Data Visualization with ggplot2 (Part 1)

Lessons from DataCamp

Contents

Introduction

The following document outlines the written portion of the lessons from DataCamp's Data Visualization with ggplot2 (Part 1). This requires Intermediate R-Knowledge.

As a note: All text is completely copied and pasted from the course. There are instances where the document refers to the "editor on the right", please note, that in this notebook document all of the instances are noted in the "r-chunks" (areas containing working r-code), which occurs below the text, rather than to the right. Furthermore, This lesson contained instructional videos at the beginning of new concepts that are not detailed in this document. However, even without these videos, the instructions are quite clear in indicating what the code is accomplishing.

If you have this document open on "R-Notebook", simply click "run" -> "Run all" (Or just press 'ctrl + alt + r'), let the "r-chunks" run (This might take a bit of time) then click "Preview". There are 5 necessary datasets to run this program, please create an r-project with this data or set a working directory (required files names are available in the "Required data for this session" section)

This document was created by Neil Yetz on 03/09/2018. Please send any questions or concerns in this document to Neil at ndyetz@gmail.com

Required packages for this session

Below are the install.packages and libraries you will need to have in order to run this session successfully.

```
#install.packages("RColorBrewer")
#install.packages("titanic")
library(ggplot2)
library(tidyr)
library(tidyverse)
library(car) #<- Vocab dataset
library(RColorBrewer)
library(titanic) # <- titanic dataset
```

Required data for this session

```
load("diamonds.Rdata")
load("fish.Rdata")
load("iris.Rdata")
load("recess.Rdata")


titanic <- as_data_frame(titanic_train) %>%
  select(Survived, Pclass, Sex, Age) %>%
  na.omit()
```

Course Description

The ability to produce meaningful and beautiful data visualizations is an essential part of your skill set as a data scientist. This course, the first R data visualization tutorial in the series, introduces you to the principles of good visualizations and the grammar of graphics plotting concepts implemented in the ggplot2 package. ggplot2 has become the go-to tool for flexible and professional plots in R. Here, we'll examine the first three essential layers for making a plot - Data, Aesthetics and Geometries. By the end of the course you will be able to make complex exploratory plots.

Chapter 1: Introduction

In this chapter we'll get you into the right frame of mind for developing meaningful visualizations with R. You'll understand that as a communications tool, visualizations require you to think about your audience first. You'll also be introduced to the basics of ggplot2 - the 7 different grammatical elements (layers) and aesthetic mappings.

Exploring ggplot2, part 1

To get a first feel for `ggplot2`, let's try to run some basic `ggplot2` commands. Together, they build a plot of the `mtcars` dataset that contains information about 32 cars from a 1973 Motor Trend magazine. This dataset is small, intuitive, and contains a variety of continuous and categorical variables.

INSTRUCTIONS

Load the `ggplot2` package using `library()`. It is already installed on DataCamp's servers.

Use `str()` to explore the structure of the `mtcars` dataset.

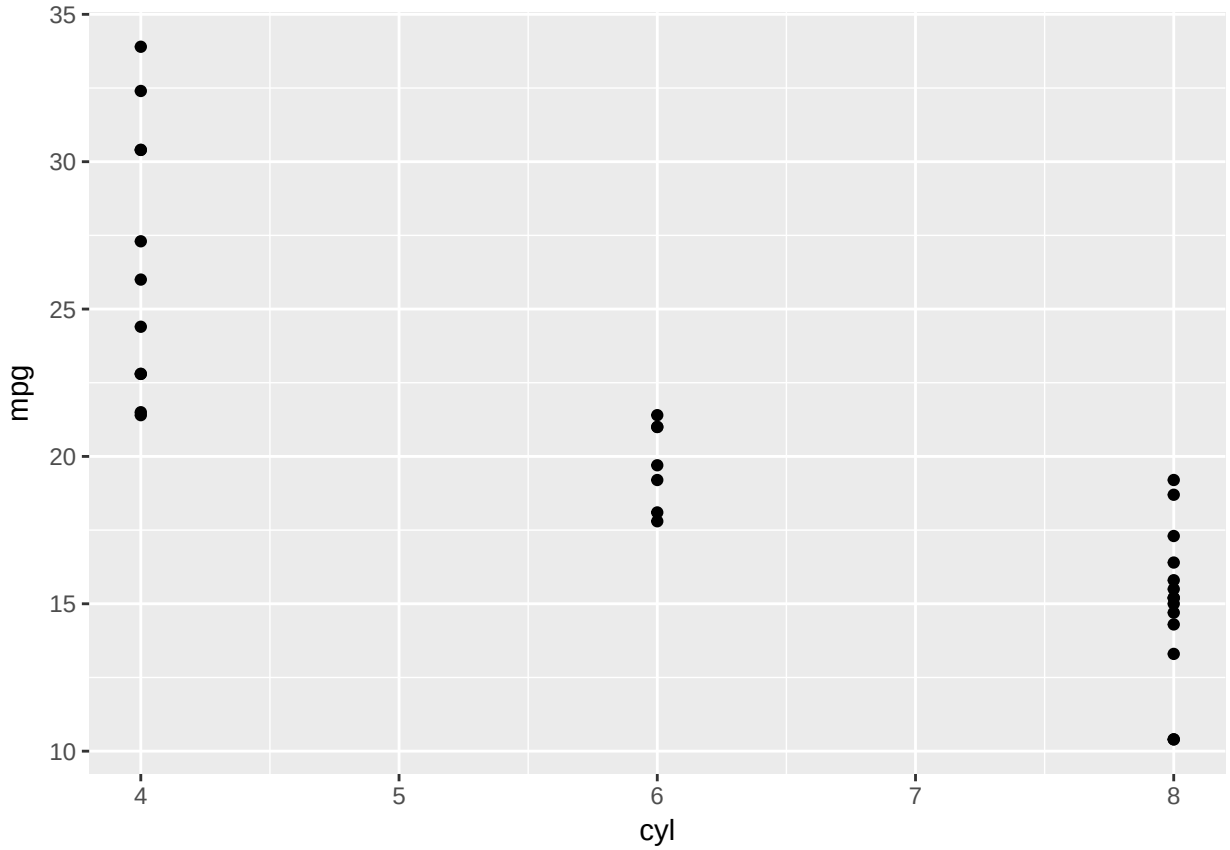
Hit Submit Answer. This will execute the example code on the right. See if you can understand what `ggplot` does with the data.

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

# Explore the mtcars data frame with str()
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
```

```
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
# Execute the following command
ggplot(mtcars, aes(x = cyl, y = mpg)) +
  geom_point()
```



Exploring ggplot2, part 2

The plot from the previous exercise wasn't really satisfying. Although `cyl` (the number of cylinders) is categorical, it is classified as numeric in `mtcars`. You'll have to explicitly tell `ggplot2` that `cyl`is a categorical variable.

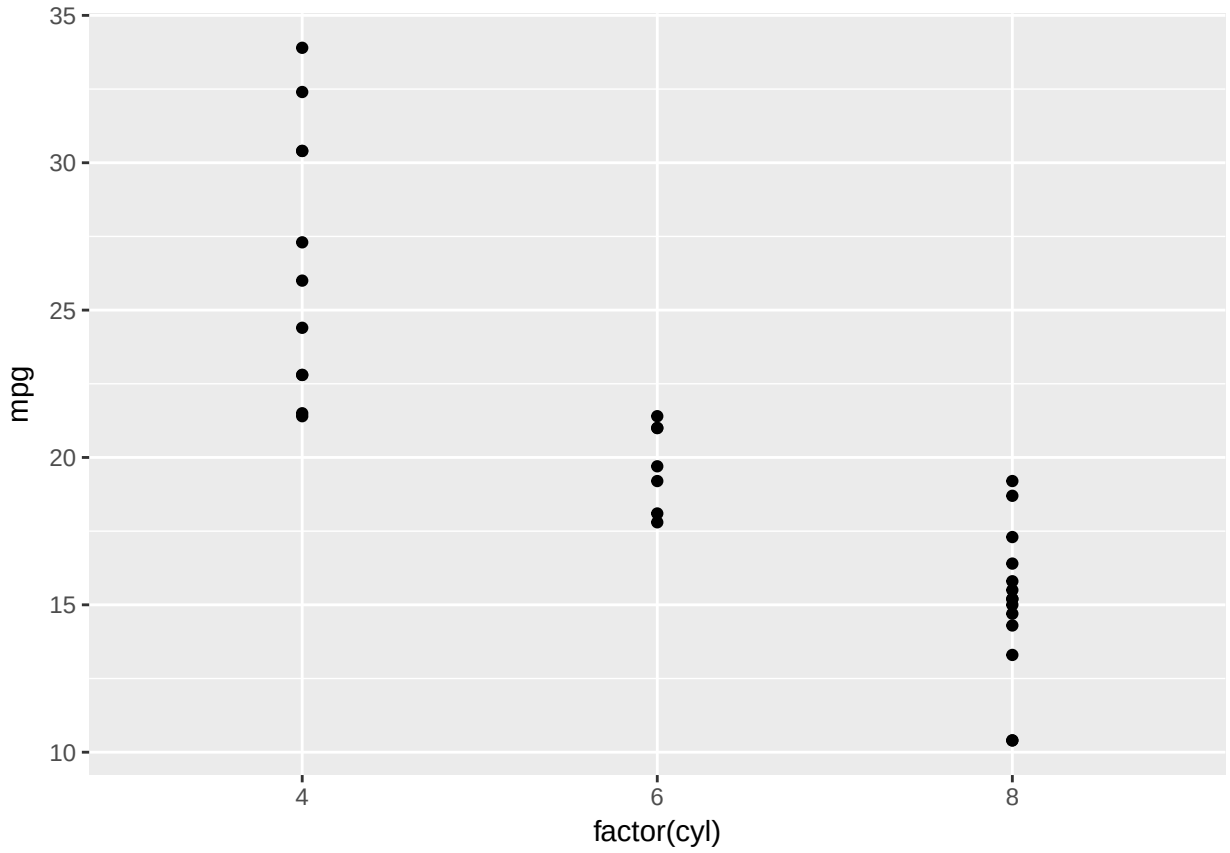
INSTRUCTIONS

Change the `ggplot()` command by wrapping`factor()` around `cyl`.

Hit Submit Answer and see if the resulting plot is better this time.

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

```
# Change the command below so that cyl is treated as factor
ggplot(mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_point()
```



Exploring ggplot2, part 3

We'll use several datasets throughout the courses to showcase the concepts discussed in the videos. In the previous exercises, you already got to know `mtcars`. Let's dive a little deeper to explore the three main topics in this course: The data, aesthetics, and geom layers.

The `mtcars` dataset contains information about 32 cars from 1973 Motor Trend magazine. This dataset is small, intuitive, and contains a variety of continuous and categorical variables.

You're encouraged to think about how the examples and concepts we discuss throughout these data viz courses apply to your own data-sets!

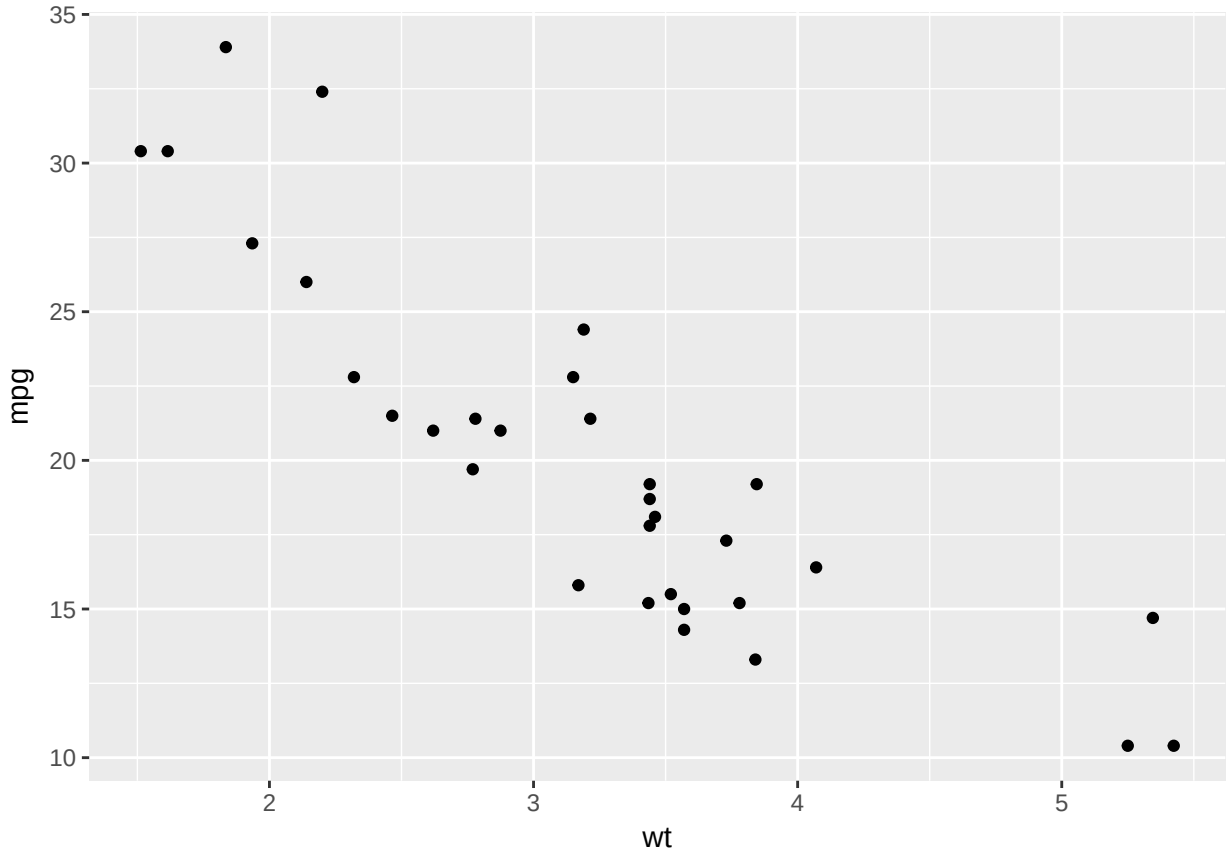
INSTRUCTIONS

`ggplot2` has already been loaded for you. Take a look at the first command. It plots the mpg (miles per galon) against the weight (in thousands of pounds). You don't have to change anything about this command.

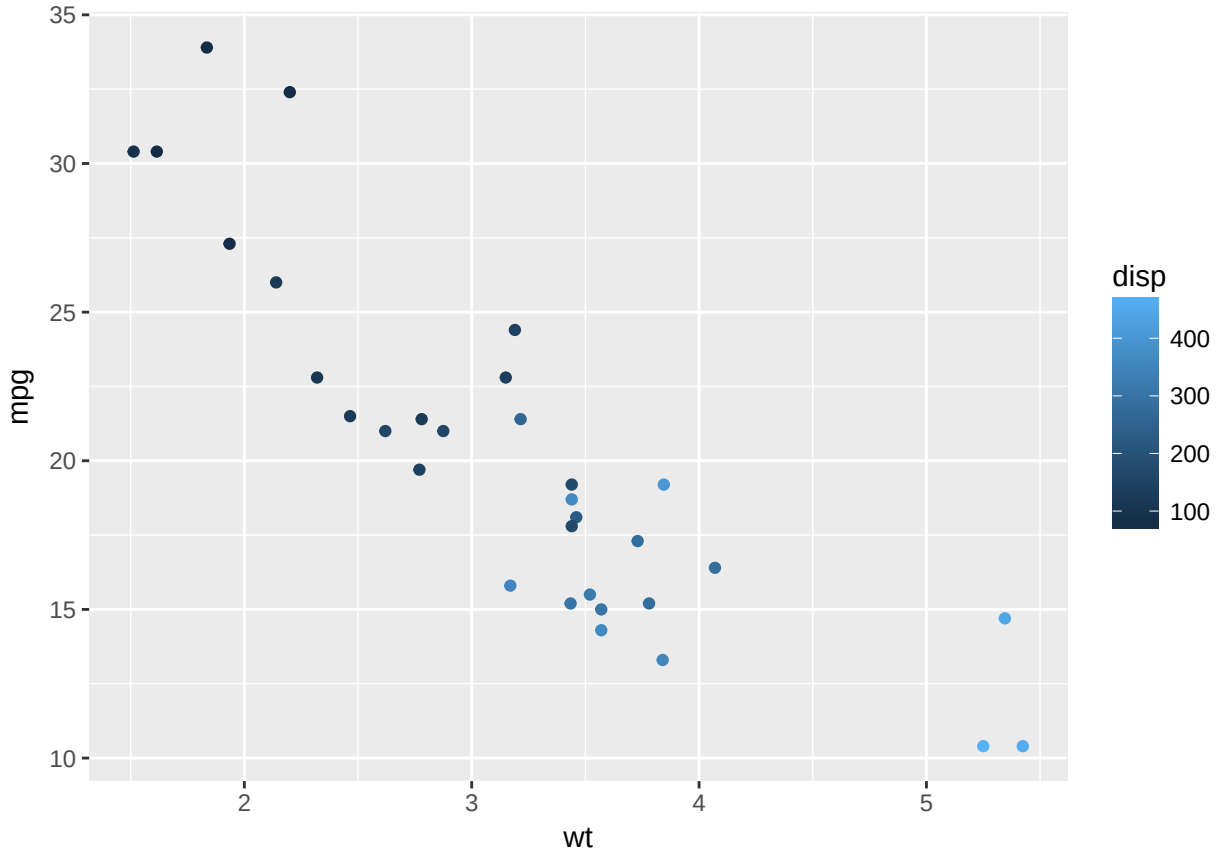
In the second call of `ggplot()` change the `color` argument in `aes()` (which stands for aesthetics). The color should be dependent on the displacement of the car engine, found in `disp`.

In the third call of `ggplot()` change the size argument in `aes()` (which stands for aesthetics). The size should be dependent on the displacement of the car engine, found in `disp`.

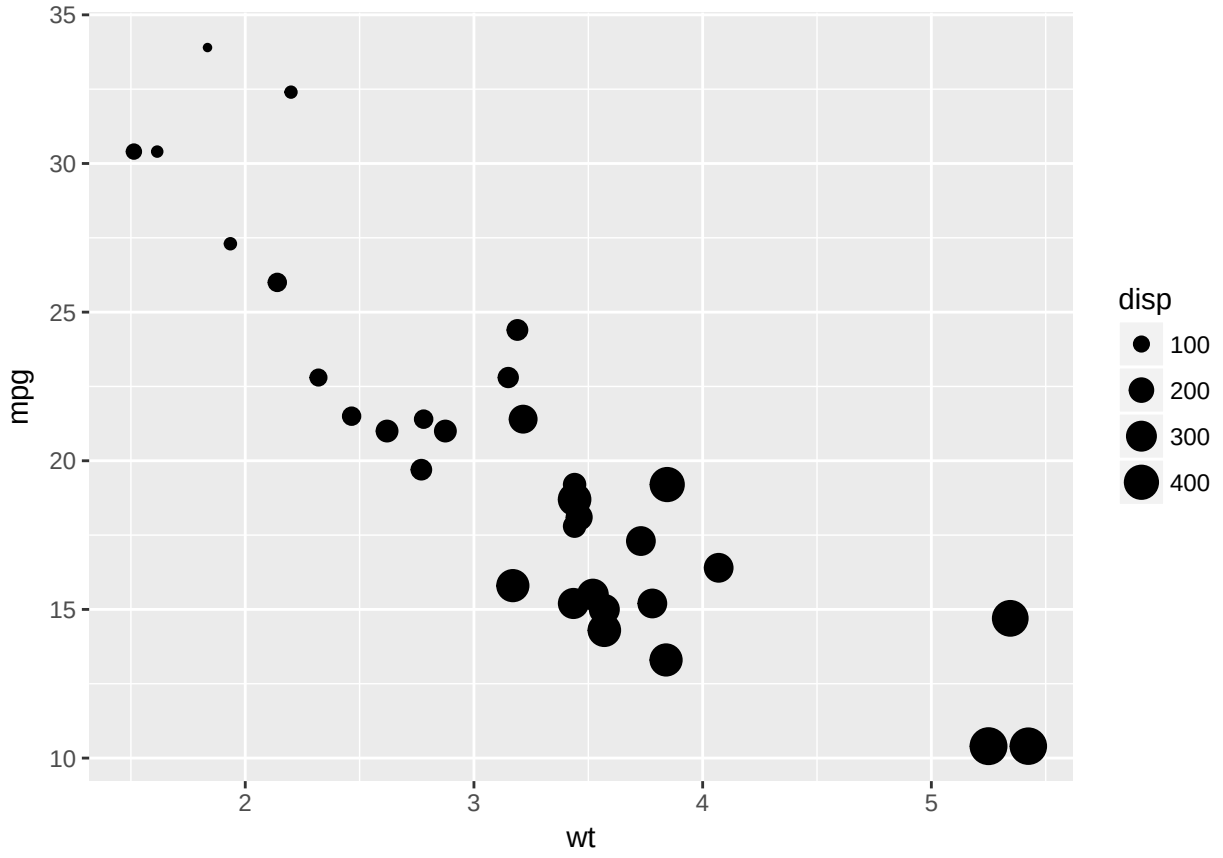
```r
# A scatter plot has been made for you
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```




```r
# Replace ___ with the correct column
ggplot(mtcars, aes(x = wt, y = mpg, color = disp)) +
  geom_point()
```




```
# Replace ___ with the correct column
ggplot(mtcars, aes(x = wt, y = mpg, size = disp)) +
  geom_point()
```



Understanding Variables

In the previous exercise you saw that `disp` can be mapped onto a color gradient or onto a continuous size scale.

