree angle and then make easier to compare the data.

Chapter 4 - Workflow Basics

4.4 Exercises

Exercise 1

> Why does this code not work? Look carefully! (This may seem like an exercise in pointlessness, but training your brain to notice even the tiniest difference will pay off when programming.)

```
my_variable <- 10
my_varıable
```

(Answer) This code does not work because there is an error in the variable name when used to print (my_varıable != my_variable).

Exercise 2

> Tweak each of the following R commands so that they run correctly:

```
library(tidyverse)

ggplot(dota = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

fliter(mpg, cyl = 8)
filter(diamond, carat > 3)
```

(Answer) The errors in the code are listed below:

- dota
- fliter()
- cyl = 8
- diamond

See below the correct implementation to this code:

```
library(tidyverse)

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

filter(mpg, cyl == 8)
filter(diamonds, carat > 3)
```

Always make sure you are typing correctly!

Exercise 3

Press Alt + Shift + K. What happens? How can you get to the same place using the menus?

(Answer) *If you are a Mac user (like me), you should type Option + Shift + K.* This is a keyboard shortcut to check the quick reference for keyboard shortcuts - *inception!*. Besides, this reference can be found in the menu bar (Tools -> Keyboard Shortcuts Help). *However, use a keyboard shortcut is much more awesome. I encourage you to make an effort to use* ***much more*** *your keyboard than your mouse. In the matter of time you will code like a pro (and this will impress your friends and beautiful girls, of course).*

Chapter 5 - Data Transformation

In this chapter we are going to use `nycflights13` dataset and `dplyr` package.

```
library('nycflights13')
library('tidyverse')
```

```
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
```

```
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
# View(flights) to see the whole dataset
```

As you can see, is a litte bit different than the datasets we used until now. This dataset is a special dataframe, called **tibble**, which is a dataframe optimized to work with `tidyverse`.

dplyr quick reference

- `filter(df, vars)`: pick observation by their values
- `arrange(df, vars)`: reorder the rows
- `select(df, vars)`: pick variables by their names
- `mutate(df, vars)`: create new variables with functions of existing variables
- `summarise(df, vars)`: collapse many values down to a single summary

These can **all** be used in conjunction with `group_by()`.

Cool R feature: `filter(flights, month == 11 | month == 12)` and `filter(flights, month %in% c(11,12))` does the same.

Determine if the value iss missing (NA - 'not availables'): `is.na(var)` and you can filter using something like this `filter(df, !is.na(var)`.

5.2.4 Exercises

Exercise 1

Find all flights that:

Had an arrival delay of two or more hours

(Answer) `arr_delay` in minutes

```
filter(flights, arr_delay >= 120)
```

```
## # A tibble: 10,200 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      811            630       101     1047
##  2  2013     1     1      848           1835       853     1001
##  3  2013     1     1      957            733       144     1056
##  4  2013     1     1     1114            900       134     1447
##  5  2013     1     1     1505           1310       115     1638
##  6  2013     1     1     1525           1340       105     1831
##  7  2013     1     1     1549           1445        64     1912
##  8  2013     1     1     1558           1359       119     1718
##  9  2013     1     1     1732           1630        62     2028
## 10  2013     1     1     1803           1620       103     2008
## # ... with 10,190 more rows, and 12 more variables: sched_arr_time <int>,
```

```
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Flew to Houston (`IAH` or `HOU`)

(Answer)

```
filter(flights, dest %in% c('IAH','HOU'))
```

```
## # A tibble: 9,313 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      623            627        -4      933
##  4  2013     1     1      728            732        -4     1041
##  5  2013     1     1      739            739         0     1104
##  6  2013     1     1      908            908         0     1228
##  7  2013     1     1     1028           1026         2     1350
##  8  2013     1     1     1044           1045        -1     1352
##  9  2013     1     1     1114            900       134     1447
## 10  2013     1     1     1205           1200         5     1503
## # ... with 9,303 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Were operated by United, American, or Delta

(Answer)

```
filter(flights, carrier %in% c('UA','AA', 'DL'))
```

```
## # A tibble: 139,504 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      554            600        -6      812
##  5  2013     1     1      554            558        -4      740
##  6  2013     1     1      558            600        -2      753
##  7  2013     1     1      558            600        -2      924
##  8  2013     1     1      558            600        -2      923
##  9  2013     1     1      559            600        -1      941
## 10  2013     1     1      559            600        -1      854
## # ... with 139,494 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Departed in summer (July, August, and September)

(Answer) `month` is a integer between 1 and 12