Another argument of `aes()` is the shape of the points. There are a finite number of shapes which `ggplot()` can automatically assign to the points. However, if you try this command in the console to the right:

```
ggplot(mtcars, aes(x = wt, y = mpg, shape = disp)) +
  geom_point()
```

```
## Error: A continuous variable can not be mapped to shape
```

It gives an error. What does this mean?

INSTRUCTIONS

Possible Answers (Correct Answer is **Bolded**)

`shape` is not a defined argument.

`shape` only makes sense with categorical data, and `disp` is continuous.

`shape` only makes sense with continuous data, and `disp` is categorical.

`shape` is not a variable in your dataset.

`shape` has to be defined as a function.

Exploring ggplot2, part 4

The `diamonds` data frame contains information on the prices and various metrics of 50,000 diamonds. Among the variables included are `carat` (a measurement of the size of the diamond) and `price`. For the next exercises, you'll be using a subset of 1,000 diamonds.

Here you'll use two common geom layer functions: `geom_point()` and `geom_smooth()`. We already saw in the earlier exercises how these are added using the `+` operator.

INSTRUCTIONS

Explore the `diamonds` data frame with the `str()` function.

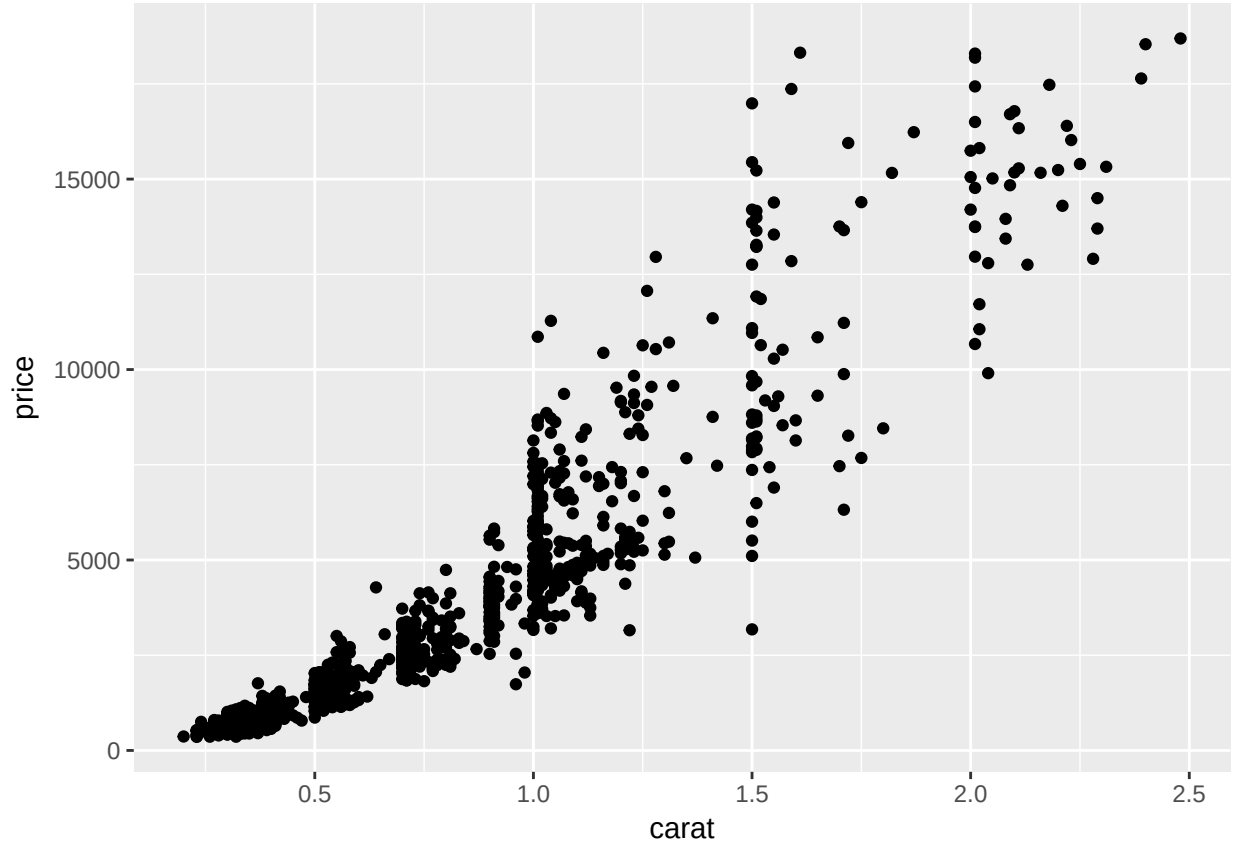
Use the `+` operator to add `geom_point()` to the first `ggplot()` command. This will tell `ggplot2` to draw points on the plot.

Use the + operator to add `geom_point()` and `geom_smooth()`. These just stack on each other! `geom_smooth()` will draw a smoothed line over the points.

```
# Explore the diamonds data frame with str()
str(diamonds)
```

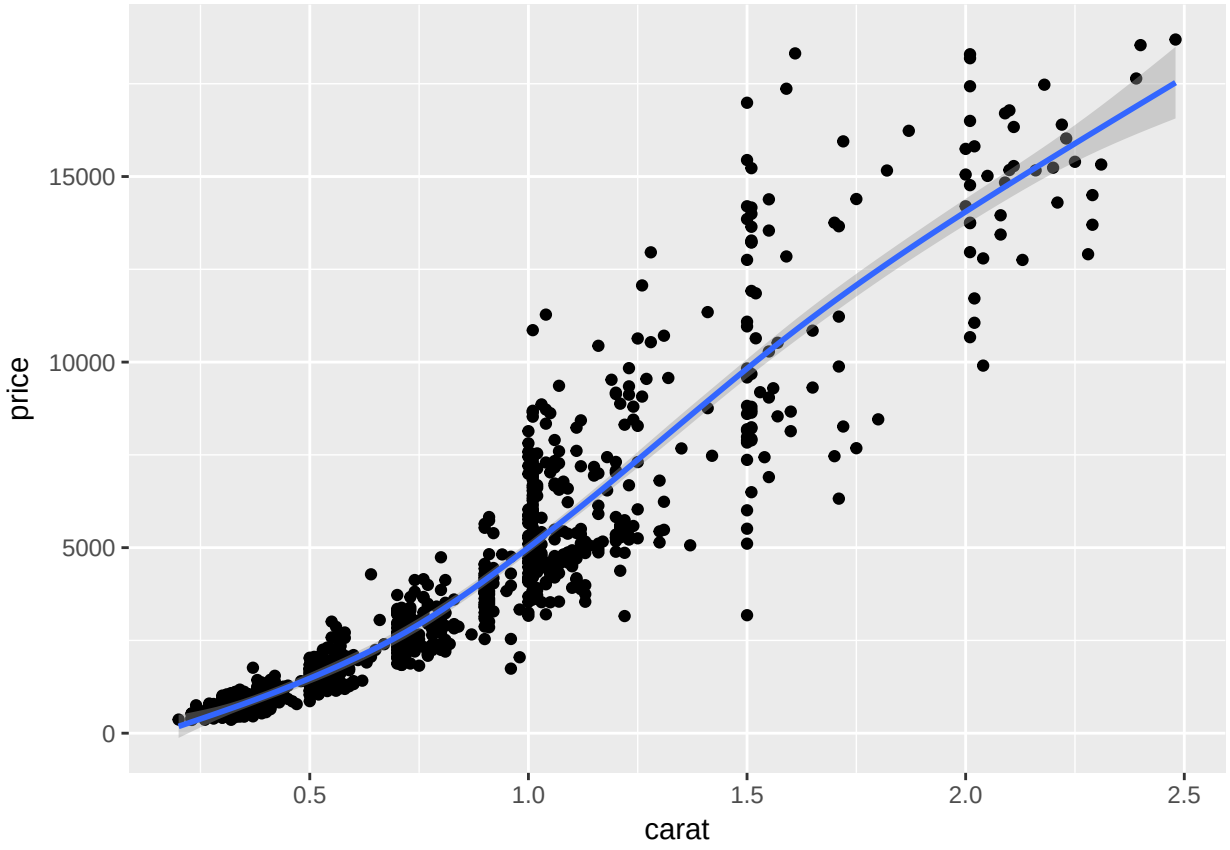
```
## Classes 'tbl_df', 'tbl' and 'data.frame':    1000 obs. of  10 variables:
##  $ carat  : num  0.31 1.5 0.9 1.01 0.33 1.08 1.07 0.33 0.44 1 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 2 4 5 3 5 4 4 5 4 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 4 4 5 3 1 4 4 5 5 4 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 5 2 4 4 5 5 3 5 8 5 ...
##  $ depth  : num  62.4 64.3 62.8 60.9 63.2 62 61.6 59.5 62 58.6 ...
##  $ table  : num  55 57 58 58 56 55 58 59 57 61 ...
##  $ price  : int  802 8190 3810 7411 1109 6779 5453 743 1255 6989 ...
##  $ x      : num  4.35 7.29 6.17 6.43 4.45 6.62 6.6 4.53 4.87 6.57 ...
##  $ y      : num  4.33 7.2 6.13 6.47 4.44 6.57 6.56 4.48 4.91 6.5 ...
##  $ z      : num  2.71 4.66 3.86 3.93 2.81 4.09 4.05 2.68 3.02 3.83 ...
```

```
# Add geom_point() with +
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point()
```




```
# Add geom_point() and geom_smooth() with +
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point() +
  geom_smooth()
```

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



Exploring ggplot2, part 5

The code for last plot of the previous exercise is available in the script on the right. It builds a scatter plot of the `diamonds` dataset, with `carat` on the x-axis and `price` on the y-axis. `geom_smooth()` is used to add a smooth line.

With this plot as a starting point, let's explore some more possibilities of combining geoms.

INSTRUCTIONS

Plot 2 - Copy and paste plot 1, but show only the smooth line, no points.

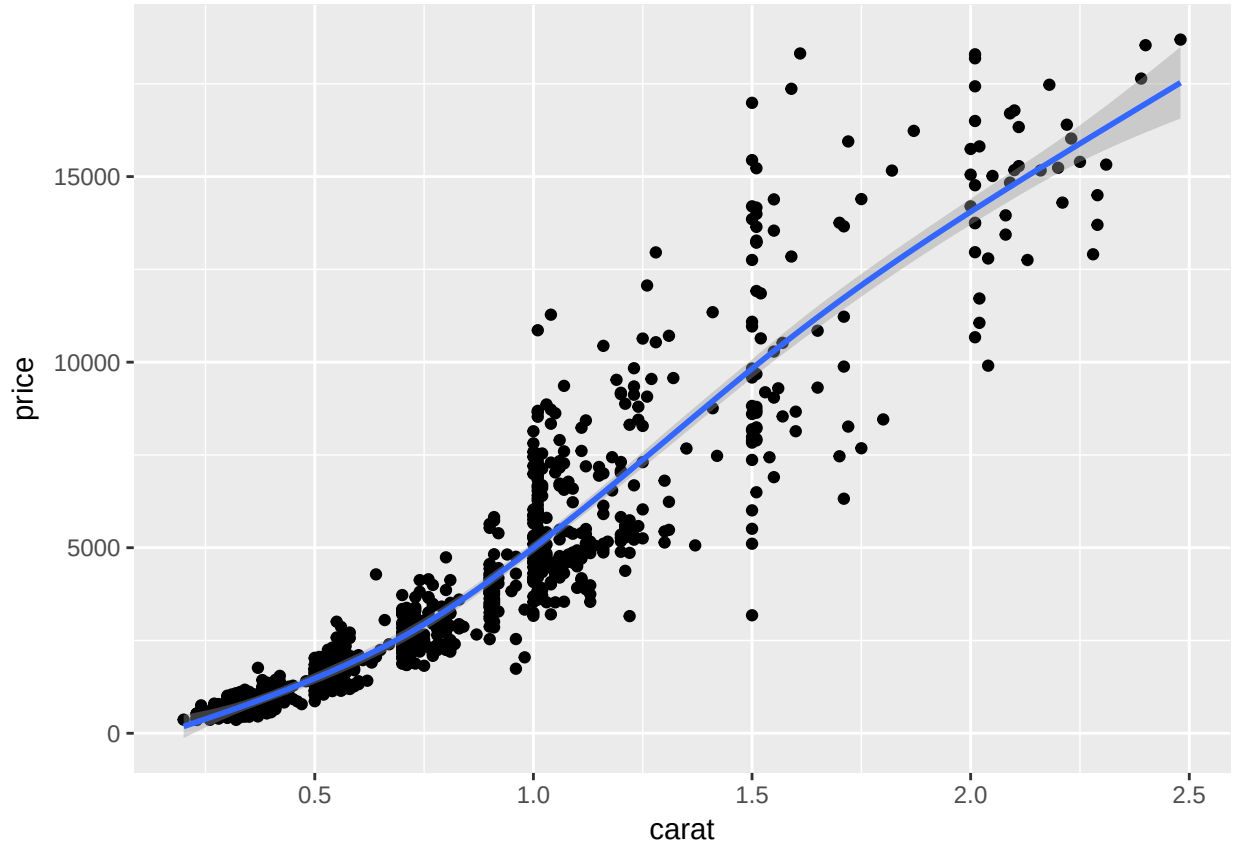
Plot 3 - Show only the smooth line, but color according to clarity by placing the argument `color = clarity` in the `aes()` function of your `ggplot()` call.

Plot 4 - Draw translucent colored points.

- Copy the `ggplot()` command from plot 3 (with clarity mapped to color).
- Remove the smooth layer.
- Add the points layer back in.
- Set `alpha = 0.4` inside `geom_point()`. This will make the points 40% transparent.

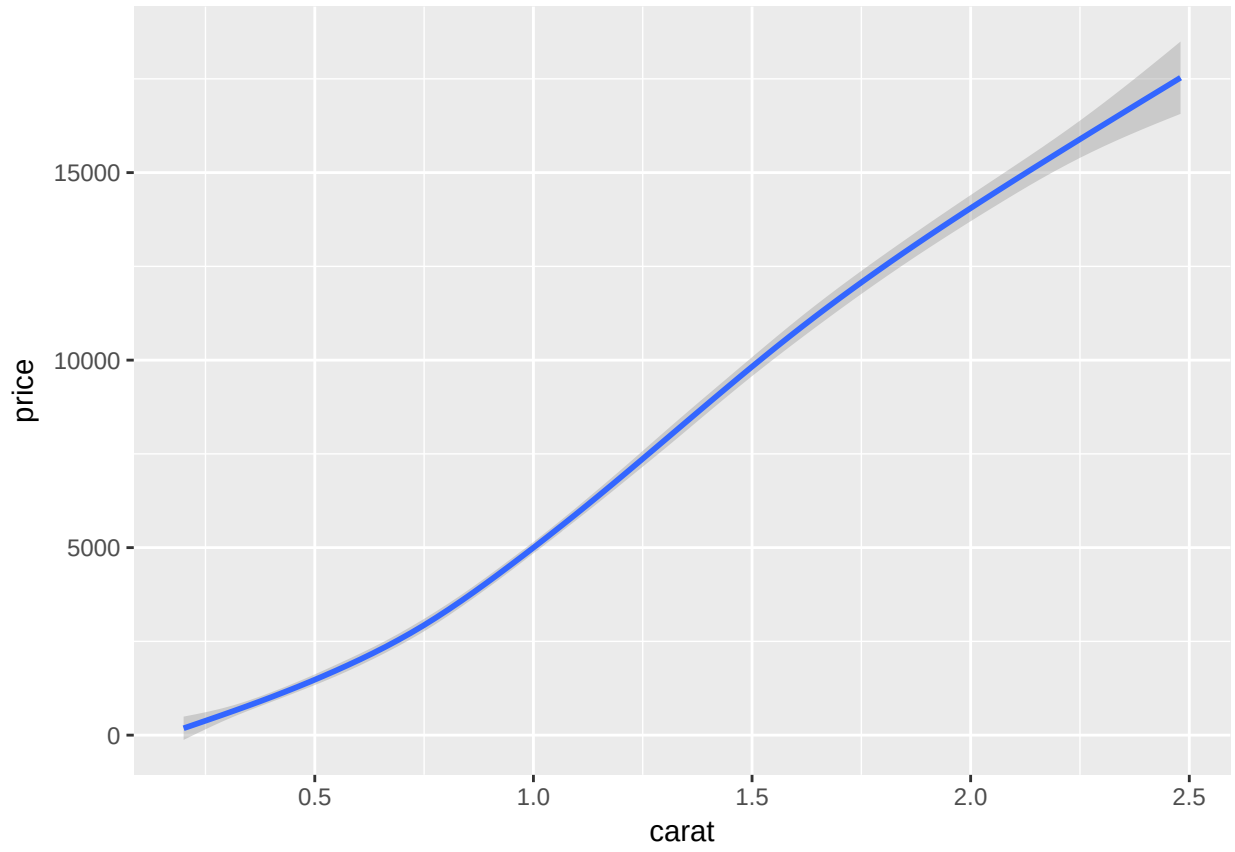
```
# 1 - The plot you created in the previous exercise
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point() +
  geom_smooth()
```

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



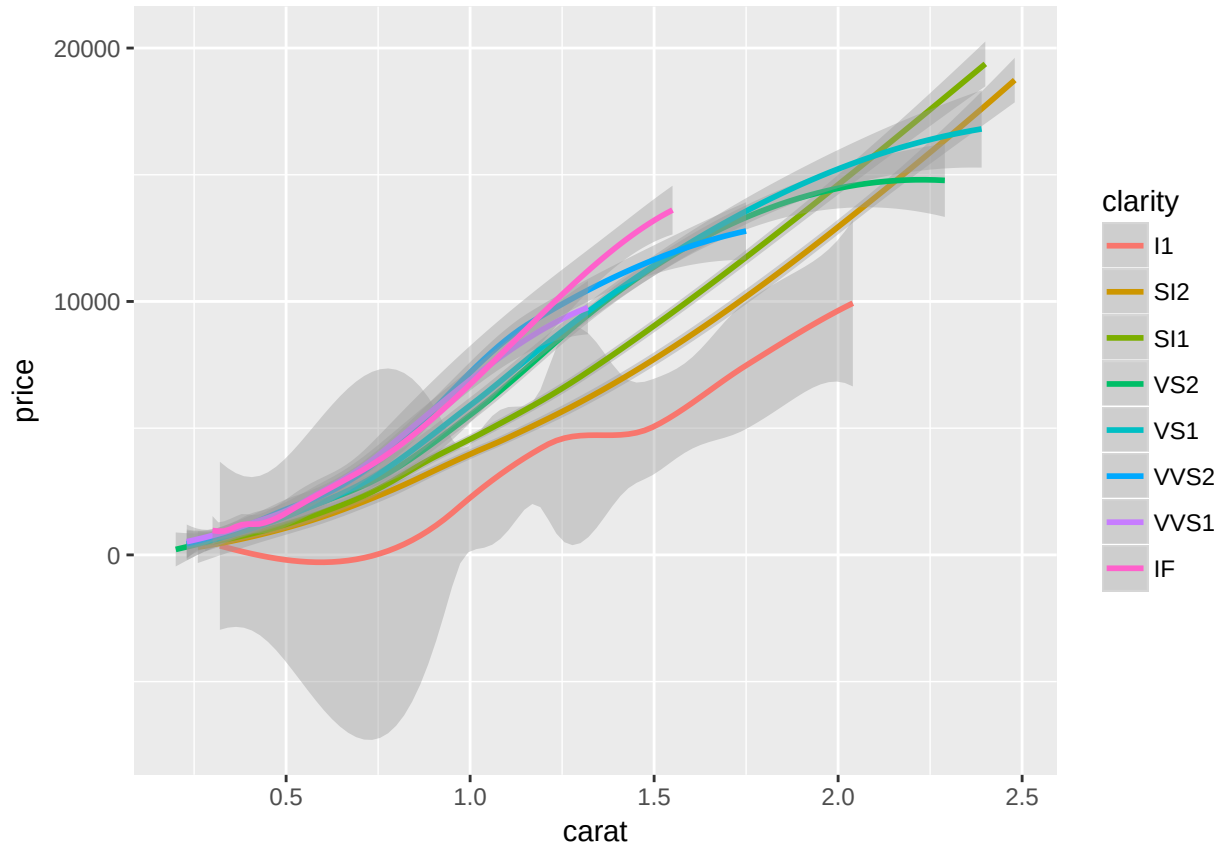

```
# 2 - Copy the above command but show only the smooth line
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_smooth()
```

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

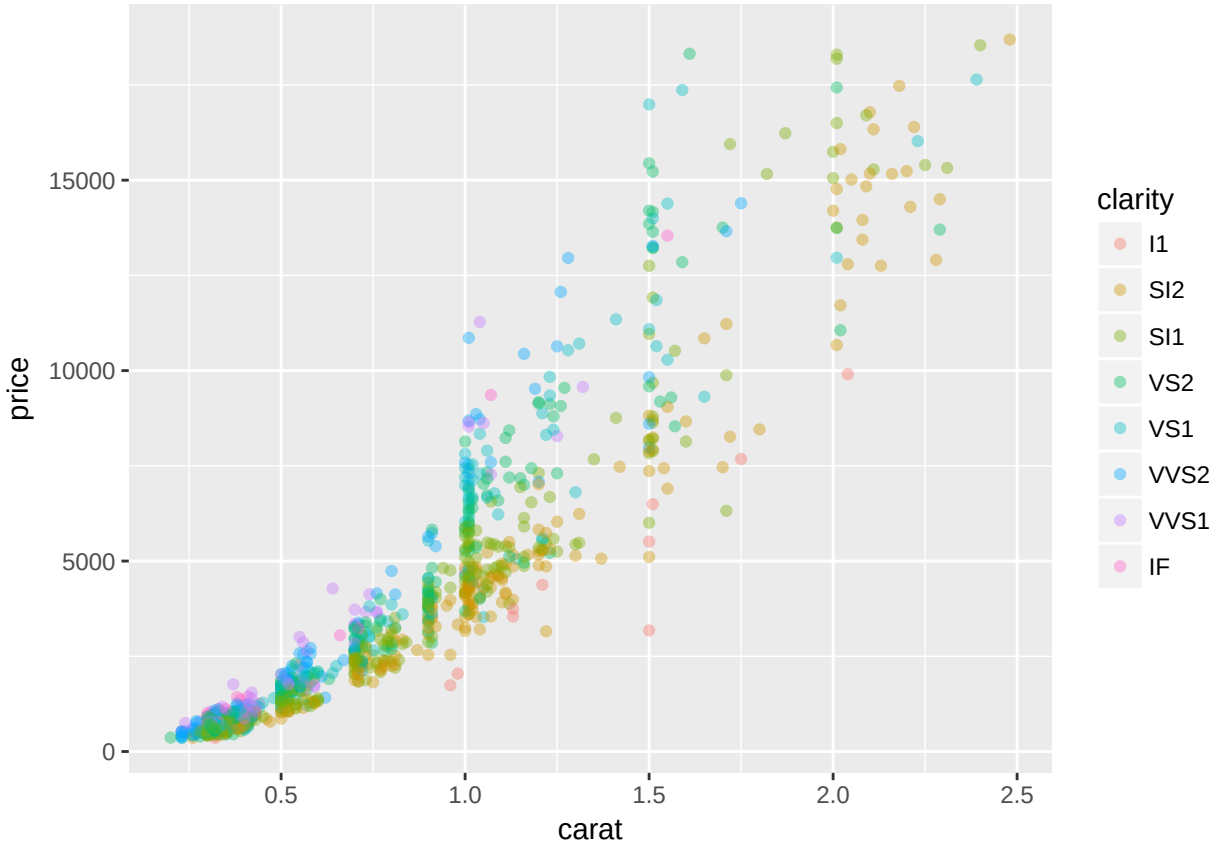



```
# 3 - Copy the above command and assign the correct value to col in aes()
ggplot(diamonds, aes(x = carat, y = price, color = clarity)) +
  geom_smooth()
```

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




```
# 4 - Keep the color settings from previous command. Plot only the points with argument alpha.
ggplot(diamonds, aes(x = carat, y = price, color = clarity)) +
  geom_point(alpha = 0.4)
```



Understanding the grammar, part 1

Here you'll explore some of the different grammatical elements. Throughout this course, you'll discover how they can be combined in all sorts of ways to develop unique plots.

In the following instructions, you'll start by creating a `ggplot` object from the `diamonds` dataset. Next, you'll add layers onto this object to build beautiful & informative plots.

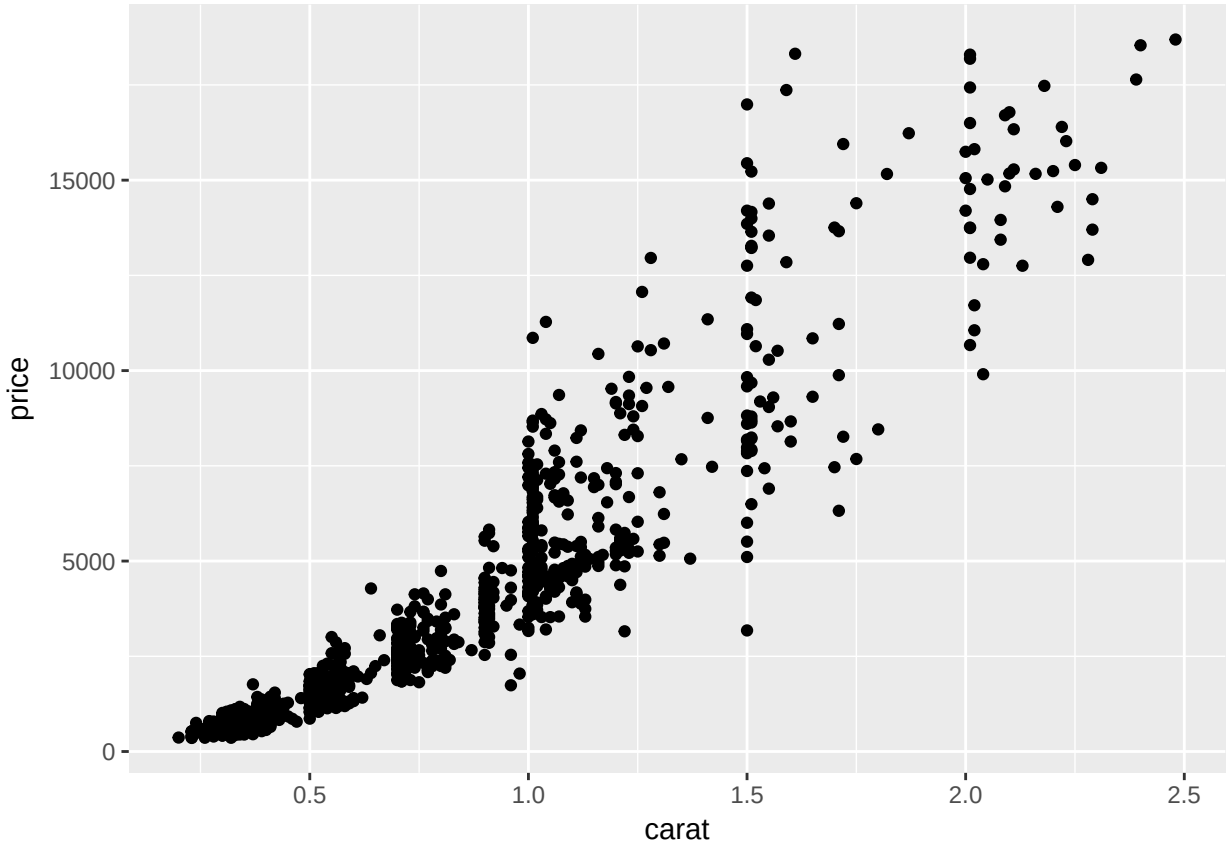
INSTRUCTIONS

Define the data (`diamonds`) and aesthetics layers. Map `carat` on the x axis and `price` on the y axis. Assign it to an object: `dia_plot`.

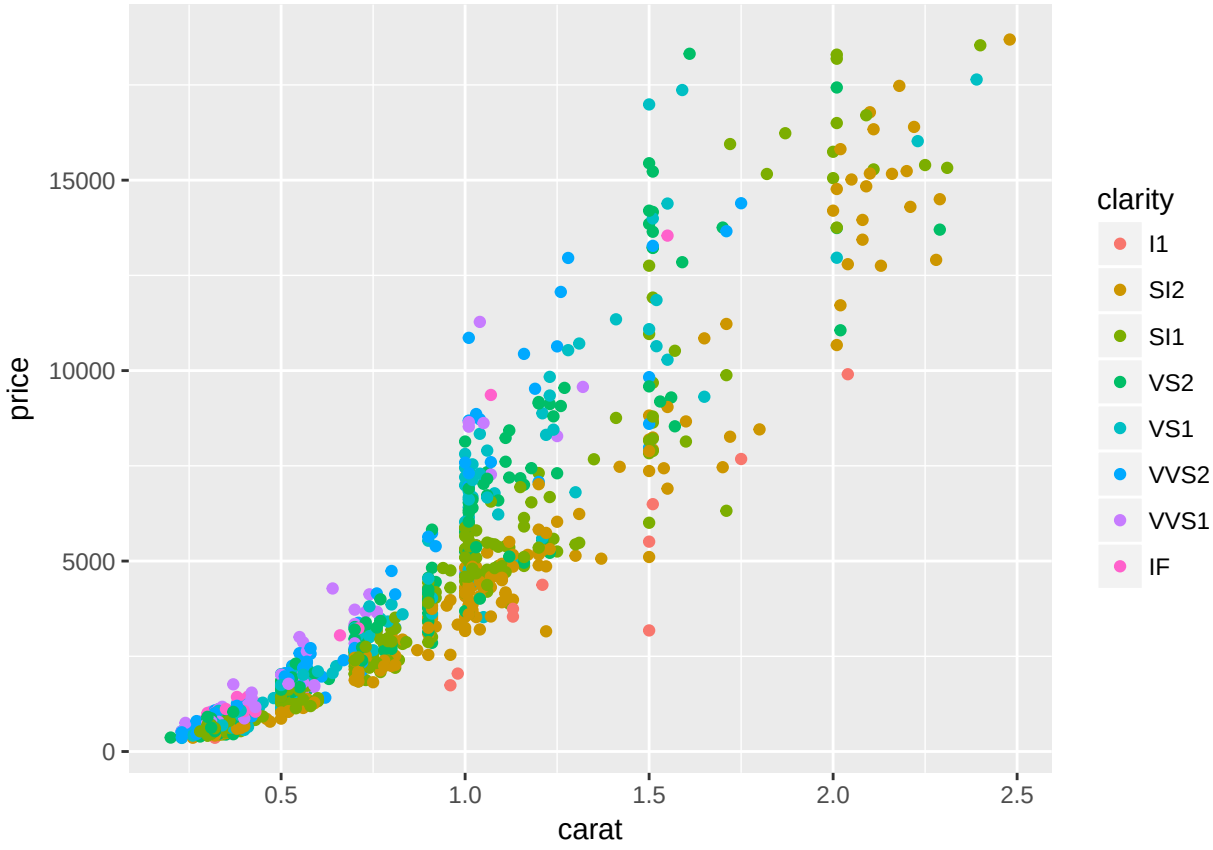
Using `+`, add a `geom_point()` layer (with no arguments), to the `dia_plot` object. This can be in a single or multiple lines.

Note that you can also call `aes()` within the `geom_point()` function. Map `clarity` to the `color` argument in this way.

```
# Create the object containing the data and aes layers: dia_plot
dia_plot <- ggplot(diamonds, aes(x = carat, y = price))

# Add a geom layer with + and geom_point()
dia_plot + geom_point()
```




```
# Add the same geom layer, but with aes() inside
dia_plot + geom_point(aes(color = clarity))
```



Understanding the grammar, part 2

Continuing with the previous exercise, here you'll explore mixing arguments and aesthetics in a single geometry.

You're still working on the `diamonds` dataset.

INSTRUCTIONS

1 - The `dia_plot` object has been created for you.

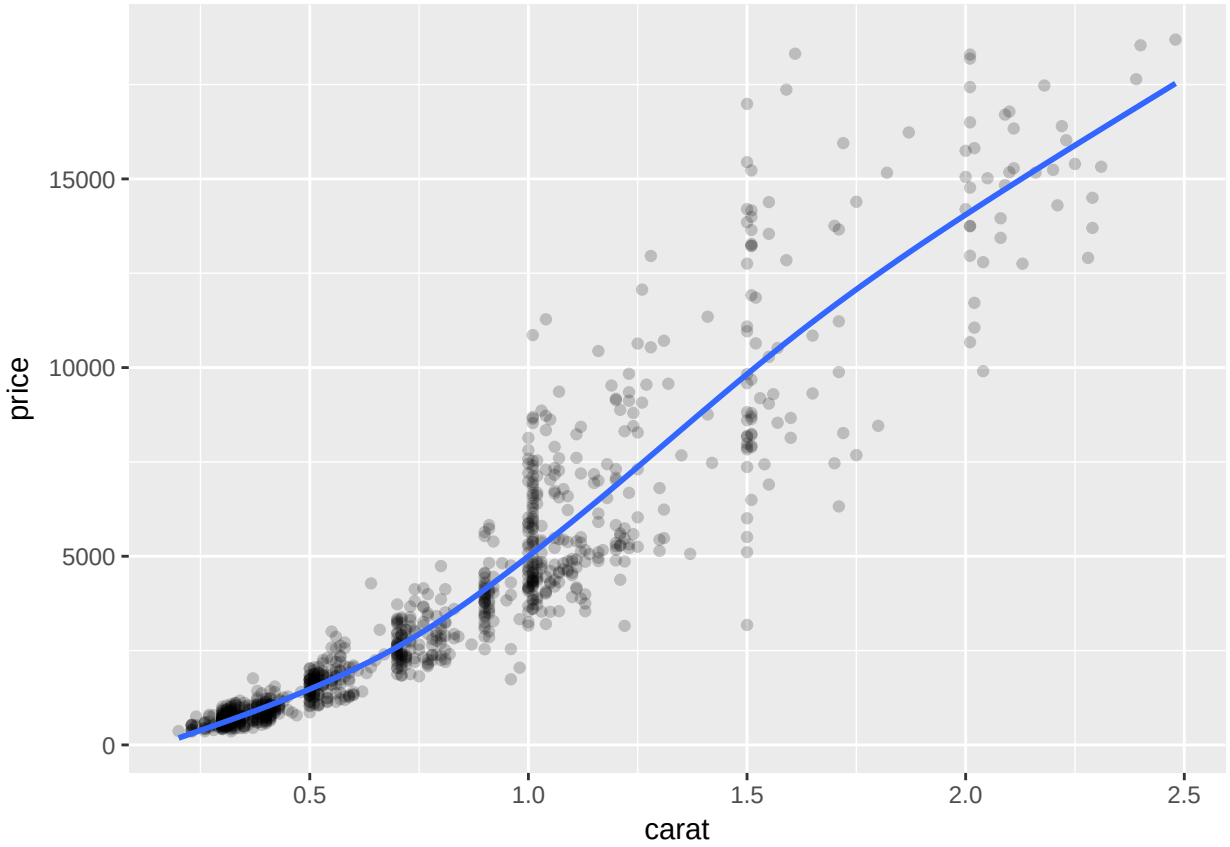
2 - Update `dia_plot` so that it contains all the functions to make a scatter plot by using `geom_point()` for the geom layer. Set `alpha = 0.2`.

3 - Using `+`, plot the `dia_plot` object with a `geom_smooth()` layer on top. You don't want any error shading, which can be achieved by setting the `se = FALSE` in `geom_smooth()`.

4 - Modify the `geom_smooth()` function from the previous instruction so that it contains `aes()` and map `clarity` to the `col` argument.

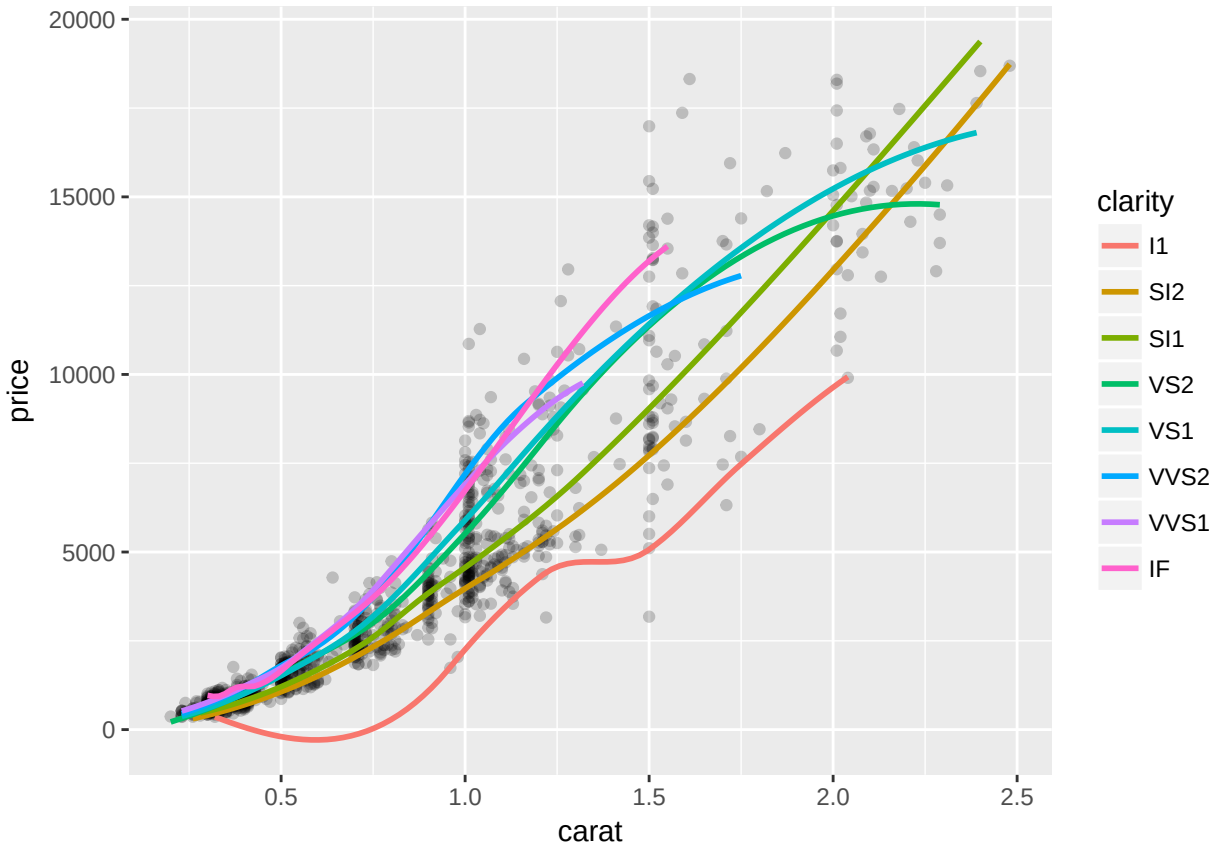
```
# 1 - The dia_plot object has been created for you
dia_plot <- ggplot(diamonds, aes(x = carat, y = price))

# 2 - Expand dia_plot by adding geom_point() with alpha set to 0.2
dia_plot <- dia_plot + geom_point(alpha = 0.2)

# 3 - Plot dia_plot with additional geom_smooth() with se set to FALSE
dia_plot + geom_smooth(se = FALSE)
```

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




```
# 4 - Copy the command from above and add aes() with the correct mapping to geom_smooth()
dia_plot + geom_smooth(aes(col = clarity), se = FALSE)
```

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



Chapter 2: Data

The structure of your data will dictate how you construct plots in ggplot2. In this chapter, you'll explore the iris dataset from several different perspectives to showcase this concept. You'll see that making your data conform to a structure that matches the plot in mind will make the task of visualization much easier through several R data visualization examples.

base package and ggplot2, part 1 - plot

These courses are about understanding data visualization in the context of the grammar of graphics. To gain a better appreciation of `ggplot2` and to understand how it operates differently from base package, it's useful to make some comparisons.

In the video, you already saw one example of how to make a (poor) multivariate plot in base package. In this series of exercises you'll take a look at a better way using the equivalent version in ggplot2.

First, let's focus on base package. You want to make a plot of `mpg` (miles per gallon) against `wt` (weight in thousands of pounds) in the `mtcars` data frame, but this time you want the dots colored according to the number of cylinders, `cyl`. How would you do that in base package? You can use a little trick to color the dots by specifying a `factor` variable as a color. This works because factors are just a special class of the `integer` type.

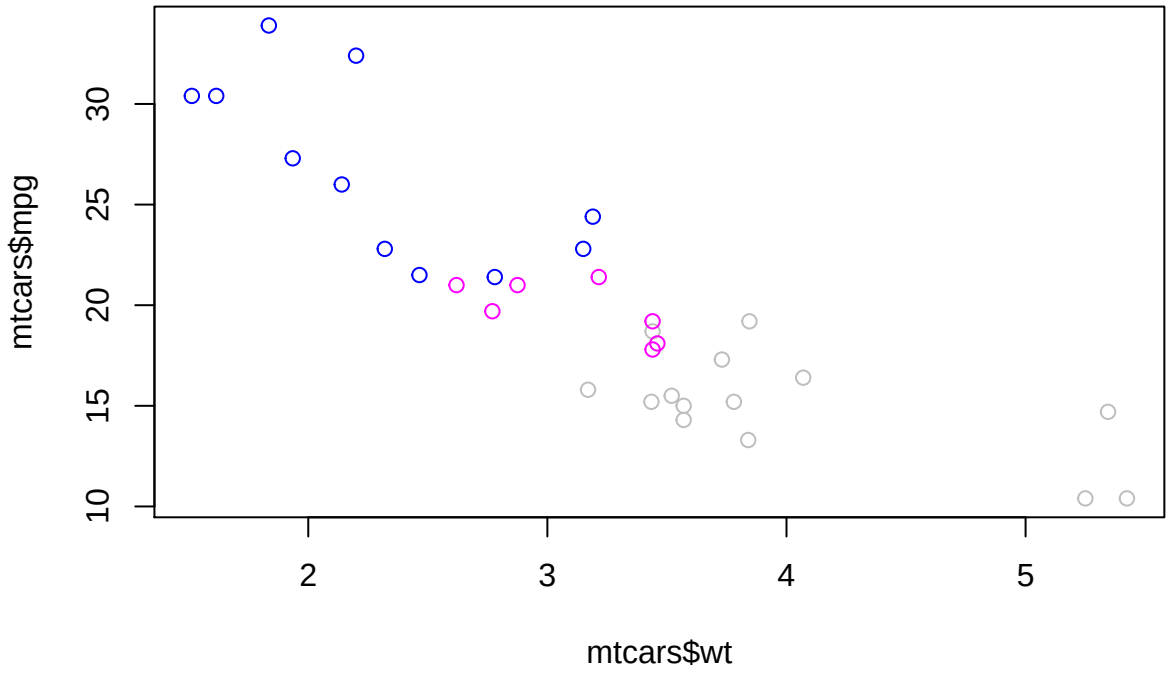
INSTRUCTIONS

Using the base package `plot()`, make a scatter plot with `mtcars$wt` on the x-axis and `mtcars$mpg` on the y-axis, colored according to `mtcars$cyl` (use the col argument). You can specify `data =` but you'll just do it the long way here.