```
filter(flights, month %in% c(7:9))
```

```
## # A tibble: 86,326 x 19
```

```
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     7     1        1           2029       212      236
##  2  2013     7     1        2           2359         3      344
##  3  2013     7     1       29           2245       104      151
##  4  2013     7     1       43           2130       193      322
##  5  2013     7     1       44           2150       174      300
##  6  2013     7     1       46           2051       235      304
##  7  2013     7     1       48           2001       287      308
##  8  2013     7     1       58           2155       183      335
##  9  2013     7     1      100           2146       194      327
## 10  2013     7     1      100           2245       135      337
## # ... with 86,316 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Arrived more than two hours late, but didn't leave late

(Answer) `arr_delay` in minutes

```
filter(flights, arr_delay > 120 & dep_delay <= 0)
```

```
## # A tibble: 29 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1    27     1419           1420        -1     1754
##  2  2013    10     7     1350           1350         0     1736
##  3  2013    10     7     1357           1359        -2     1858
##  4  2013    10    16      657            700        -3     1258
##  5  2013    11     1      658            700        -2     1329
##  6  2013     3    18     1844           1847        -3       39
##  7  2013     4    17     1635           1640        -5     2049
##  8  2013     4    18      558            600        -2     1149
##  9  2013     4    18      655            700        -5     1213
## 10  2013     5    22     1827           1830        -3     2217
## # ... with 19 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Were delayed by at least an hour, but made up over 30 minutes in flight

(Answer) `dep_delay` and `arr_delay` in minutes

```
filter(flights, dep_delay >= 60 & air_time > 30)
```

```
## # A tibble: 26,657 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      811            630       101     1047
##  2  2013     1     1      826            715        71     1136
##  3  2013     1     1      848           1835       853     1001
##  4  2013     1     1      957            733       144     1056
##  5  2013     1     1     1114            900       134     1447
##  6  2013     1     1     1120            944        96     1331
##  7  2013     1     1     1301           1150        71     1518
```

```
##  8  2013     1     1     1337           1220        77     1649
##  9  2013     1     1     1400           1250        70     1645
## 10  2013     1     1     1505           1310       115     1638
## # ... with 26,647 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Departed between midnight and 6am (inclusive)

(Answer) `dep_time` exactly in midnight or before 6AM (including 6AM). *Here, '600' is 6:00 and '2400' is 00:00*

```
filter(flights, dep_time == 2400 | dep_time < 600)
```

```
## # A tibble: 8,759 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 8,749 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Exercise 2

Another useful `dplyr` filtering helper is `between()`. What does it do? Can you use it to simplify the code needed to answer the previous challenges?

(Answer) `between()` is used to check if the variable is between a range of values (`x >= value1 & x <= value2`). You can use between to filter the flights between months, check this following code:

```
filter(flights, between(month, 7, 9))
```

```
## # A tibble: 86,326 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     7     1        1           2029       212      236
##  2  2013     7     1        2           2359         3      344
##  3  2013     7     1       29           2245       104      151
##  4  2013     7     1       43           2130       193      322
##  5  2013     7     1       44           2150       174      300
##  6  2013     7     1       46           2051       235      304
##  7  2013     7     1       48           2001       287      308
##  8  2013     7     1       58           2155       183      335
##  9  2013     7     1      100           2146       194      327
## 10  2013     7     1      100           2245       135      337
```

```
## # ... with 86,316 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Exercise 3

How many flights have a missing `dep_time`? What other variables are missing? What might these rows represent?

(Answer) Is pretty easy to check this by using `is.na(var)`.

```
filter(flights, is.na(dep_time))
```

```
## # A tibble: 8,255 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1       NA           1630        NA       NA
##  2  2013     1     1       NA           1935        NA       NA
##  3  2013     1     1       NA           1500        NA       NA
##  4  2013     1     1       NA            600        NA       NA
##  5  2013     1     2       NA           1540        NA       NA
##  6  2013     1     2       NA           1620        NA       NA
##  7  2013     1     2       NA           1355        NA       NA
##  8  2013     1     2       NA           1420        NA       NA
##  9  2013     1     2       NA           1321        NA       NA
## 10  2013     1     2       NA           1545        NA       NA
## # ... with 8,245 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
count(filter(flights, is.na(dep_time)))
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1  8255
```

As you can see above, 8225 flights have a missing `dep_time`. More than that, is possible to notice that there is more `NA` values for more variables (`dep_delay`, `arr_time`, `arr_delay`). Puting these 'blocks' together, we can conclude that these 8255 entries are cancelled flights.