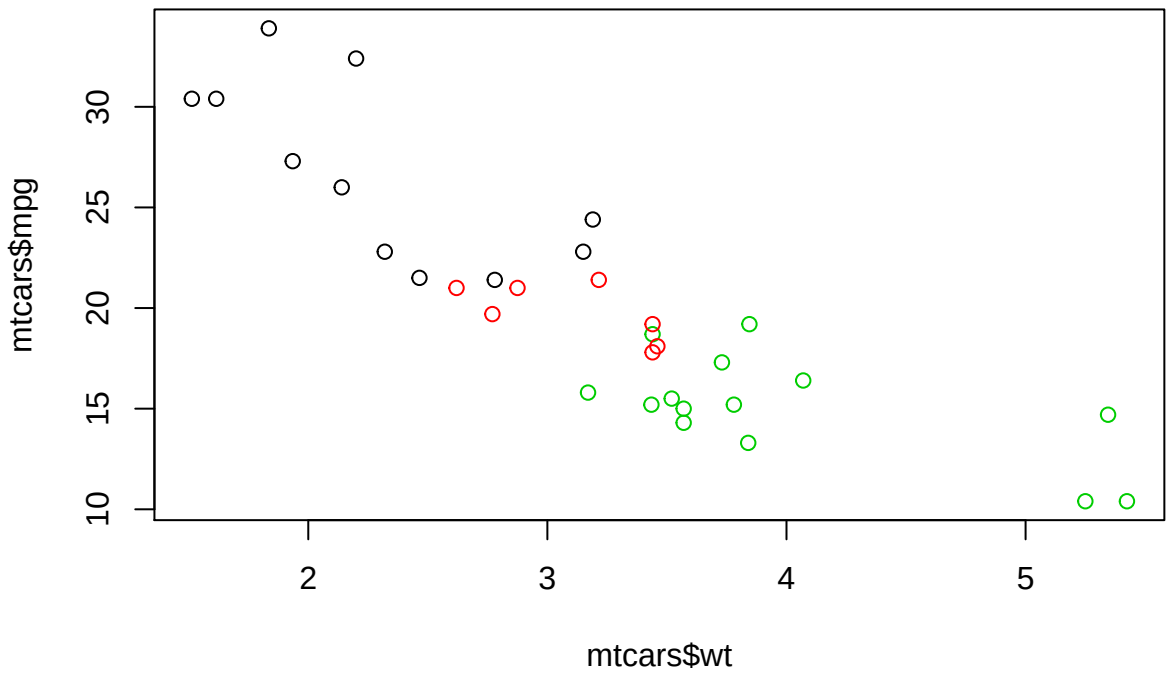
Add a new column, `fcyl`, to the mtcars data frame. This should be `cyl` converted to a factor.

Create a similar plot to instruction 1, but this time, use `fcyl` (which is cyl as a factor) to set the col.

```
# Plot the correct variables of mtcars
plot(mtcars$wt, mtcars$mpg, col = mtcars$cyl)
```




```
# Change cyl inside mtcars to a factor
mtcars$fcyl <- as.factor(mtcars$cyl)

# Make the same plot as in the first instruction
plot(mtcars$wt, mtcars$mpg, col = mtcars$fcyl)
```



base package and ggplot2, part 2 - lm

If you want to add a linear model to your plot, shown right, you can define it with `lm()` and then plot the resulting linear model with `abline()`. However, if you want a model for each subgroup, according to cylinders, then you have a couple of options.

You can subset your data, and then calculate the `lm()` and plot each subset separately. Alternatively, you can vectorize over the `cyl` variable using `lapply()` and combine this all in one step. This option is already prepared for you.

The code to the right contains a call to the function `lapply()`, which you might not have seen before. This function takes as input a vector and a function. Then `lapply()` applies the function it was given to each element of the vector and returns the results in a list. In this case, `lapply()` takes each element of `mtcars$cyl` and calls the function defined in the second argument. This function takes a value of `mtcars$cyl` and then subsets the data so that only rows with `cyl == x` are used. Then it fits a linear model to the filtered dataset and uses that model to add a line to the plot with the `abline()` function.

Now that you have an interesting plot, there is a very important aspect missing - the legend!

In base package you have to take care of this using the `legend()` function. This has been done for you in the predefined code.

INSTRUCTIONS

Fill in the `lm()` function to calculate a linear model of `mpg` described by `wt` and save it as an object called `carModel`.

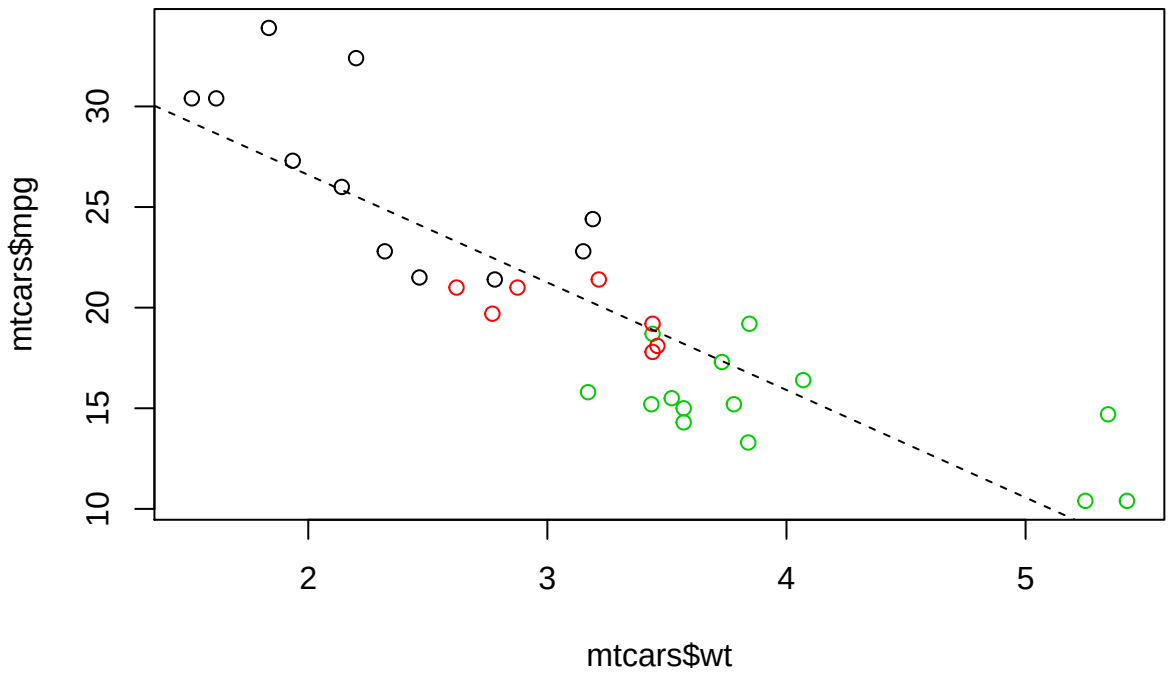
Draw the linear model on the scatterplot.

Write code that calls `abline()` with `carModel` as the first argument. Set the line type by passing the argument `lty = 2`.

Run the code that generates the basic plot and the call to `abline()` all at once by highlighting both parts of the script and hitting `control/command` + `enter` on your keyboard. These lines must all be run together in the DataCamp R console so that R will be able to find the plot you want to add a line to.

Run the code already given to generate the plot with a different model for each group. You don't need to modify any of this.

```
# Use lm() to calculate a linear model and save it as carModel
carModel <- lm(mpg ~ wt, data = mtcars)

# Basic plot
mtcars$cyl <- as.factor(mtcars$cyl)
plot(mtcars$wt, mtcars$mpg, col = mtcars$cyl)

# Call abline() with carModel as first argument and set lty to 2
abline(carModel, lty = 2)
```



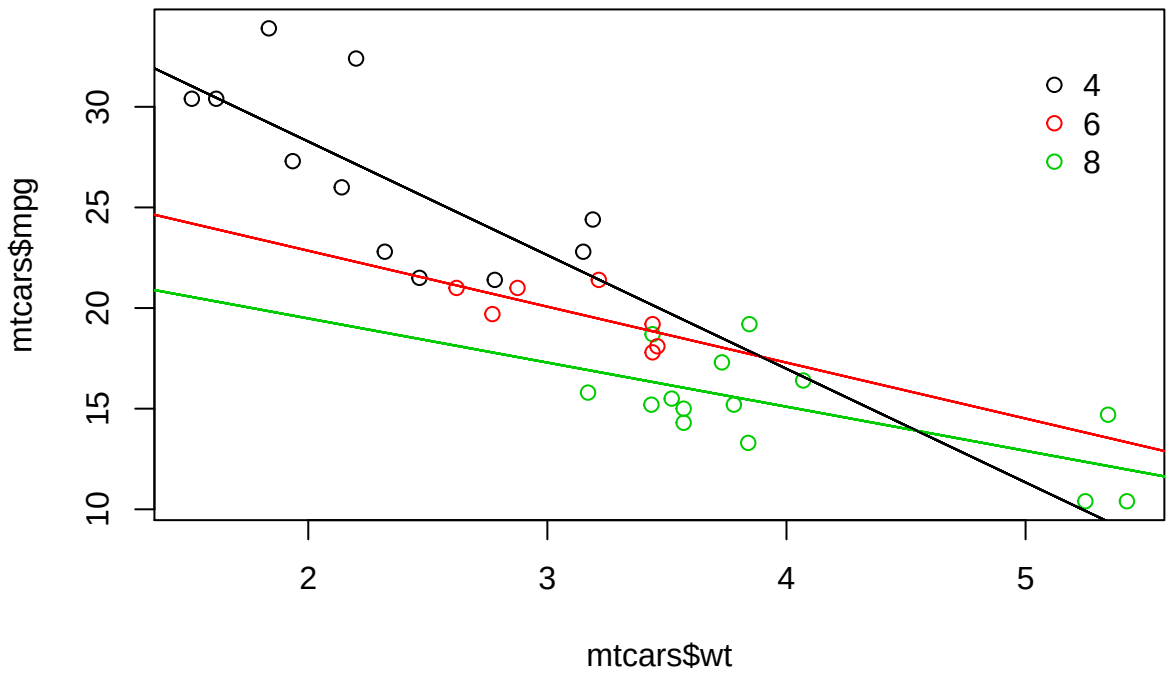

```
# Plot each subset efficiently with lapply
# You don't have to edit this code
plot(mtcars$wt, mtcars$mpg, col = mtcars$cyl)
lapply(mtcars$cyl, function(x) {
  abline(lm(mpg ~ wt, mtcars, subset = (cyl == x)), col = x)
  })
```

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

```
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
##
## [[7]]
## NULL
##
## [[8]]
## NULL
##
## [[9]]
## NULL
##
## [[10]]
## NULL
##
## [[11]]
## NULL
##
## [[12]]
## NULL
##
## [[13]]
## NULL
##
## [[14]]
## NULL
##
## [[15]]
## NULL
##
## [[16]]
## NULL
##
## [[17]]
## NULL
##
## [[18]]
## NULL
##
## [[19]]
```

```
## NULL
##
## [[20]]
## NULL
##
## [[21]]
## NULL
##
## [[22]]
## NULL
##
## [[23]]
## NULL
##
## [[24]]
## NULL
##
## [[25]]
## NULL
##
## [[26]]
## NULL
##
## [[27]]
## NULL
##
## [[28]]
## NULL
##
## [[29]]
## NULL
##
## [[30]]
## NULL
##
## [[31]]
## NULL
##
## [[32]]
## NULL
```

```r
# This code will draw the legend of the plot
# You don't have to edit this code
legend(x = 5, y = 33, legend = levels(mtcars$cyl),
       col = 1:3, pch = 1, bty = "n")
```



base package and ggplot2, part 3

In this exercise you'll recreate the base package plot in `ggplot2`.

The code for base `R` plotting is given at the top. The first line of code already converts the `cyl` variable of `mtcars` to a factor.

INSTRUCTIONS

Plot 1: add `geom_point()` in order to make a scatter plot.

Plot 2: copy and paste Plot 1

Add a linear model for each subset according to `cyl` by adding a `geom_smooth()` layer

Inside this `geom_smooth()`, set `method` to `"lm"` and `se` to `FALSE`.

Note: `geom_smooth()` will automatically draw a line per `cyl` subset. It recognizes the groups you want to identify by color in the `aes()` call within the `ggplot()` command.

Plot 3: copy and paste Plot 2

Plot a linear model for the entire dataset, do this by adding another `geom_smooth()` layer

Set the group aesthetic inside this `geom_smooth()` layer to 1. This has to be set within the `aes()` function.

Set method to `"lm"`, `se` to `FALSE` and `linetype` to 2. These have to be set outside `aes()` of the `geom_smooth()`.

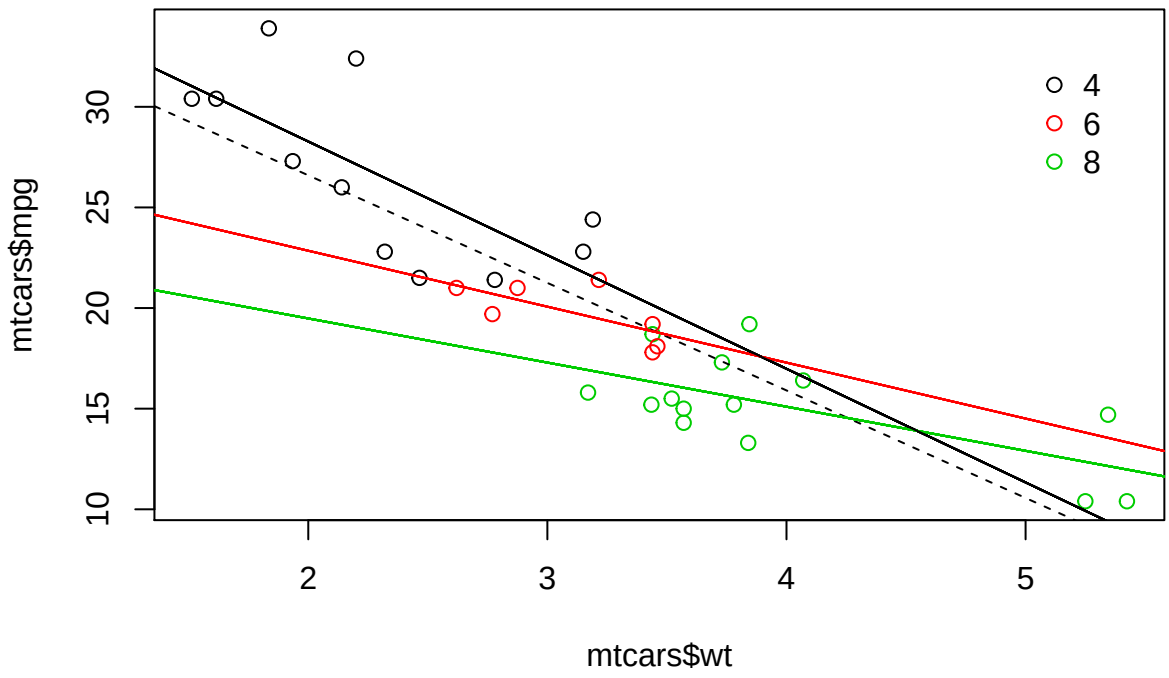
Note: the group aesthetic will tell `ggplot()` to draw a single linear model through all the points.

```r
# Convert cyl to factor (don't need to change)
mtcars$cyl <- as.factor(mtcars$cyl)

# Example from base R (don't need to change)
plot(mtcars$wt, mtcars$mpg, col = mtcars$cyl)
abline(lm(mpg ~ wt, data = mtcars), lty = 2)
lapply(mtcars$cyl, function(x) {
  abline(lm(mpg ~ wt, mtcars, subset = (cyl == x)), col = x)
  })
```

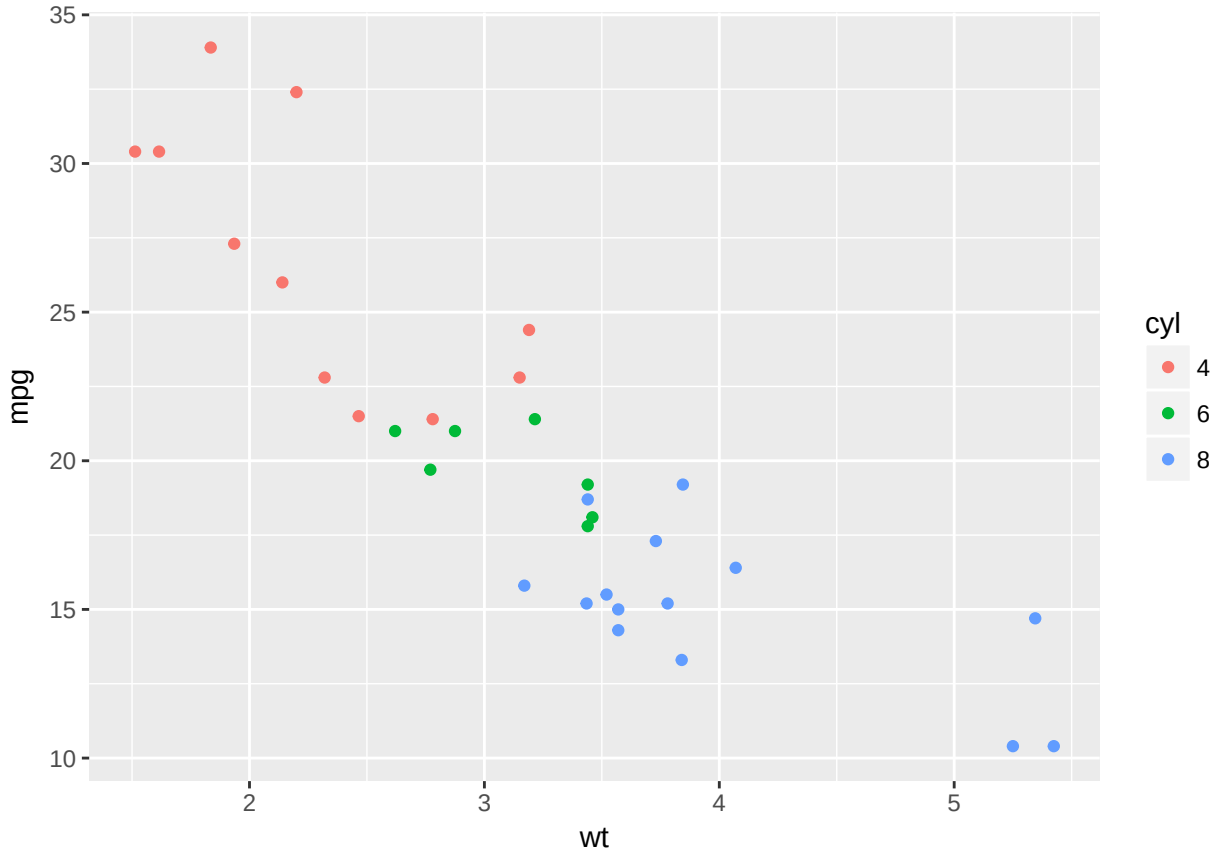
```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
##
## [[7]]
## NULL
##
## [[8]]
## NULL
##
## [[9]]
## NULL
##
## [[10]]
## NULL
##
## [[11]]
## NULL
##
## [[12]]
## NULL
##
## [[13]]
## NULL
##
## [[14]]
## NULL
##
## [[15]]
## NULL
```

```
##
## [[16]]
## NULL
##
## [[17]]
## NULL
##
## [[18]]
## NULL
##
## [[19]]
## NULL
##
## [[20]]
## NULL
##
## [[21]]
## NULL
##
## [[22]]
## NULL
##
## [[23]]
## NULL
##
## [[24]]
## NULL
##
## [[25]]
## NULL
##
## [[26]]
## NULL
##
## [[27]]
## NULL
##
## [[28]]
## NULL
##
## [[29]]
## NULL
##
## [[30]]
## NULL
##
## [[31]]
## NULL
##
## [[32]]
## NULL
```

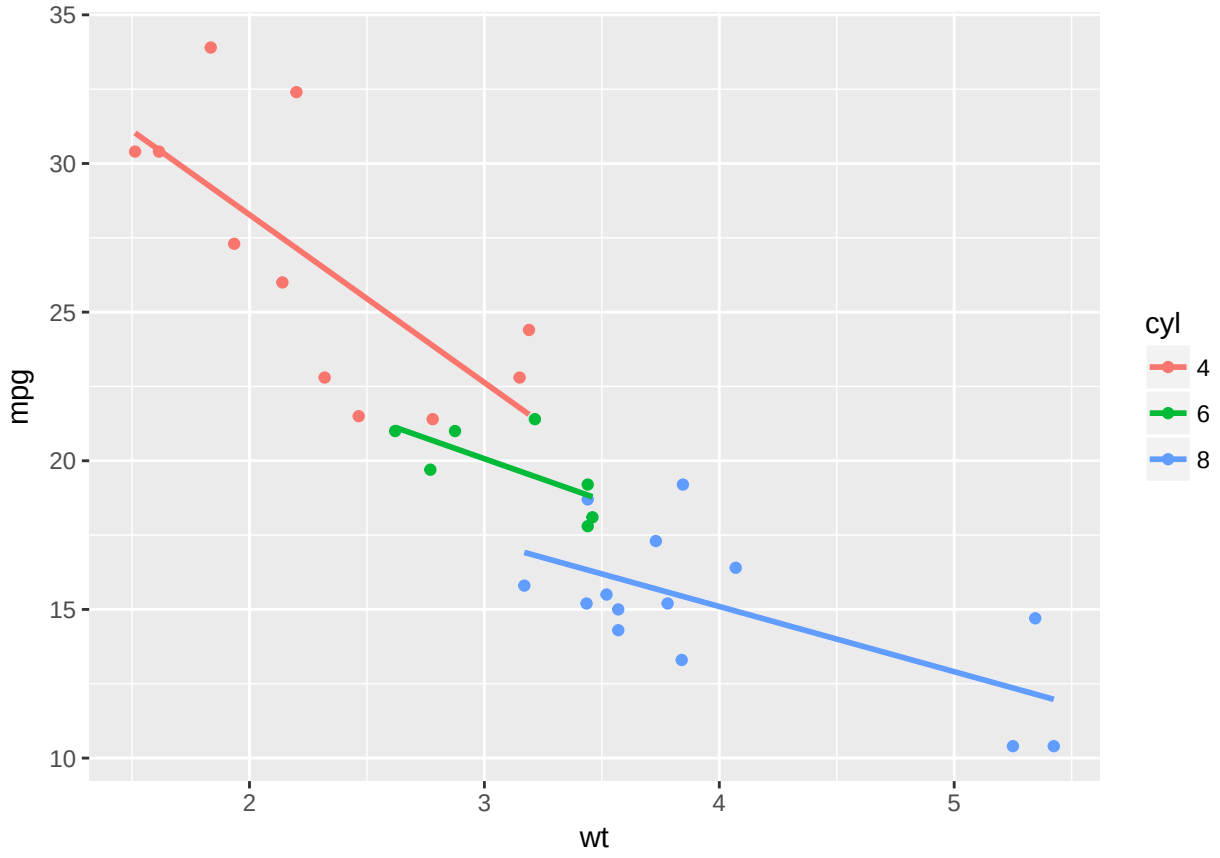
```
legend(x = 5, y = 33, legend = levels(mtcars$cyl),
       col = 1:3, pch = 1, bty = "n")
```



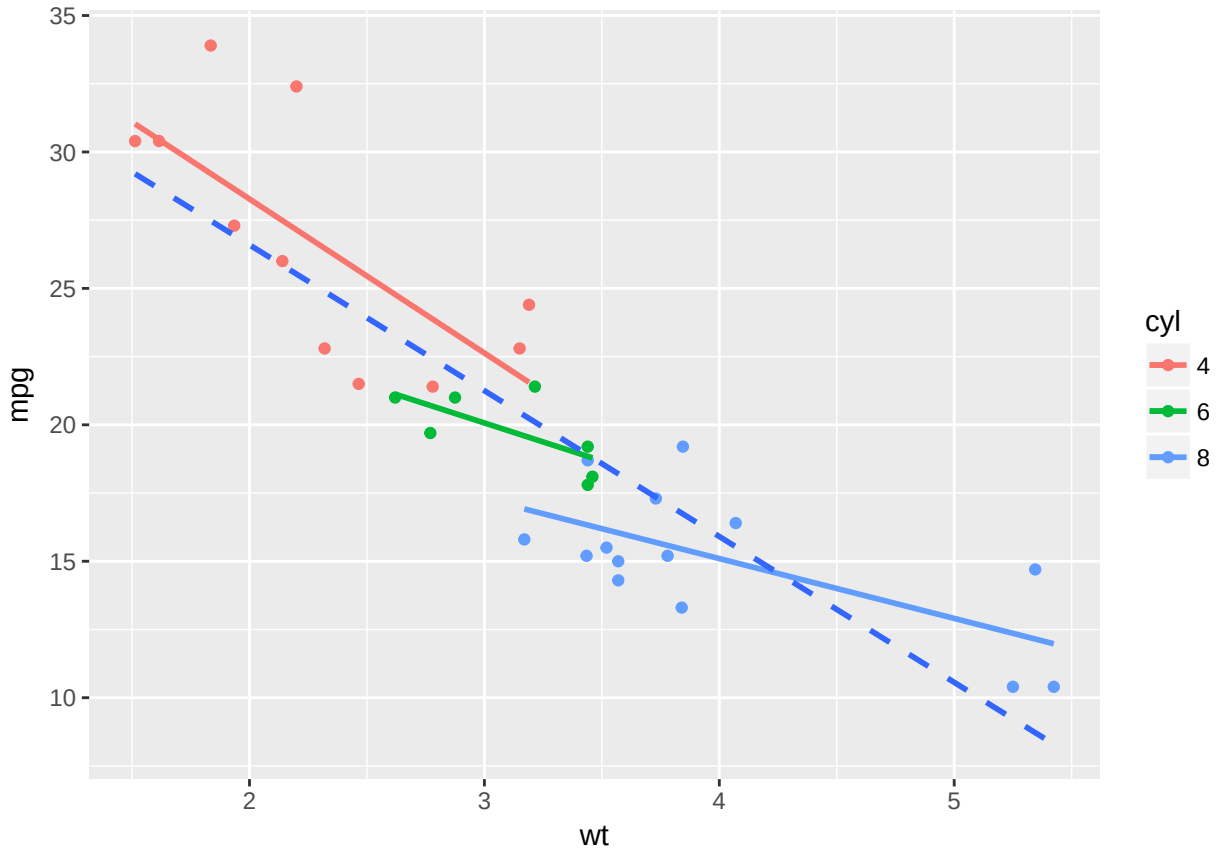

```
# Plot 1: add geom_point() to this command to create a scatter plot
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point()  # Fill in using instructions Plot 1
```




```
# Plot 2: include the lines of the linear models, per cyl
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point() + # Copy from Plot 1
  geom_smooth(method = "lm", se = FALSE)    # Fill in using instructions Plot 2
```




```
# Plot 3: include a lm for the entire dataset in its whole
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point()  + # Copy from Plot 2
  geom_smooth(method = "lm", se = FALSE) + # Copy from Plot 2
  geom_smooth(aes(group = 1), method = "lm", se = FALSE, linetype = 2)   # Fill in using instructions P
```



ggplot2 compared to base package

ggplot2 has become very popular and for many people it's the go-to plotting package in R. What does ggplot2 do that base package doesn't?

ANSWER THE QUESTION

Possible Answers (Correct answer is **Bolded**)

ggplot2 creates plotting objects, which can be manipulated.

ggplot2 takes care of a lot of the leg work for you, such as choosing nice color pallettes and making legends.

ggplot2 is built upon the grammar of graphics plotting philosophy, making it more flexible and intuitive for understanding the relationship between your visuals and your data.

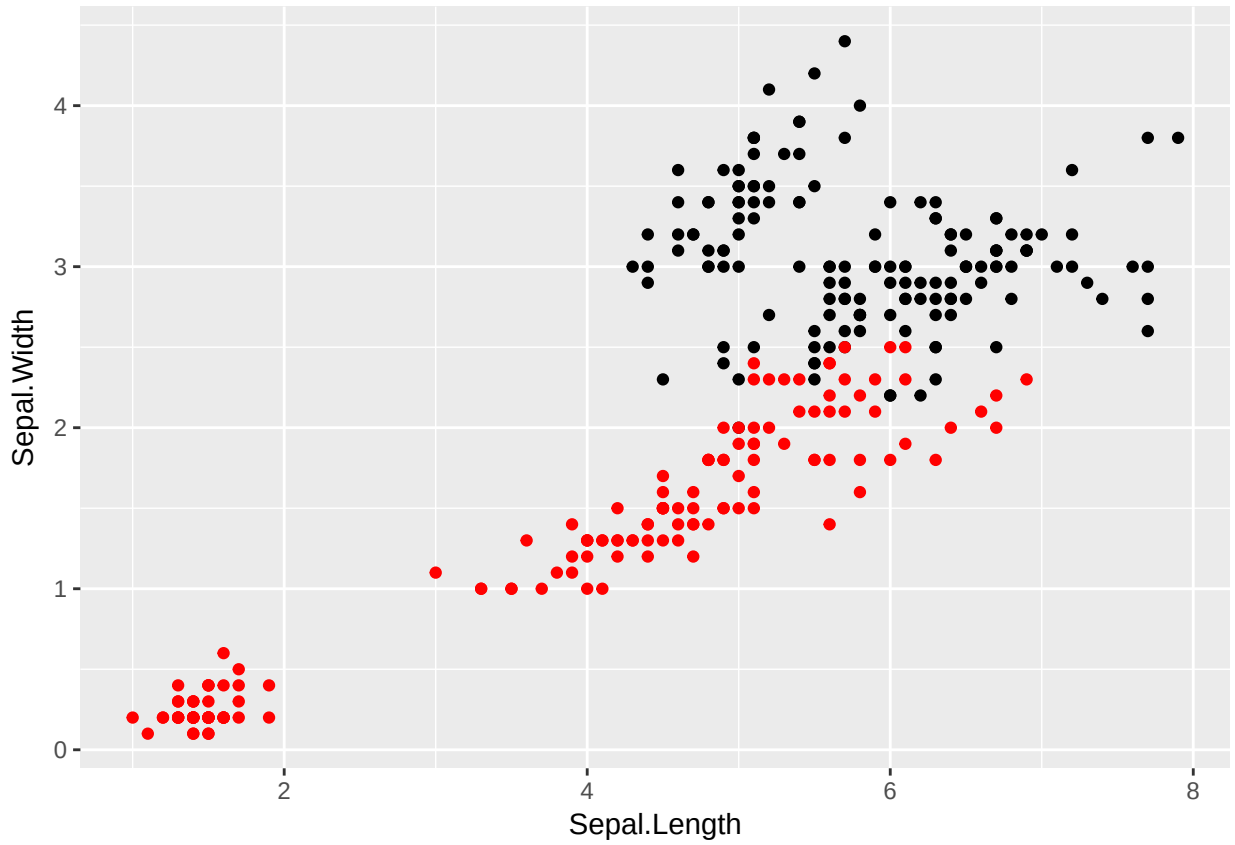
Options 1, 2, and 3.

ggplot2 is effectively a replacement for all base-package plotting functions.

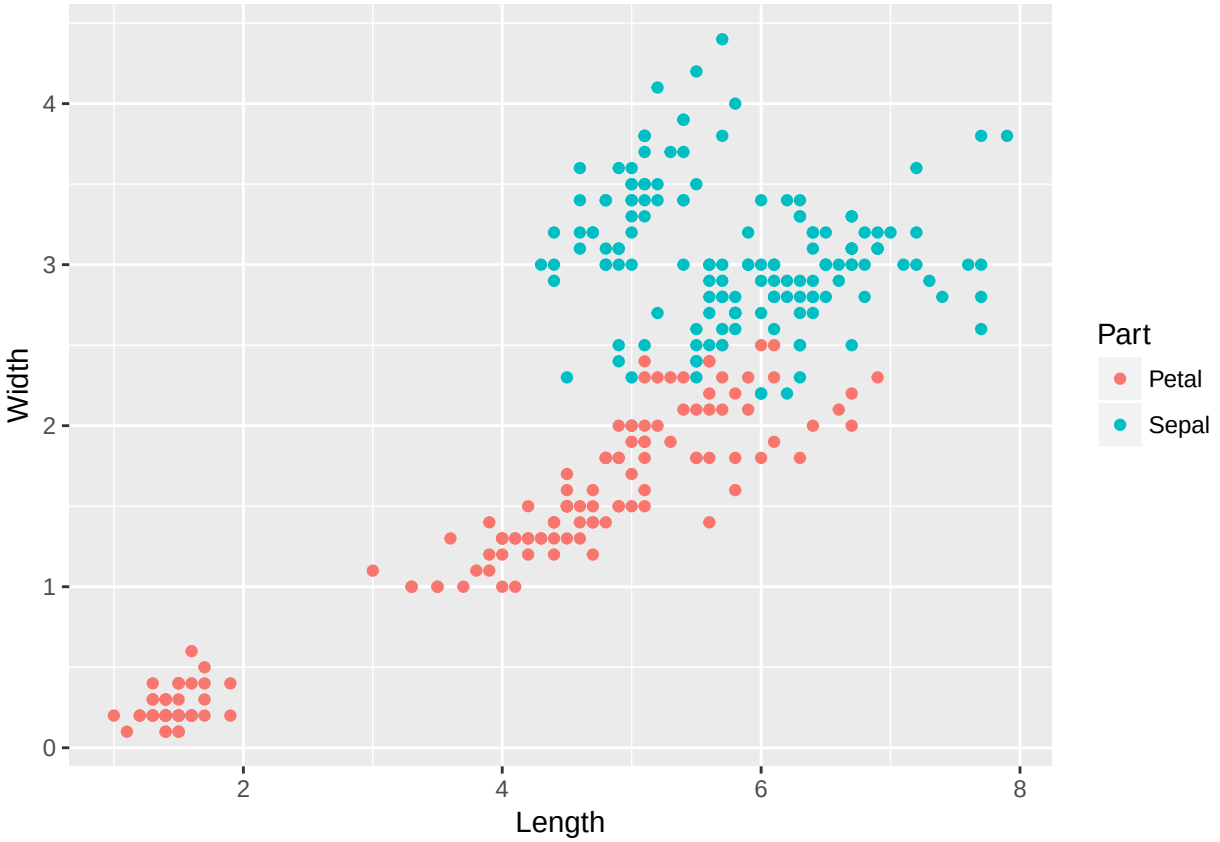
Plotting the ggplot2 way

In the video, Rick showed you different ggplot2 calls to plot two groups of data onto the same plot:

```
# Option 1
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  geom_point(aes(x = Petal.Length, y = Petal.Width), col = "red")
```




```
# Option 2
ggplot(iris.wide, aes(x = Length, y = Width, col = Part)) +
  geom_point()
```



Which one is preferable? Both iris and iris.wide are available in the workspace, so you can experiment in the R Console straight away!

Possible Answers (Correct Answer is **Bolded**)

Option 1.

Option 2.

Both are equally preferable.

Variables to visuals, part 1

So far you've seen four different forms of the iris dataset: `iris`, `iris.wide`, `iris.wide2` and `iris.tidy`. Don't let all these different forms confuse you! It's exactly the same data, just rearranged so that your plotting functions become easier.

To see this in action, consider the plot in the graphics device at right. Which form of the dataset would be the most appropriate to use here?

INSTRUCTIONS

Look at the structures of `iris`, `iris.wide` and `iris.tidy` using `str()`.

Fill in the `ggplot` function with the appropriate data frame and variable names. The variable names of the aesthetics of the plot will match the ones you found using the `str()` command in the previous step.

```
# Consider the structure of iris, iris.wide and iris.tidy (in that order)
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

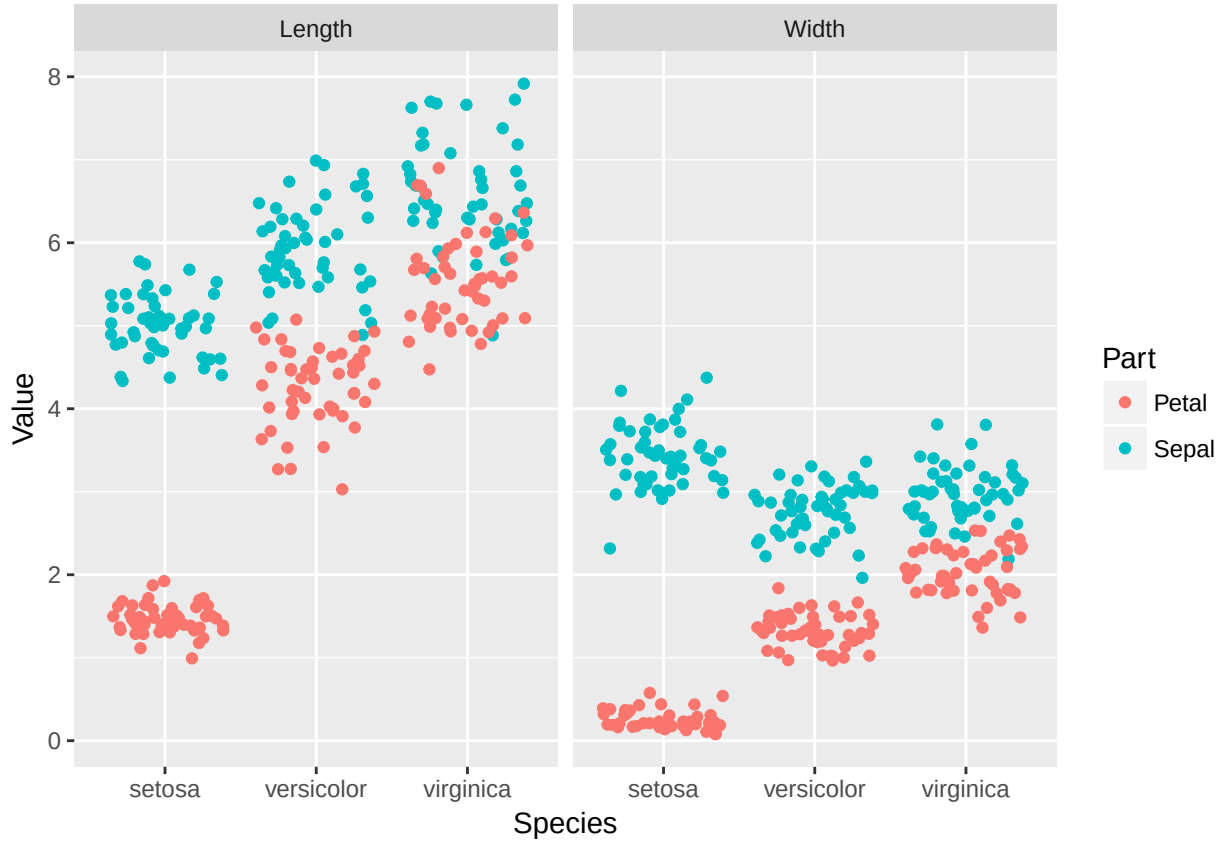
```
str(iris.wide)
```

```
## 'data.frame':    300 obs. of  4 variables:
##  $ Species: Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Part   : chr  "Petal" "Petal" "Petal" "Petal" ...
##  $ Length : num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Width  : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

```
str(iris.tidy)
```

```
## 'data.frame':    600 obs. of  4 variables:
##  $ Species: Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Part   : chr  "Sepal" "Sepal" "Sepal" "Sepal" ...
##  $ Measure: chr  "Length" "Length" "Length" "Length" ...
##  $ Value  : num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

```
# Think about which dataset you would use to get the plot shown right
# Fill in the ___ to produce the plot given to the right
ggplot(iris.tidy, aes(x = Species, y = Value, col = Part)) +
  geom_jitter() +
  facet_grid(. ~ Measure)
```



Variables to visuals, part 1b

In the last exercise you saw how `iris.tidy` was used to make a specific plot. It's important to know how to rearrange your data in this way so that your plotting functions become easier. In this exercise you'll use functions from the `tidyr` package to convert `iris` to `iris.tidy`.

The resulting `iris.tidy` data should look as follows:

```
  Species  Part Measure Value
1  setosa Sepal  Length   5.1
2  setosa Sepal  Length   4.9
3  setosa Sepal  Length   4.7
4  setosa Sepal  Length   4.6
5  setosa Sepal  Length   5.0
6  setosa Sepal  Length   5.4
...
```

You can have a look at the `iris` dataset by typing `head(iris)` in the console.

Note: If you're not familiar with`%>%`, `gather()` and `separate()`, you may want to take the *Cleaning Data in R* course. In a nutshell, a dataset is called tidy when every row is an observation and every column is a variable. The gather() function moves information from the columns to the rows. It takes multiple columns and gathers them into a single column by adding rows. The `separate()` function splits one column into two or more columns according to a pattern you define. Lastly, the `%>%` (or "pipe") operator passes the result of the left-hand side as the first argument of the function on the right-hand side.

INSTRUCTIONS

You'll use two functions from the `tidyr` package:

`gather()` rearranges the data frame by specifying the columns that are categorical variables with a `-` notation. Complete the command. Notice that only one variable is categorical in `iris`.

`separate()` splits up the new key column, which contains the former headers, according to `..` The new column names `"Part"` and `"Measure"` are given in a character vector. Don't forget the quotes.

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

# Fill in the ___ to produce to the correct iris.tidy dataset
iris.tidy <- iris %>%
  gather(key, Value, -Species) %>%
  separate(key, c("Part", "Measure"), "\\.")
```

Variables to visuals, part 2

Here you'll take a look at another plot variant, shown at right. Which of your data frames would be used to produce this plot?

INSTRUCTIONS

Look at the heads of `iris`, `iris.wide` and `iris.tidy` using `head()`.