Exercise 4

Why is `NA ^ 0` not missing?

(Answer) That is easy. It's beacause `NA^0` must be one (anything zero power is one).

```
NA^0
```

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

Why is `NA | TRUE` not missing?

(Answer) It's because is a `OR` operator. It will test if `NA` is `TRUE` and if not, will check if `TRUE` is `TRUE` (it is). Then, once `TRUE` is `TRUE`, this code returns `TRUE`. *For logical OR operator, the results is TRUE if one (or more) variables is TRUE*

```
NA | TRUE
```

```
## [1] TRUE
```

Why is `FALSE & NA` not missing?

(Answer) It's because to get `TRUE` as result for a logical `AND` operator **all** the variables involved to this operation **must be** `TRUE`. Anything `AND FALSE` will always going to be `FALSE`.

```
FALSE & NA
```

```
## [1] FALSE
```

Besides, `TRUE & NA` or `FALSE | NA` are missing.

5.3.1. Exercises

Exercise 1

How could you use `arrange()` to sort all missing values to the start? (Hint: use `is.na()`).

(Answer)

Exercise 2

Sort flights to find the most delayed flights. Find the flights that left earliest.

(Answer) To get the most delayed flights you should use `arrange()` with `desc()` in `dep_delay` variable:

```
arrange(flights, desc(dep_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     9      641            900      1301     1242
##  2  2013     6    15     1432           1935      1137     1607
##  3  2013     1    10     1121           1635      1126     1239
##  4  2013     9    20     1139           1845      1014     1457
##  5  2013     7    22      845           1600      1005     1044
##  6  2013     4    10     1100           1900       960     1342
##  7  2013     3    17     2321            810       911      135
##  8  2013     6    27      959           1900       899     1236
##  9  2013     7    22     2257            759       898      121
## 10  2013    12     5      756           1700       896     1058
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

To find the flights that left earliest, just use `arrange()` with `dep_delay` variable:

```
arrange(flights, dep_delay)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013    12     7     2040           2123       -43       40
##  2  2013     2     3     2022           2055       -33     2240
```

```
##  3  2013    11    10     1408           1440       -32     1549
##  4  2013     1    11     1900           1930       -30     2233
##  5  2013     1    29     1703           1730       -27     1947
##  6  2013     8     9      729            755       -26     1002
##  7  2013    10    23     1907           1932       -25     2143
##  8  2013     3    30     2030           2055       -25     2213
##  9  2013     3     2     1431           1455       -24     1601
## 10  2013     5     5      934            958       -24     1225
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Exercise 3

Sort flights to find the fastest flights

(Answer)

```
arrange(flights, desc(air_time))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     3    17     1337           1335         2     1937
##  2  2013     2     6      853            900        -7     1542
##  3  2013     3    15     1001           1000         1     1551
##  4  2013     3    17     1006           1000         6     1607
##  5  2013     3    16     1001           1000         1     1544
##  6  2013     2     5      900            900         0     1555
##  7  2013    11    12      936            930         6     1630
##  8  2013     3    14      958           1000        -2     1542
##  9  2013    11    20     1006           1000         6     1639
## 10  2013     3    15     1342           1335         7     1924
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Exercise 4

Which flights travelled the longest? Which travelled the shortest?

(Answer) To find the flights that travelled the longest distance you should use `arrange()` and `desc()` with `distance` variable:

```
arrange(flights, desc(distance))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      857            900        -3     1516
##  2  2013     1     2      909            900         9     1525
##  3  2013     1     3      914            900        14     1504
##  4  2013     1     4      900            900         0     1516
```

```
##  5  2013     1     5      858            900        -2     1519
##  6  2013     1     6     1019            900        79     1558
##  7  2013     1     7     1042            900       102     1620
##  8  2013     1     8      901            900         1     1504
##  9  2013     1     9      641            900      1301     1242
## 10  2013     1    10      859            900        -1     1449
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