Fill in the `ggplot` function with the appropriate data frame and variable names. The names of the aesthetics of the plot will match with variable names in your dataset. The previous instruction will help you match variable names in datasets with the ones in the plot.

```
# The 3 data frames (iris, iris.wide and iris.tidy) are available in your environment
# Execute head() on iris, iris.wide and iris.tidy (in that order)
head(iris)
```

```
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1  setosa          5.1         3.5          1.4         0.2
## 2  setosa          4.9         3.0          1.4         0.2
## 3  setosa          4.7         3.2          1.3         0.2
## 4  setosa          4.6         3.1          1.5         0.2
## 5  setosa          5.0         3.6          1.4         0.2
## 6  setosa          5.4         3.9          1.7         0.4
```

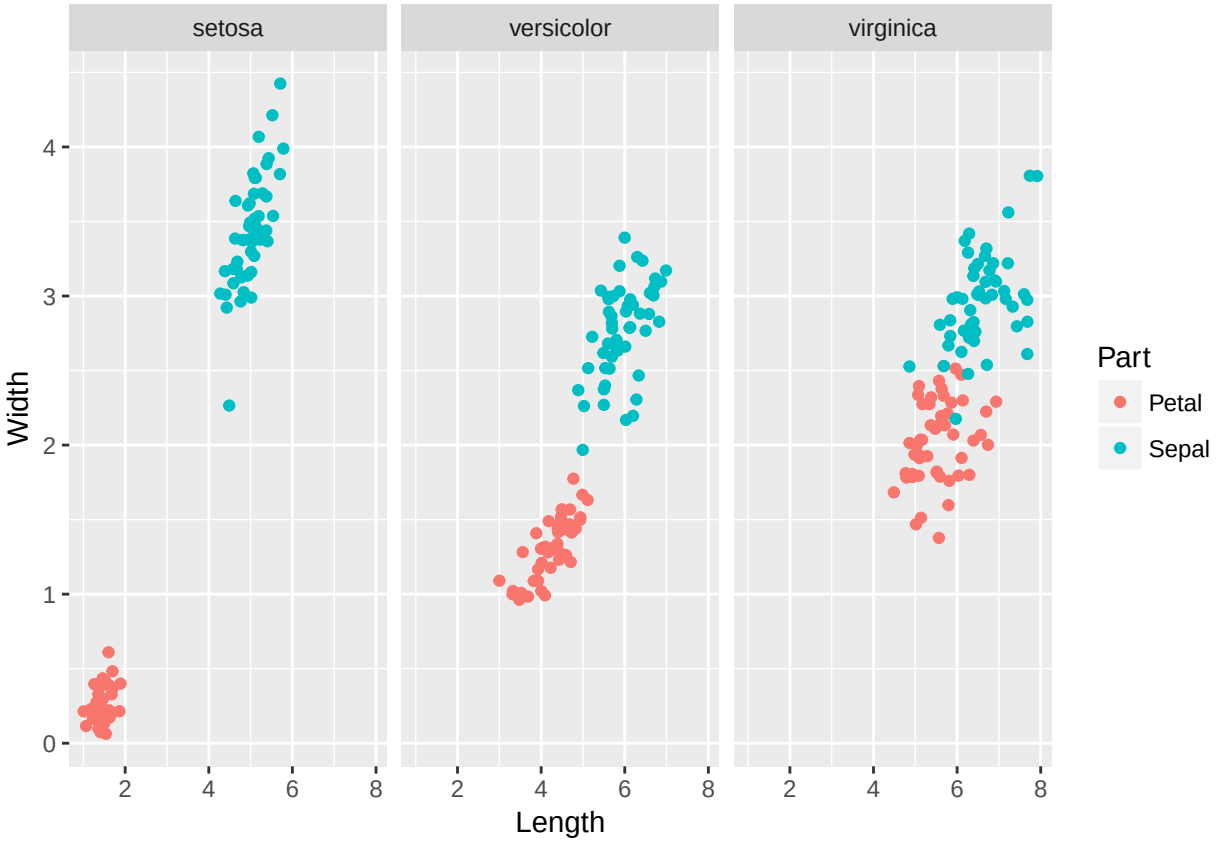
```
head(iris.wide)
```

```
##   Species  Part Length Width
## 1  setosa Petal    1.4   0.2
## 2  setosa Petal    1.4   0.2
## 3  setosa Petal    1.3   0.2
## 4  setosa Petal    1.5   0.2
## 5  setosa Petal    1.4   0.2
## 6  setosa Petal    1.7   0.4
```

```
head(iris.tidy)
```

```
##   Species  Part Measure Value
## 1  setosa Sepal  Length   5.1
## 2  setosa Sepal  Length   4.9
## 3  setosa Sepal  Length   4.7
## 4  setosa Sepal  Length   4.6
## 5  setosa Sepal  Length   5.0
## 6  setosa Sepal  Length   5.4
```

```
# Think about which dataset you would use to get the plot shown right
# Fill in the ___ to produce the plot given to the right
ggplot(iris.wide, aes(x = Length, y = Width, color = Part)) +
  geom_jitter() +
  facet_grid(. ~ Species)
```



Variables to visuals, part 2b

In the last exercise you saw how `iris.wide` was used to make a specific plot. You also saw previously how you can derive `iris.tidy` from `iri`. Now you'll move on to produce `iris.wide`.

The head of the `iris.wide` should look like this in the end:

```
  Species  Part Length Width
1  setosa Petal    1.4   0.2
2  setosa Petal    1.4   0.2
3  setosa Petal    1.3   0.2
4  setosa Petal    1.5   0.2
5  setosa Petal    1.4   0.2
6  setosa Petal    1.7   0.4
...
```

You can have a look at the `iris` dataset by typing `head(iris)` in the console.

INSTRUCTIONS

Before you begin, you need to add a new column called `Flower` that contains a unique identifier for each row in the data frame. This is because you'll rearrange the data frame afterwards and you need to keep track of which row, or which specific flower, each value came from. It's done for you, no need to add anything yourself.

`gather()` rearranges the data frame by specifying the columns that are categorical variables with a - notation. In this case, `Species` and `Flower` are categorical. Complete the command.

`separate()` splits up the new key column, which contains the former headers, according to `..` The new column names `"Part"` and `"Measure"` are given in a character vector.

The last step is to use `spread()` to distribute the new `Measure` column and associated value column into two columns.

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

# Add column with unique ids (don't need to change)
iris$Flower <- 1:nrow(iris)

# Fill in the ___ to produce to the correct iris.wide dataset
iris.wide <- iris %>%
  gather(key, value, -Species, -Flower) %>%
  separate(key, c("Part", "Measure"), "\\.") %>%
  spread(Measure, value)
```

Chapter 3: Aesthetics

Aesthetic mappings are the cornerstone of the grammar of graphics plotting concept. This is where the magic happens - converting continuous and categorical data into visual scales that provide access to a large amount of information in a very short time. In this chapter you'll understand how to choose the best aesthetic mappings for your data.

All about aesthetics, part 1

In the video you saw 9 visible aesthetics. Let's apply them to a categorical variable - the cylinders in `mtcars`, `cyl`.

(You'll consider line type when you encounter line plots in the next chapter).

These are the aesthetics you can consider within `aes()` in this chapter: `x`,`y`, `color`, `fill`, `size`, `alpha`, `labels` and `shape`.

In the following exercise you can assume that the `cyl` column is categorical. It has already been transformed into a factor for you.

INSTRUCTIONS

The `mtcars` data frame is available in your workspace. For each of the following four plots, use `geom_point()`:

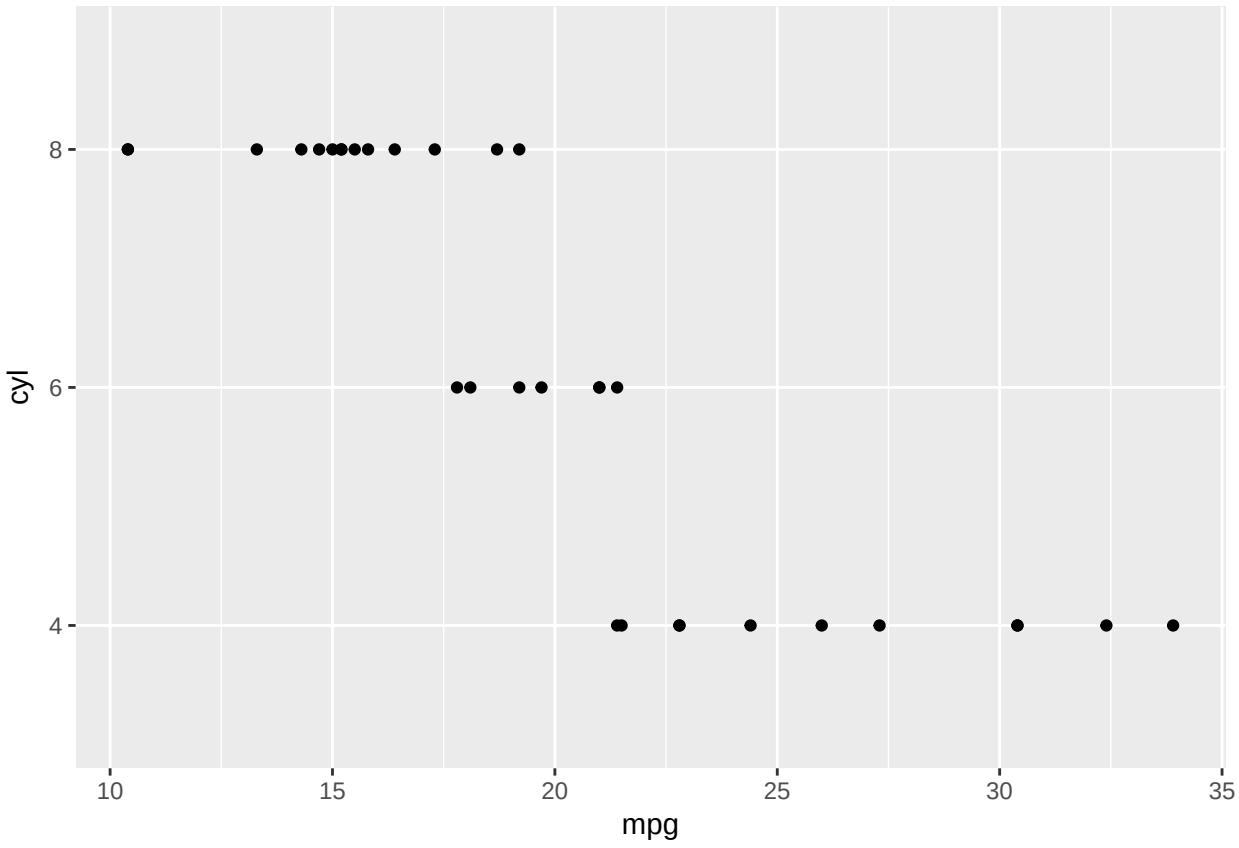
1 - Map `mpg` onto the `x` aesthetic, and `cyl` onto the `y`.

2 - Reverse the mappings of the first plot.

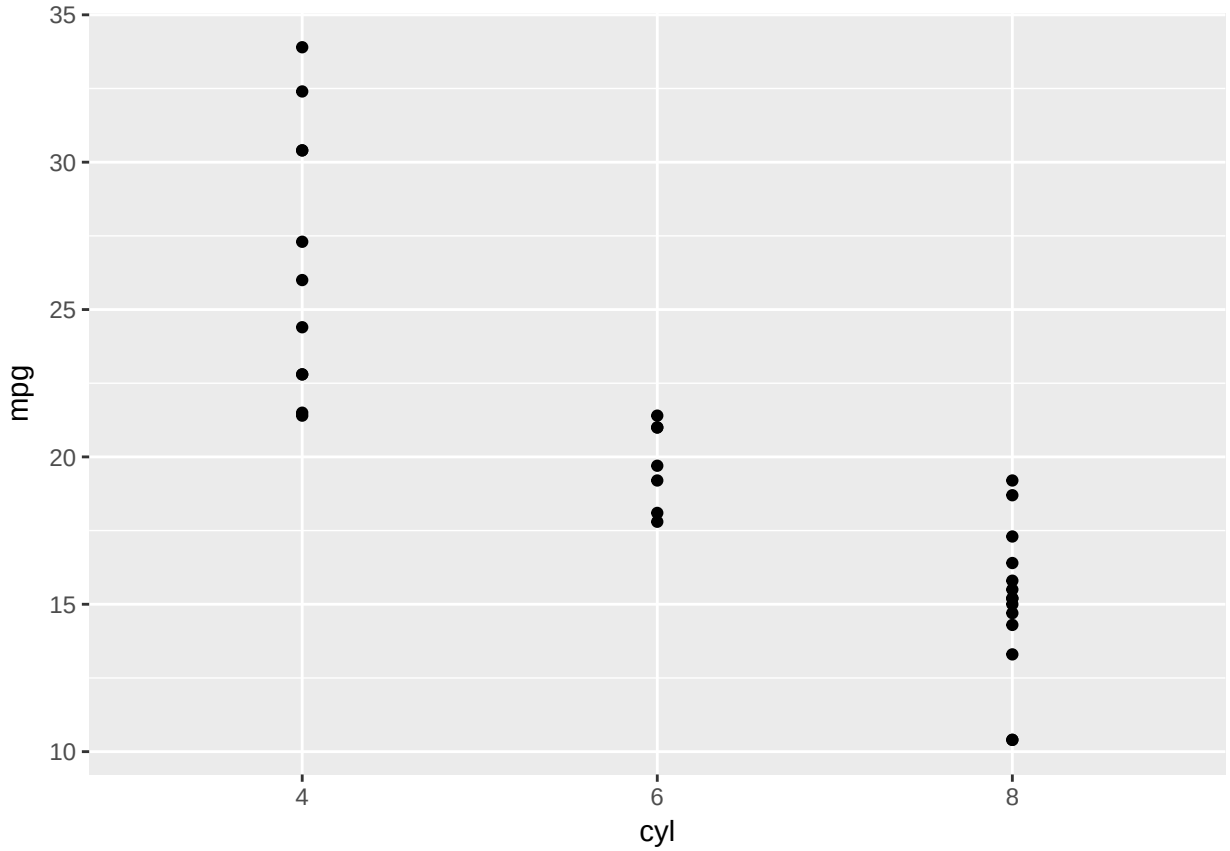
3 - Map `wt` onto `x`, `mpg` onto `y`, and `cyl` onto `color`.

Modify the previous plot by changing the `shape` argument of the geom to `1` and increase the `size` to `4`. These are attributes that you should specify inside `geom_point()`.

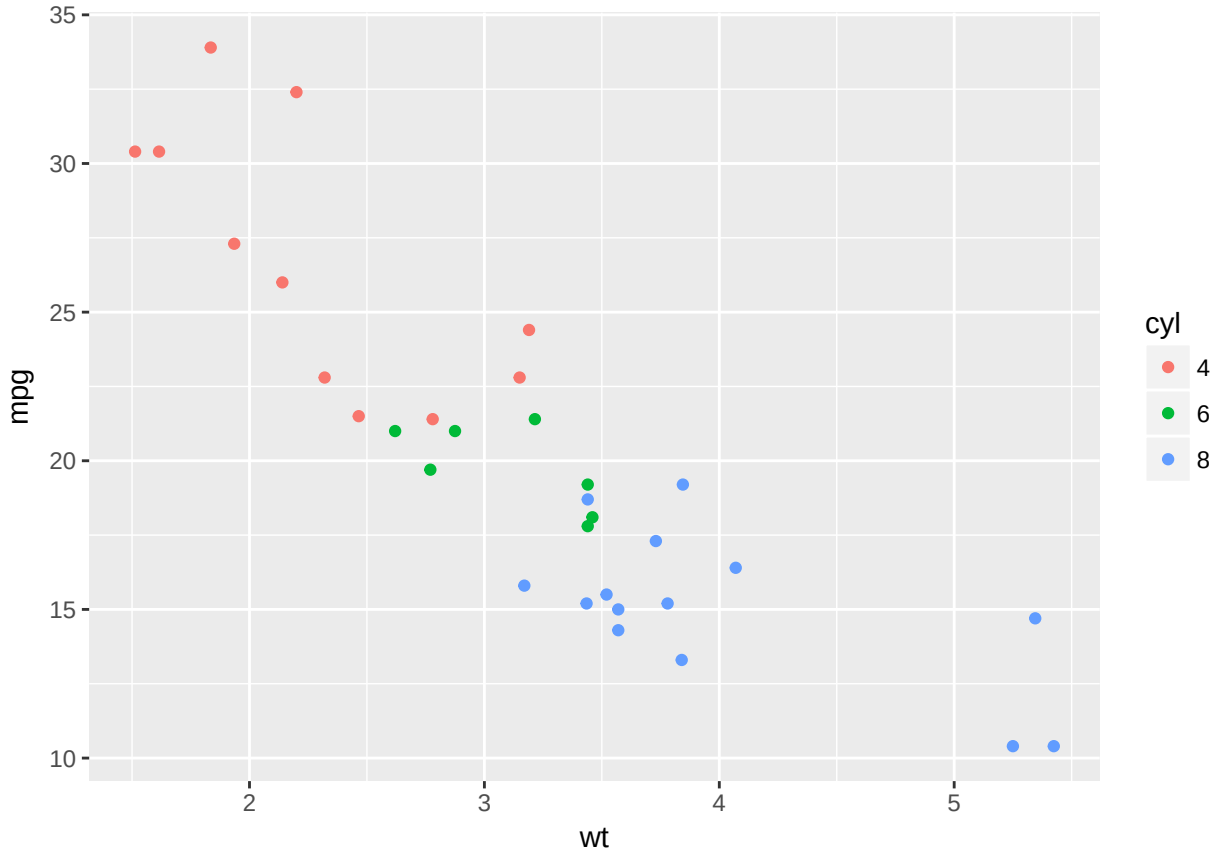
```
# 1 - Map mpg to x and cyl to y
ggplot(mtcars, aes(x = mpg, y = cyl)) +
  geom_point()
```



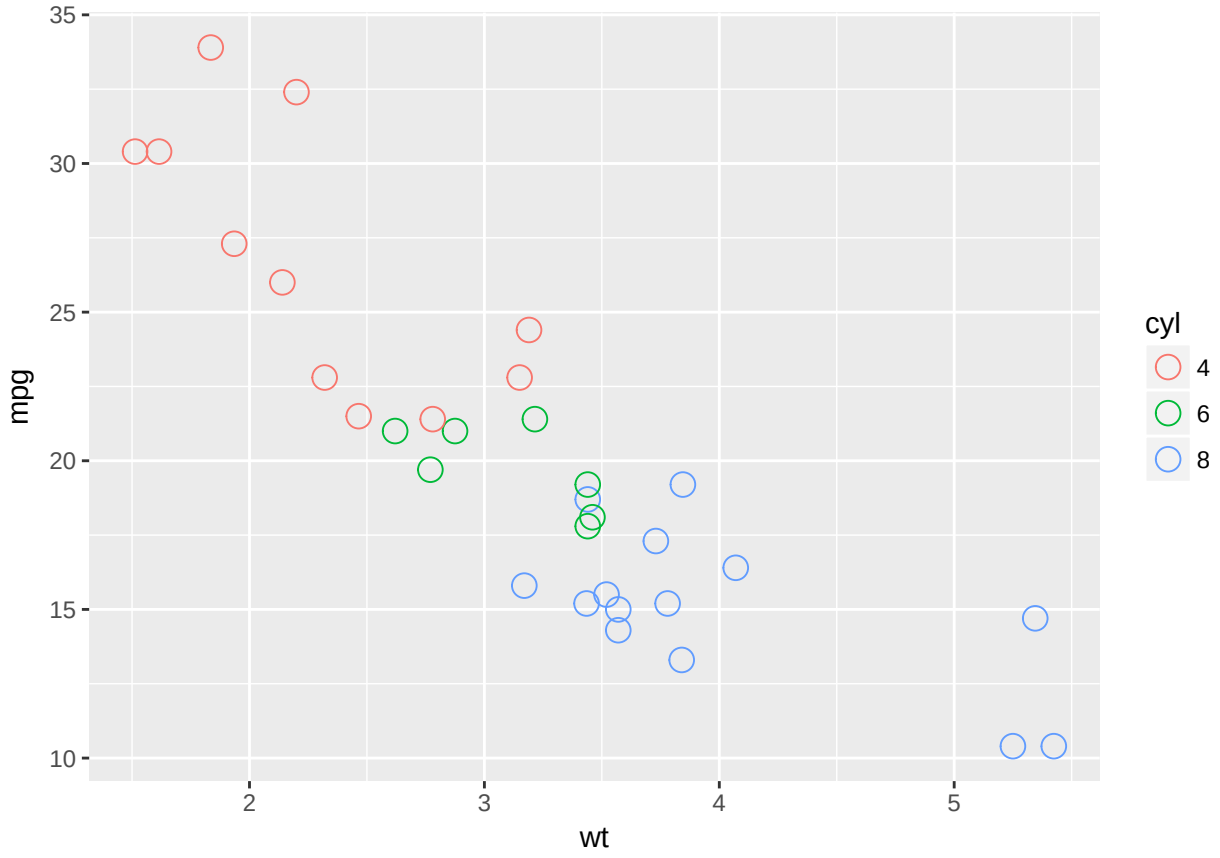

```
# 2 - Reverse: Map cyl to x and mpg to y
ggplot(mtcars, aes(x = cyl, y = mpg)) +
  geom_point()
```




```
# 3 - Map wt to x, mpg to y and cyl to col
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point()
```




```
# 4 - Change shape and size of the points in the above plot
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point(shape = 1, size = 4)
```



All about aesthetics, part 2

The `color` aesthetic typically changes the outside outline of an object and the `fill` aesthetic is typically the inside shading. However, as you saw in the last exercise, `geom_point()` is an exception. Here you use `color`, instead of `fill` for the inside of the point. But it's a bit subtler than that.

Which shape to use? The default `geom_point()` uses `shape = 19` (a solid circle with an outline the same colour as the inside). Good alternatives are `shape = 1` (hollow) and `shape = 16` (solid, no outline). These all use the col aesthetic (don't forget to set `alpha` for solid points).

A really nice alternative is `shape = 21` which allows you to use both fill for the inside and col for the outline! This is a great little trick for when you want to map two aesthetics to a dot.

What happens when you use the wrong aesthetic mapping? This is a very common mistake! The code from the previous exercise is in the editor. Using this as your starting point complete the instructions.

INSTRUCTIONS

Note: In the `mtcars` dataset, `cyl` and am have been converted to factor for you.

1 - Copy & paste the first plot's code. Change the aesthetics so that `cyl` maps to `fill` rather than `col`.

2 - Copy & paste the second plot's code. In `geom_point()` change the `shape` argument to `21` and add an `alpha` argument set to `0.6`.

3 - Copy & paste the third plot's code. In the `ggplot()` aesthetics, map `am` to `col`.

```
mtcars <- mtcars %>%
  mutate(am = factor(am))
```

```
# am and cyl are factors, wt is numeric
class(mtcars$am)
```

```
## [1] "factor"
```

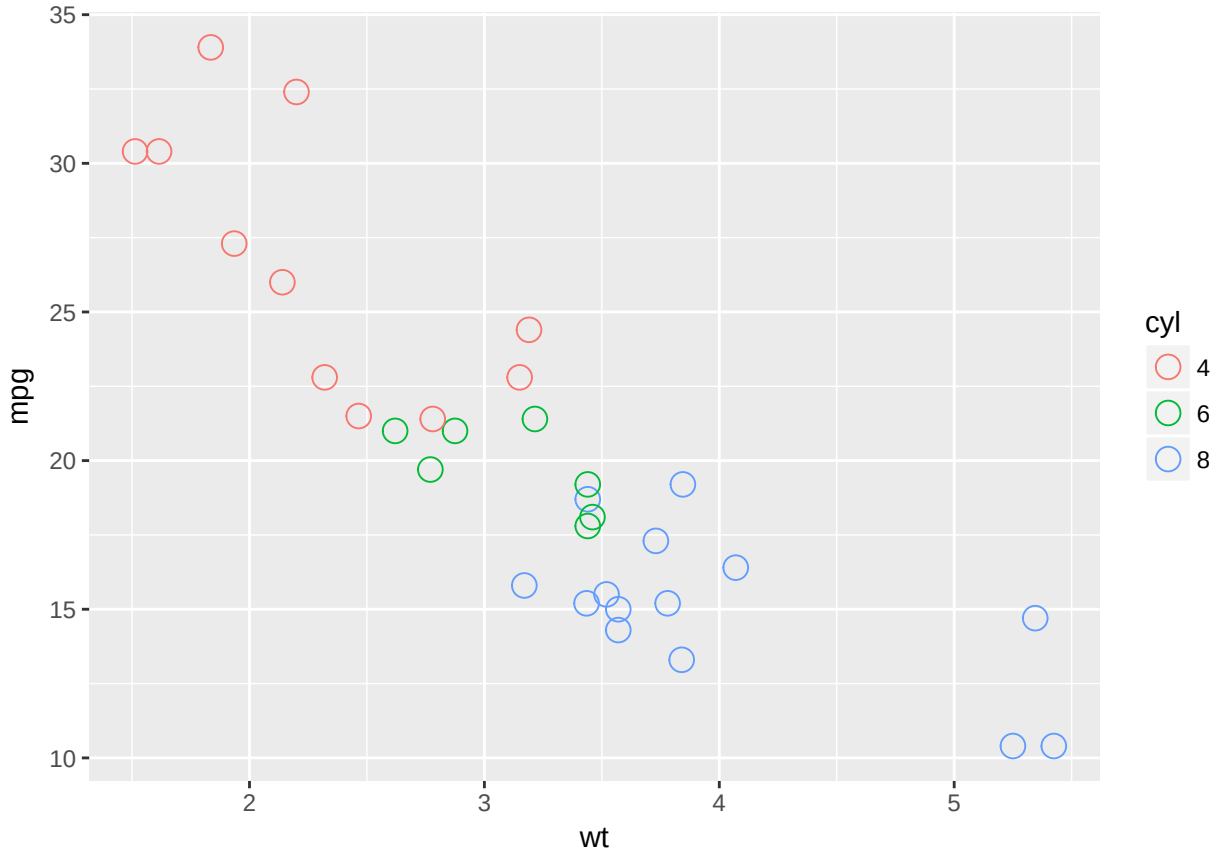
```
class(mtcars$cyl)
```

```
## [1] "factor"
```

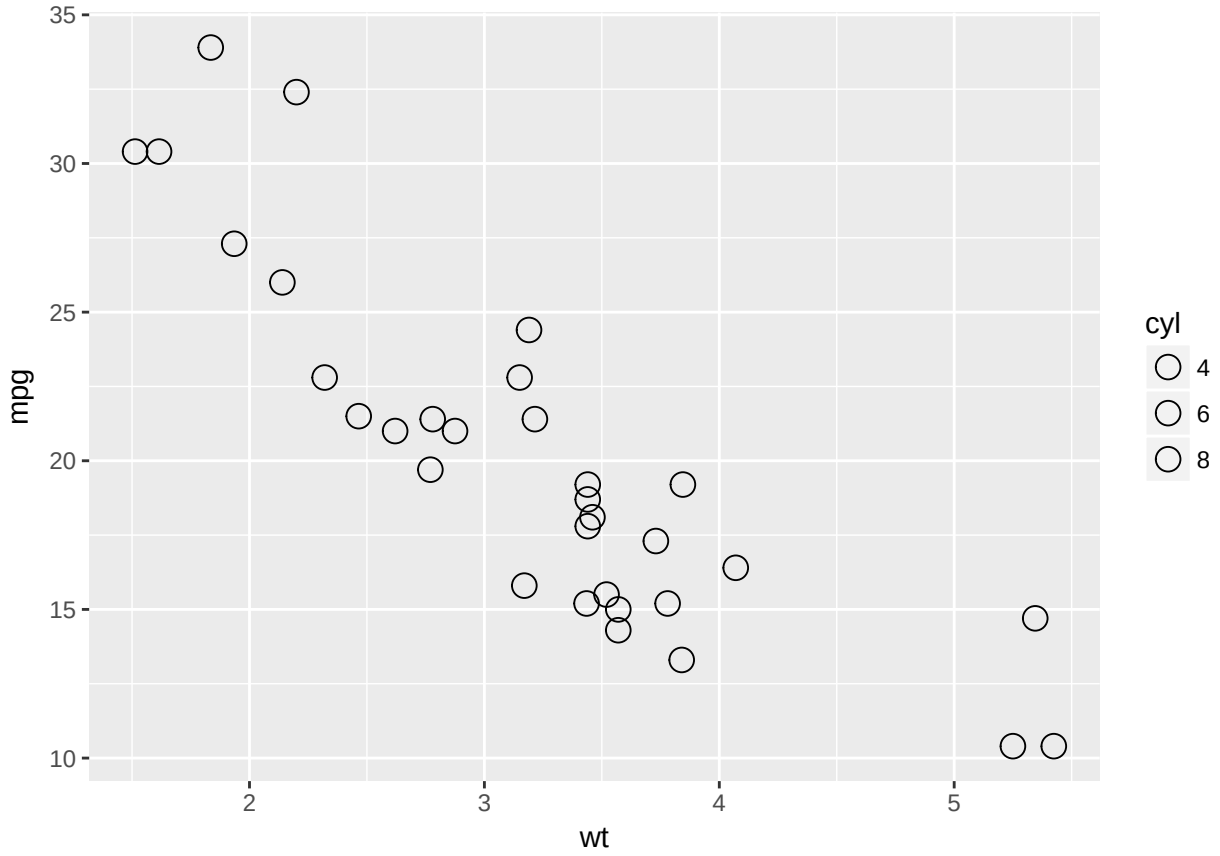
```
class(mtcars$wt)
```

```
## [1] "numeric"
```

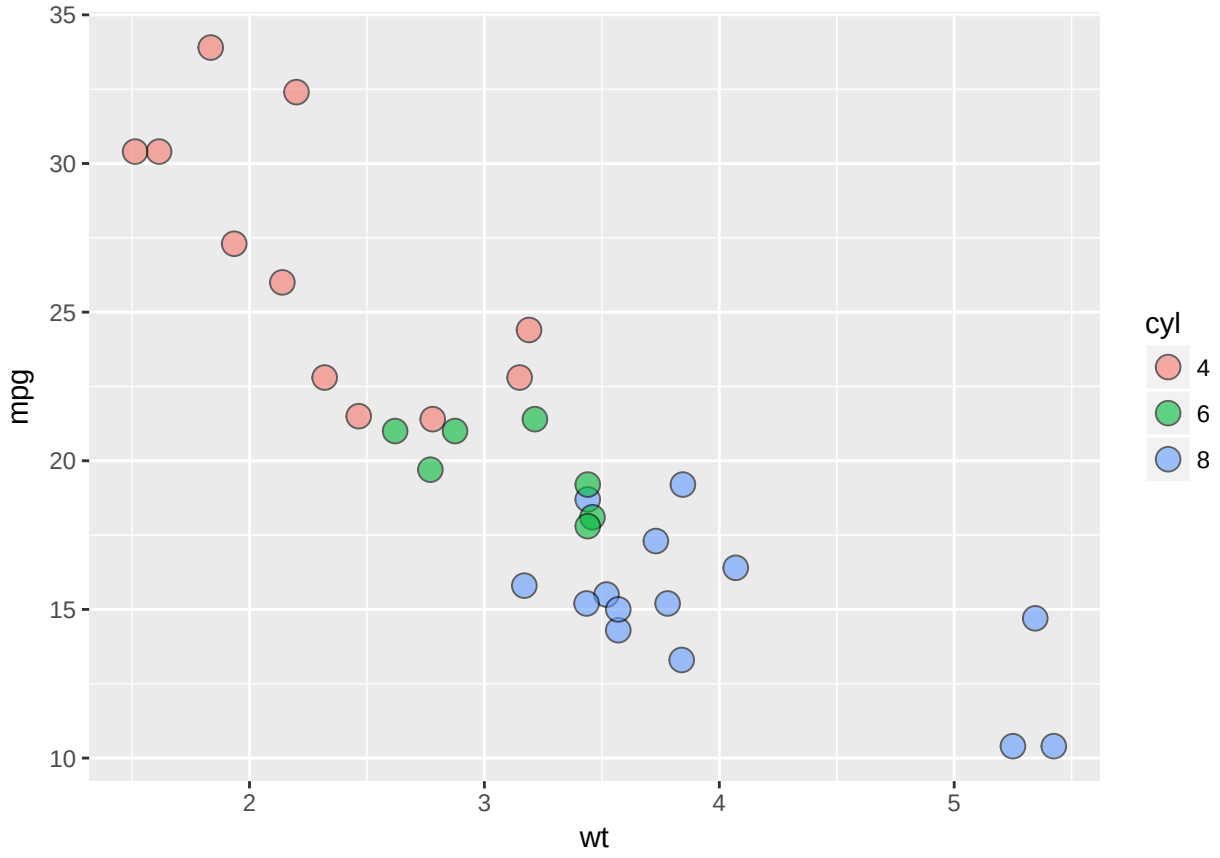
```
# From the previous exercise
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point(shape = 1, size = 4)
```



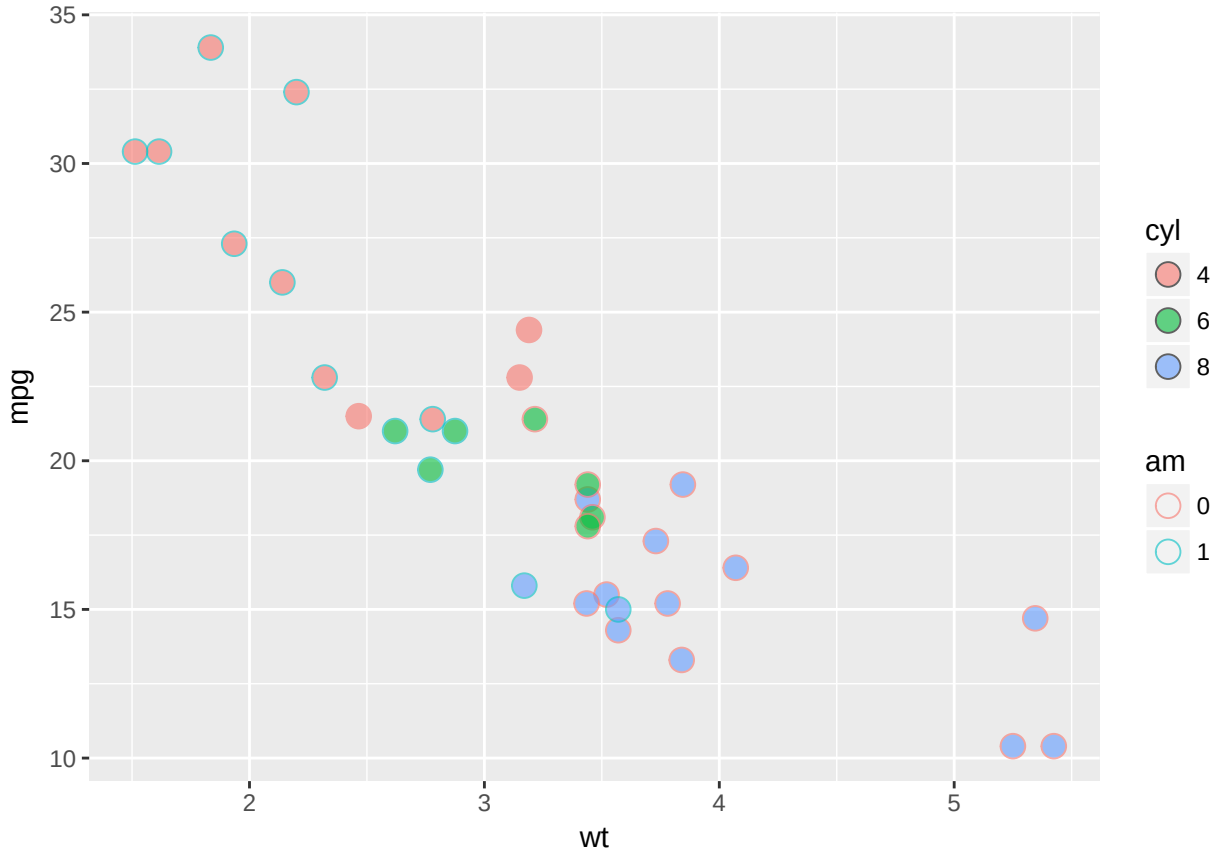

```
# 1 - Map cyl to fill
ggplot(mtcars, aes(x = wt, y = mpg, fill = cyl)) +
  geom_point(shape = 1, size = 4)
```




```
# 2 - Change shape and alpha of the points in the above plot
ggplot(mtcars, aes(x = wt, y = mpg, fill = cyl)) +
  geom_point(shape = 21, size = 4, alpha = 0.6)
```




```
# 3 - Map am to col in the above plot
ggplot(mtcars, aes(x = wt, y = mpg, fill = cyl, col = am)) +
  geom_point(shape = 21, size = 4, alpha = 0.6)
```



All about aesthetics, part 3

Now that you've got some practice with incrementally building up plots, you can try to do it from scratch! The `mtcars` dataset is pre-loaded in the workspace.

INSTRUCTIONS

Use `ggplot()` to create a basic scatter plot. Inside `aes()`, map `wt` onto `x` and `mpg` onto `y`. Typically, you would say "`mpg` described by `wt`" or "`mpg` vs `wt`", but in `aes()`, it's `x` first, `y` second. Use `geom_point()` to make three scatter plots:

`cyl` on `size`

`cyl` on `alpha`

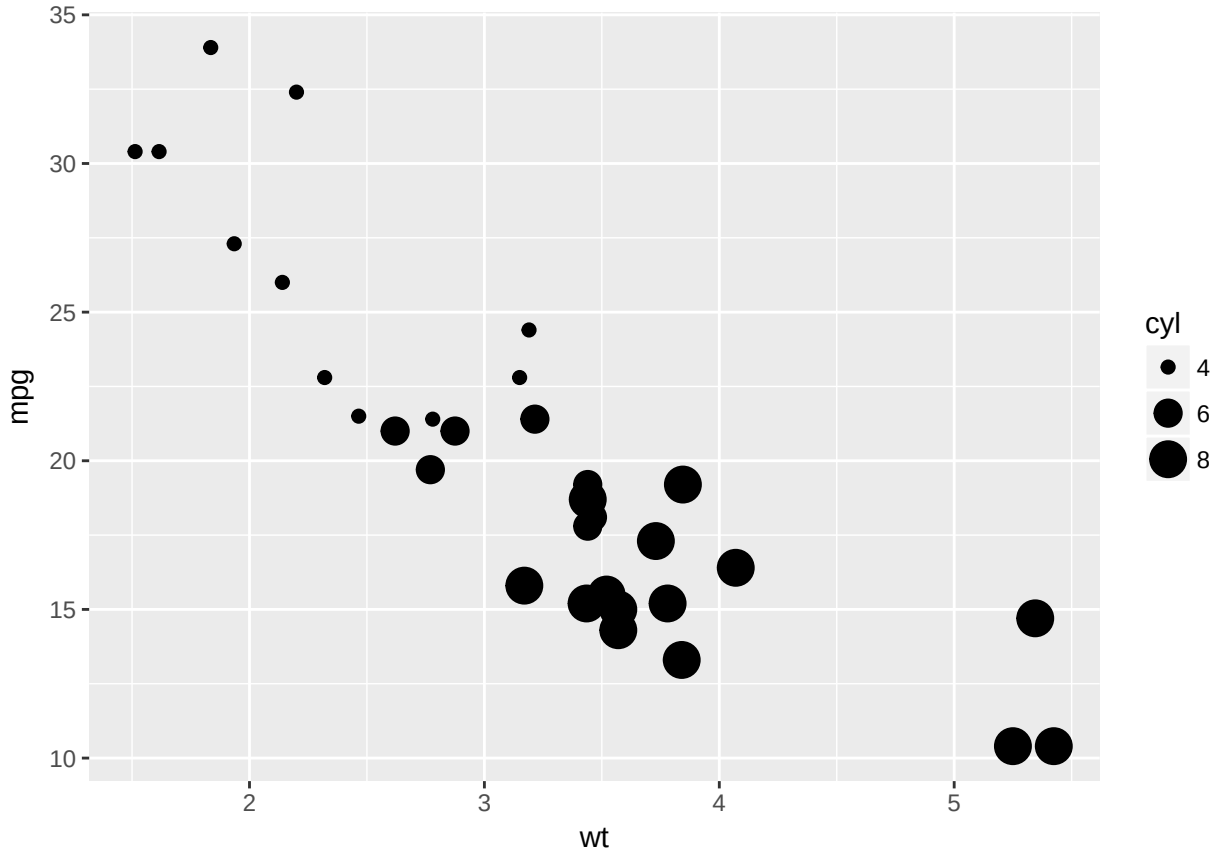
`cyl` on `shape`

Try this last variant:

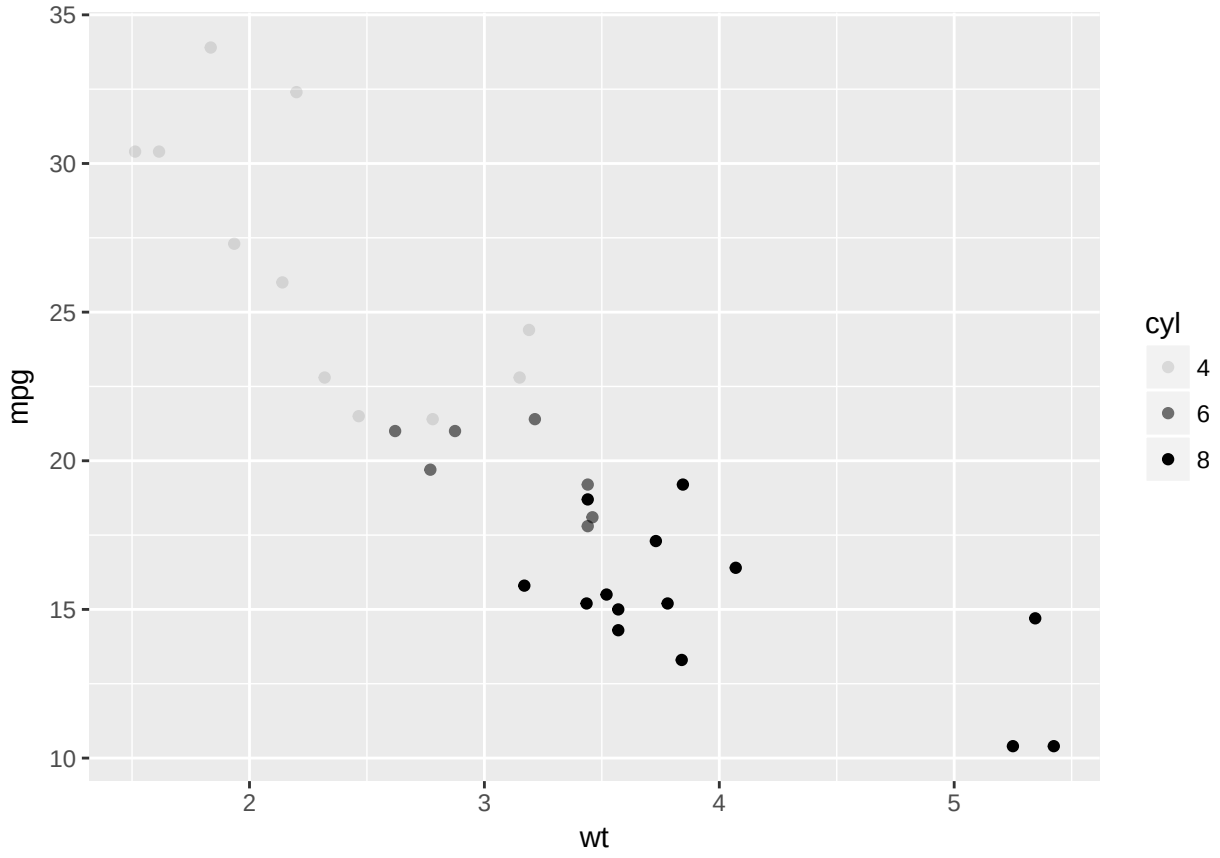
`cyl` on `label`. In order to correctly show the test (i.e. label), use `geom_text()`.