And the flights that travelled the shortest distance, just use `arrange()` with `distance`:

```
arrange(flights, distance)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     7    27       NA            106        NA       NA
##  2  2013     1     3     2127           2129        -2     2222
##  3  2013     1     4     1240           1200        40     1333
##  4  2013     1     4     1829           1615       134     1937
##  5  2013     1     4     2128           2129        -1     2218
##  6  2013     1     5     1155           1200        -5     1241
##  7  2013     1     6     2125           2129        -4     2224
##  8  2013     1     7     2124           2129        -5     2212
##  9  2013     1     8     2127           2130        -3     2304
## 10  2013     1     9     2126           2129        -3     2217
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

5.4.1 Exercises

Exercise 1

> Brainstorm as many ways as possible to select `dep_time`, `dep_delay`, `arr_time`, and `arr_delay` from `flights`.

(Answer) The most basic way to select variables is by selecting by their names:

```
select(flights, dep_time, dep_delay, arr_time, arr_delay)
```

```
## # A tibble: 336,776 x 4
##    dep_time dep_delay arr_time arr_delay
##       <int>     <dbl>    <int>     <dbl>
##  1      517         2      830        11
##  2      533         4      850        20
##  3      542         2      923        33
##  4      544        -1     1004       -18
##  5      554        -6      812       -25
##  6      554        -4      740        12
##  7      555        -5      913        19
##  8      557        -3      709       -14
```

```
##  9      557        -3      838        -8
## 10      558        -2      753         8
## # ... with 336,766 more rows
```

You also can use some help functions (like `starts_with()` and `matches()`) to select these variables:

```
select(flights, starts_with('de'), starts_with('ar'))
```

```
## # A tibble: 336,776 x 5
##    dep_time dep_delay dest  arr_time arr_delay
##       <int>     <dbl> <chr>    <int>     <dbl>
##  1      517         2 IAH        830        11
##  2      533         4 IAH        850        20
##  3      542         2 MIA        923        33
##  4      544        -1 BQN       1004       -18
##  5      554        -6 ATL        812       -25
##  6      554        -4 ORD        740        12
##  7      555        -5 FLL        913        19
##  8      557        -3 IAD        709       -14
##  9      557        -3 MCO        838        -8
## 10      558        -2 ORD        753         8
## # ... with 336,766 more rows
```

```
select(flights, starts_with('dep'), starts_with('arr'))
```

```
## # A tibble: 336,776 x 4
##    dep_time dep_delay arr_time arr_delay
##       <int>     <dbl>    <int>     <dbl>
##  1      517         2      830        11
##  2      533         4      850        20
##  3      542         2      923        33
##  4      544        -1     1004       -18
##  5      554        -6      812       -25
##  6      554        -4      740        12
##  7      555        -5      913        19
##  8      557        -3      709       -14
##  9      557        -3      838        -8
## 10      558        -2      753         8
## # ... with 336,766 more rows
```

```
select(flights, starts_with('dep_'), starts_with('arr_'))
```

```
## # A tibble: 336,776 x 4
##    dep_time dep_delay arr_time arr_delay
##       <int>     <dbl>    <int>     <dbl>
##  1      517         2      830        11
##  2      533         4      850        20
##  3      542         2      923        33
##  4      544        -1     1004       -18
##  5      554        -6      812       -25
##  6      554        -4      740        12
##  7      555        -5      913        19
##  8      557        -3      709       -14
##  9      557        -3      838        -8
## 10      558        -2      753         8
## # ... with 336,766 more rows
```

Or if you want to use regex, which is much more awesome:

```
select(flights, matches('^(dep|arr).*(time|delay)$'))
```

```
## # A tibble: 336,776 x 4
##    dep_time dep_delay arr_time arr_delay
##       <int>     <dbl>    <int>     <dbl>
##  1      517         2      830        11
##  2      533         4      850        20
##  3      542         2      923        33
##  4      544        -1     1004       -18
##  5      554        -6      812       -25
##  6      554        -4      740        12
##  7      555        -5      913        19
##  8      557        -3      709       -14
##  9      557        -3      838        -8
## 10      558        -2      753         8
## # ... with 336,766 more rows
```

Which `^` means that the string must start with `dep` OR `arr` and `$` means that the string must end with `time` OR `delay` (`.` can be any character and `*` means that any character (`.`) can repeat many times).

Exercise 2

What happens if you include the name of a variable multiple times in a `select()` call?

(Answer) Let's try:

```
select(flights, dep_time, dep_time)
```

```
## # A tibble: 336,776 x 1
##    dep_time
##       <int>
##  1      517
##  2      533
##  3      542
##  4      544
##  5      554
##  6      554
##  7      555
##  8      557
##  9      557
## 10      558
## # ... with 336,766 more rows
```

```
select(flights, dep_time, dep_time)
```

```
## # A tibble: 336,776 x 1
##    dep_time
##       <int>
##  1      517
##  2      533
##  3      542
##  4      544
##  5      554
##  6      554
```

```
##  7      555
##  8      557
##  9      557
## 10      558
## # ... with 336,766 more rows
```

As you can see, we got the same result for both. So, `select()` function ignores the repeated variables.

Exercise 3

What does the `one_of()` function do? Why might it be helpful in conjunction with this vector?

```
vars <- c("year", "month", "day", "dep_delay", "arr_delay")
```

(Answer) Let's try:

```
select(flights, one_of(vars))
```

```
## # A tibble: 336,776 x 5
##     year month   day dep_delay arr_delay
##    <int> <int> <int>     <dbl>     <dbl>
##  1  2013     1     1         2        11
##  2  2013     1     1         4        20
##  3  2013     1     1         2        33
##  4  2013     1     1        -1       -18
##  5  2013     1     1        -6       -25
##  6  2013     1     1        -4        12
##  7  2013     1     1        -5        19
##  8  2013     1     1        -3       -14
##  9  2013     1     1        -3        -8
## 10  2013     1     1        -2         8
## # ... with 336,766 more rows
```

With `one_of` you can use vectors (vector `vars` in this particular example) with `select()` to select particular variables.

Exercise 4

Does the result of running the following code surprise you? How do the select helpers deal with case by default? How can you change that default?

```
select(flights, contains("TIME"))
```

```
## # A tibble: 336,776 x 6
##    dep_time sched_dep_time arr_time sched_arr_time air_time
##       <int>          <int>    <int>          <int>    <dbl>
##  1      517            515      830            819      227
##  2      533            529      850            830      227
##  3      542            540      923            850      160
##  4      544            545     1004           1022      183
##  5      554            600      812            837      116
##  6      554            558      740            728      150
##  7      555            600      913            854      158
##  8      557            600      709            723       53
##  9      557            600      838            846      140
## 10      558            600      753            745      138
```

```
## # ... with 336,766 more rows, and 1 more variable: time_hour <dttm>
```

(Answer) I am not surprised at all. `contains()` function uses downcase as default (ignore uppercase). In most of datasets the variable names are all downcase so I think is a nice default definition. However, if you want to change this default instead of to change (downcase) all the variable names from your dataset, you can use `select()` like in this code below:

```
select(flights, contains("TIME", ignore.case = FALSE))
```

```
## # A tibble: 336,776 x 0
```

As you can see, there is no results when we use `select()` with case sensitive (considering uppercase).

5.5.2 Exercises

Exercise 1

> Currently `dep_time` and `sched_dep_time` are convenient to look at, but hard to compute with because they're not really continuous numbers. Convert them to a more convenient representation of number of minutes since midnight.

(Answer) Finally: fix this horrible representation.

```
mutate(flights,
  dep_time_mins = (dep_time %/% 100)*60 + (dep_time %% 100),
  sched_dep_time_mins = (sched_dep_time %/% 100)*100 + (sched_dep_time %% 100)
)
```

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 14 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, dep_time_mins <dbl>,
## #   sched_dep_time_mins <dbl>
```

Cool!

Exercise 2

> Compare `air_time` with `arr_time - dep_time`. What do you expect to see? What do you see? What do you need to do to fix it?

(Answer) At first time we could expect to see the `air_time` by doint this difference between `air_time` and `dep_time`. However, since `dep_time` are in many different time zones, this first assumption is not true.

Exercise 3

> Compare `dep_time`, `sched_dep_time`, and `dep_delay`. How would you expect those three numbers to be related?

(Answer) Let's check these three variables:

```
select(flights, dep_time, sched_dep_time, dep_delay)
```

```
## # A tibble: 336,776 x 3
##    dep_time sched_dep_time dep_delay
##       <int>          <int>     <dbl>
##  1      517            515         2
##  2      533            529         4
##  3      542            540         2
##  4      544            545        -1
##  5      554            600        -6
##  6      554            558        -4
##  7      555            600        -5
##  8      557            600        -3
##  9      557            600        -3
## 10      558            600        -2
## # ... with 336,766 more rows
```

At first time you can think that`dep_time` minus `dep_delay` is equal to `sched_dep_time`. However this is not true for all our entries (thanks to time zones, again).

Exercise 4

> Find the 10 most delayed flights using a ranking function. How do you want to handle ties? Carefully read the documentation for `min_rank()`.

(Answer) First, let's order by dep_delay (from maximum delay to minimum delay):

```
arrange(flights, min_rank(-dep_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     9      641            900      1301     1242
##  2  2013     6    15     1432           1935      1137     1607
##  3  2013     1    10     1121           1635      1126     1239
##  4  2013     9    20     1139           1845      1014     1457
##  5  2013     7    22      845           1600      1005     1044
##  6  2013     4    10     1100           1900       960     1342
##  7  2013     3    17     2321            810       911      135
##  8  2013     6    27      959           1900       899     1236
##  9  2013     7    22     2257            759       898      121
## 10  2013    12     5      756           1700       896     1058
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Cool, right? As you can see, to order from maximum to minimum delay you should use `min_rank()` with negative `dep_delay`.

Now, to find the 10 most delayed flights we should filter by delay

```
rank_delay_flights <- arrange(flights, min_rank(-dep_delay))
top10_delay_flights <- filter(rank_delay_flights, min_rank(-dep_delay) <= 10)
top10_delay_flights
```

```
## # A tibble: 10 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     9      641            900      1301     1242
##  2  2013     6    15     1432           1935      1137     1607
##  3  2013     1    10     1121           1635      1126     1239
##  4  2013     9    20     1139           1845      1014     1457
##  5  2013     7    22      845           1600      1005     1044
##  6  2013     4    10     1100           1900       960     1342
##  7  2013     3    17     2321            810       911      135
##  8  2013     6    27      959           1900       899     1236
##  9  2013     7    22     2257            759       898      121
## 10  2013    12     5      756           1700       896     1058
## # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

However, for this situation is better to create a new column to store the delay ranking and then use arrange and filter functions:

```
rank_delay_flights <- mutate(flights, rank_dep_delay = min_rank(-dep_delay))
rank_delay_flights <- arrange(rank_delay_flights, rank_dep_delay)
top10_delay_flights <- filter(rank_delay_flights, rank_dep_delay <= 10)
top10_delay_flights
```

```
## # A tibble: 10 x 20
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     9      641            900      1301     1242
##  2  2013     6    15     1432           1935      1137     1607
##  3  2013     1    10     1121           1635      1126     1239
##  4  2013     9    20     1139           1845      1014     1457
##  5  2013     7    22      845           1600      1005     1044
##  6  2013     4    10     1100           1900       960     1342
##  7  2013     3    17     2321            810       911      135
##  8  2013     6    27      959           1900       899     1236
##  9  2013     7    22     2257            759       898      121
## 10  2013    12     5      756           1700       896     1058
## # ... with 13 more variables: sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>, rank_dep_delay <int>
```

Exercise 5

What does 1:3 + 1:10 return? Why?

(Answer) Return the sum of two vectors (1 to 3 and 1 to 10):

```r
1:3 + 1:10
```

```
## Warning in 1:3 + 1:10: comprimento do objeto maior não é múltiplo do
## comprimento do objeto menor
```

```
##  [1]  2  4  6  5  7  9  8 10 12 11
```

However, this is not a smart thing to do. As you can see in the warning message, the two objects are not in the same size/the size of vector with more elements is not divisible by the size of the vector with less elements. For a situation like that (vectors with different sizes), the shorter vector is going to be reused. So, what this code is doing is: `(1+1, 2+,2, 3+3, 4+1, 5+2, 6+3, 7+1, 8+2, 9+3, 10+1)`.

Exercise 6

What trigonometric functions does R provide?

(Answer) The answer for this question is in helper for trigonometric functions (`help('Trig')`).

5.5.2 Exercises