```
# Map cyl to size
ggplot(mtcars, aes(x = wt, y = mpg, size = cyl)) +
  geom_point()
```

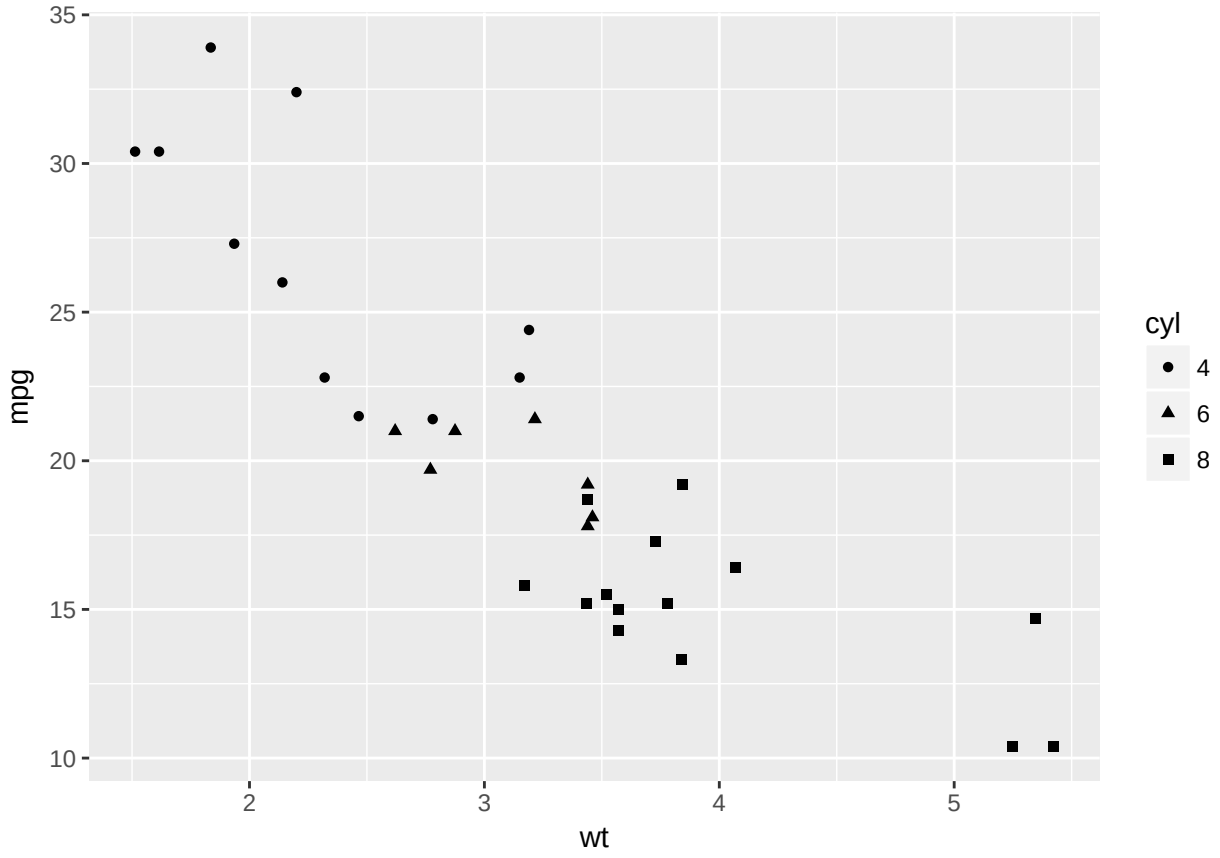
```
## Warning: Using size for a discrete variable is not advised.
```



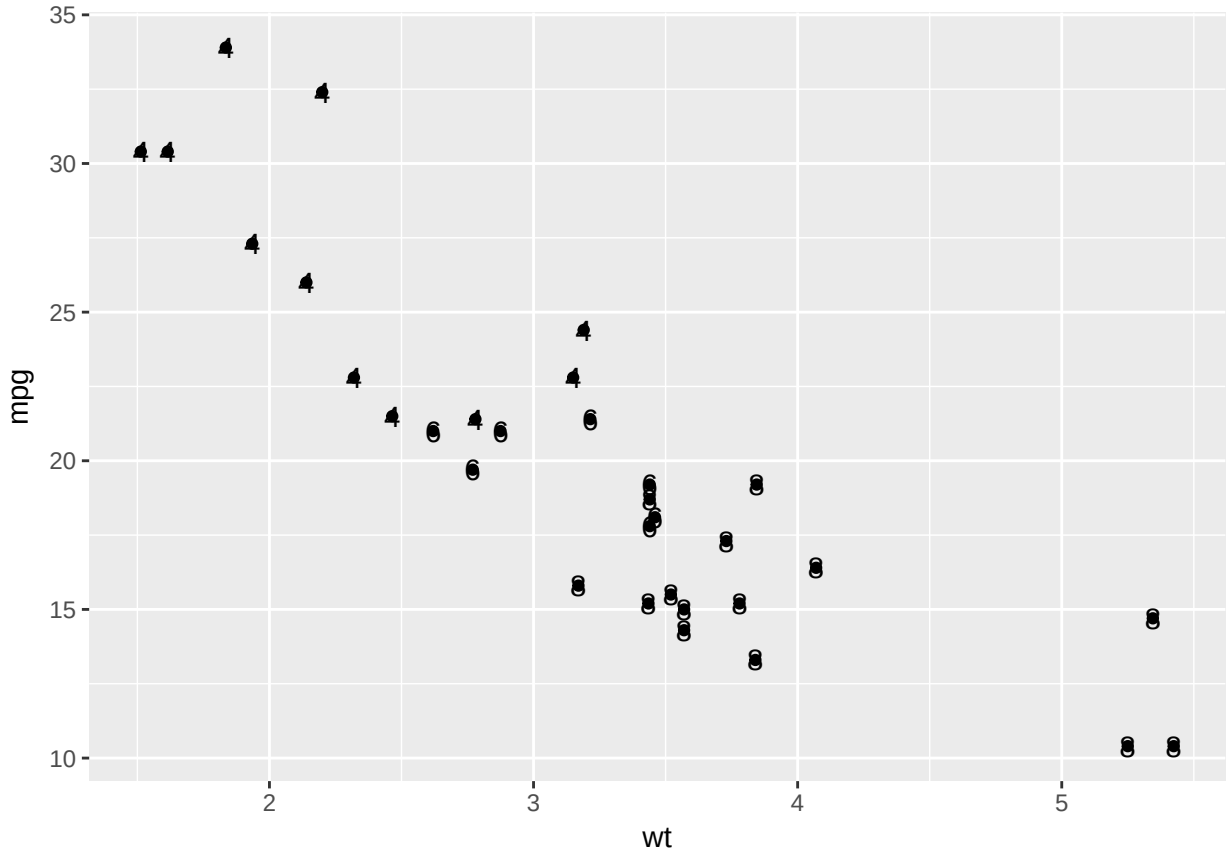

```
# Map cyl to alpha
ggplot(mtcars, aes(x = wt, y = mpg, alpha = cyl)) +
  geom_point()
```




```
# Map cyl to shape
ggplot(mtcars, aes(x = wt, y = mpg, shape = cyl)) +
  geom_point()
```




```
# Map cyl to label
ggplot(mtcars, aes(x = wt, y = mpg, label = cyl)) +
  geom_point() +
  geom_text()
```



All about attributes, part 1

In the video you saw that you can use all the aesthetics as attributes. Let's see how this works with the aesthetics you used in the previous exercises: `x`, `y`, `color`, `fill`, `size`, `alpha`, `label` and `shape`.

This time you'll use these arguments to set attributes of the plot, not aesthetics. However, there are some pitfalls you'll have to watch out for: these attributes can overwrite the aesthetics of your plot!

A word about shapes: In the exercise "All about aesthetics, part 2", you saw that `shape = 21` results in a point that has a fill and an outline. Shapes in R can have a value from 1-25. Shapes `1-20` can only accept a color aesthetic, but shapes `21-25` have both a color and a fill aesthetic. See the `pch` argument in `par()` for further discussion.

A word about hexadecimal colours: Hexadecimal, literally "related to 16", is a base-16 alphanumeric counting system. Individual values come from the ranges 0-9 and A-F. This means there are 256 possible two-digit values (i.e. 00 - FF). Hexadecimal colours use this system to specify a six-digit code for Red, Green and Blue values (`"#RRGGBB"`) of a colour (i.e. Pure blue: `"#0000FF"`, black: `"#000000"`, white: `"#FFFFFF"`). R can accept hex codes as valid colours.

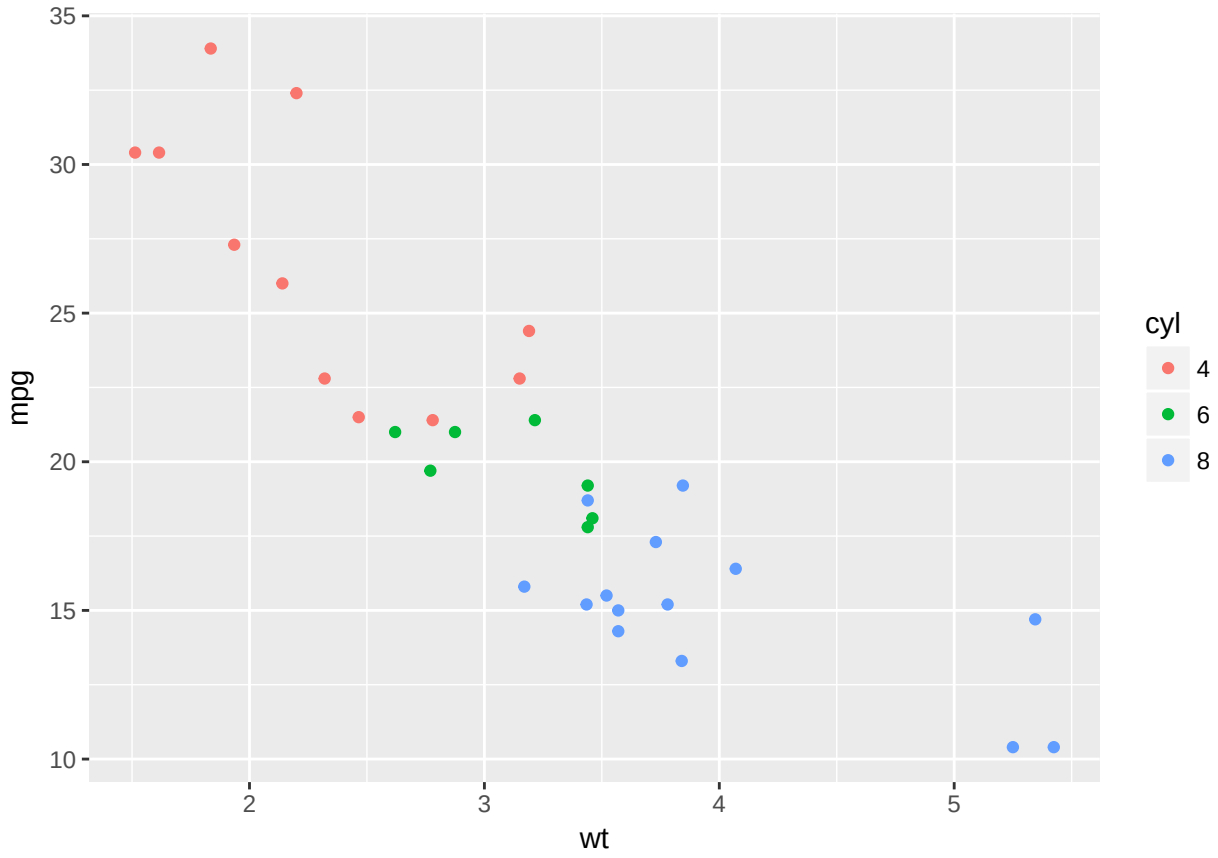
INSTRUCTIONS

1 - You will continue to work with `mtcars`. Use `ggplot()` to create a basic scatter plot: map `wt` onto `x`, `mpg` onto `y` and `cyl` onto `color`.

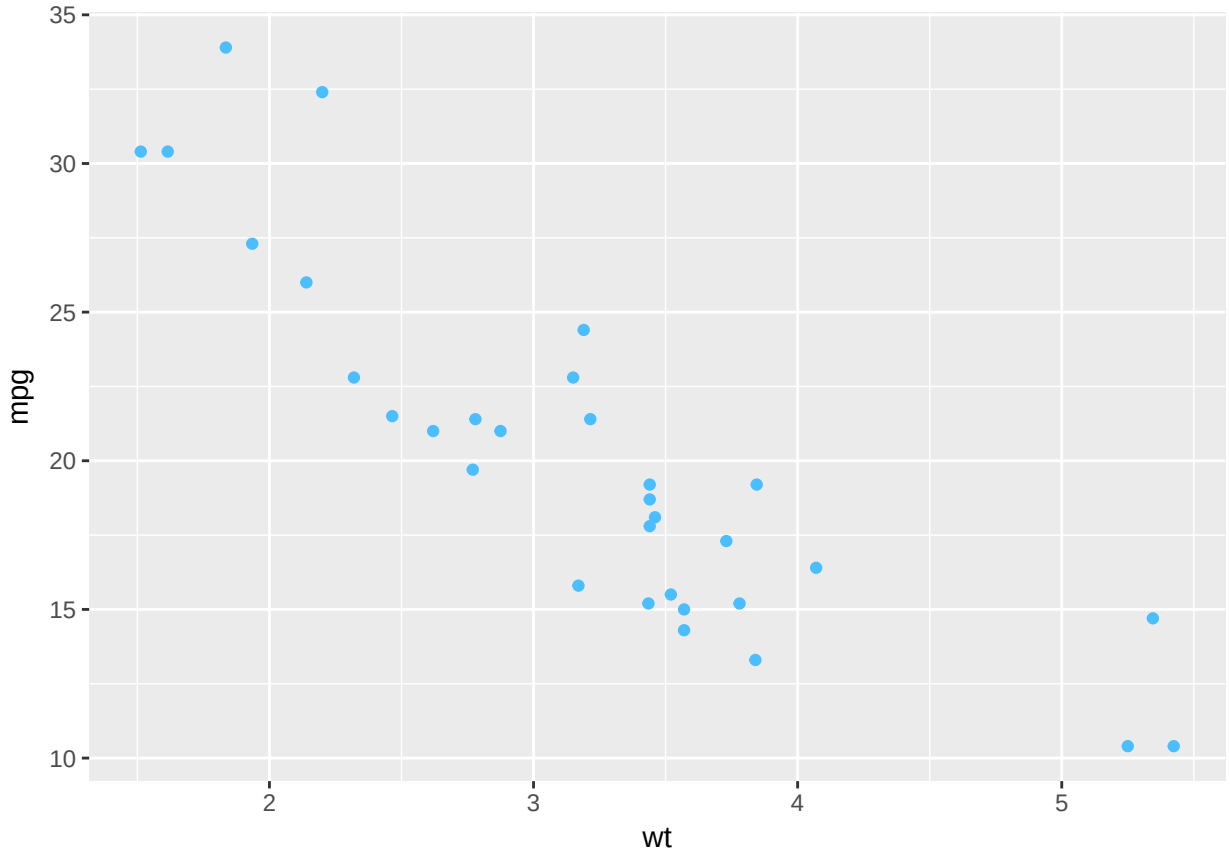
2 - Overwrite the color of the points inside `geom_point()` to `my_color`. Notice how this cancels out the colors given to the points by the number of cylinders!

3 - Starting with plot 2, map `cyl` to fill instead of `col` and set the attributes `size` to `10`, `shape` to `23` and `color` to `my_color` inside `geom_point()`.

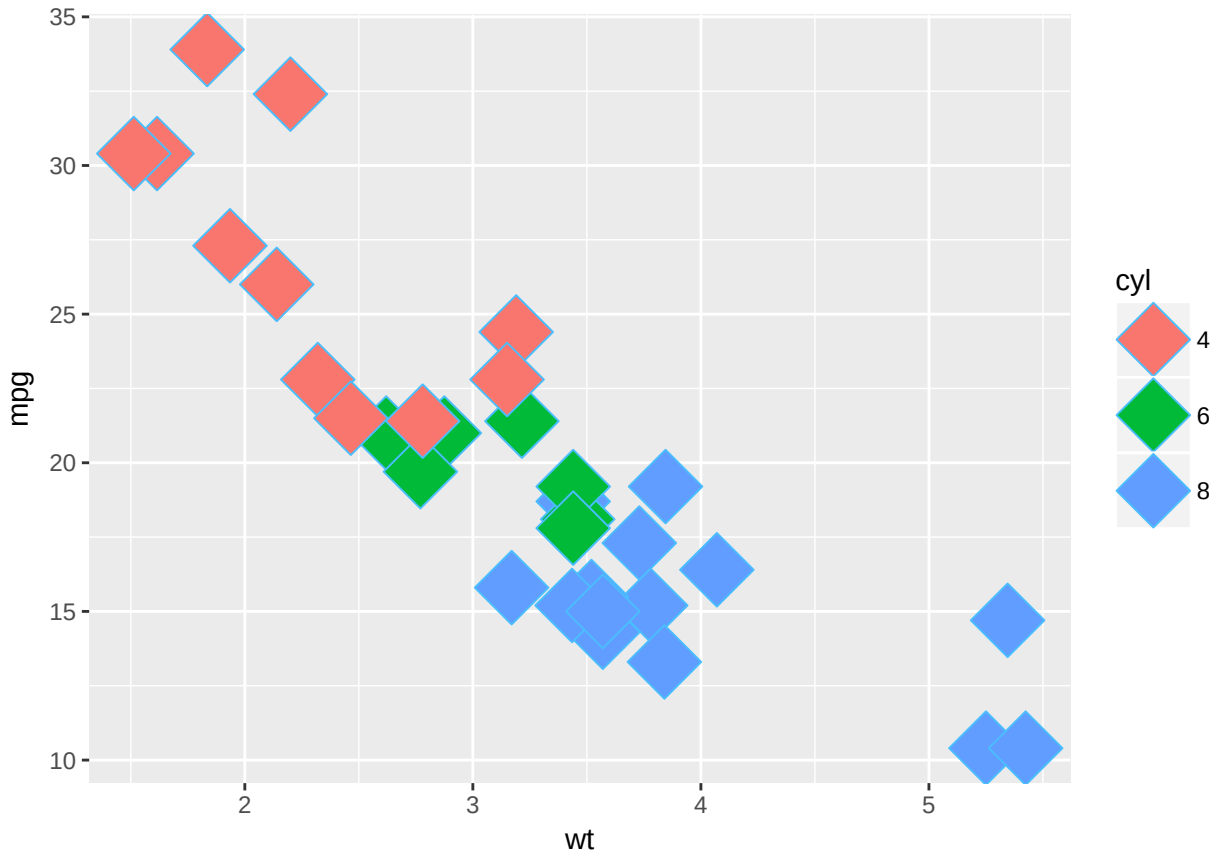
```
# Define a hexadecimal color
my_color <- "#4ABEFF"

# 1 - First scatter plot, with col aesthetic:
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point()
```




```
# 2 - Plot 1, but set col attributes in geom layer:
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point(color = my_color)
```




```
# 3 - Plot 2, with fill instead of col aesthetic, plut shape and size attributes in geom layer.
ggplot(mtcars, aes(x = wt, y = mpg, fill = cyl)) +
  geom_point(size = 10, shape = 23, color = my_color)
```



All about attributes, part 2

In the videos you saw that you can use all the aesthetics as attributes. Let's see how this works with the aesthetics you used in the previous exercises: `x`, `y`, `color`, `fill`, `size`, `alpha`, `label` and `shape`.

In this exercise you will set all kinds of attributes of the points!

You will continue to work with `mtcars`.

INSTRUCTIONS

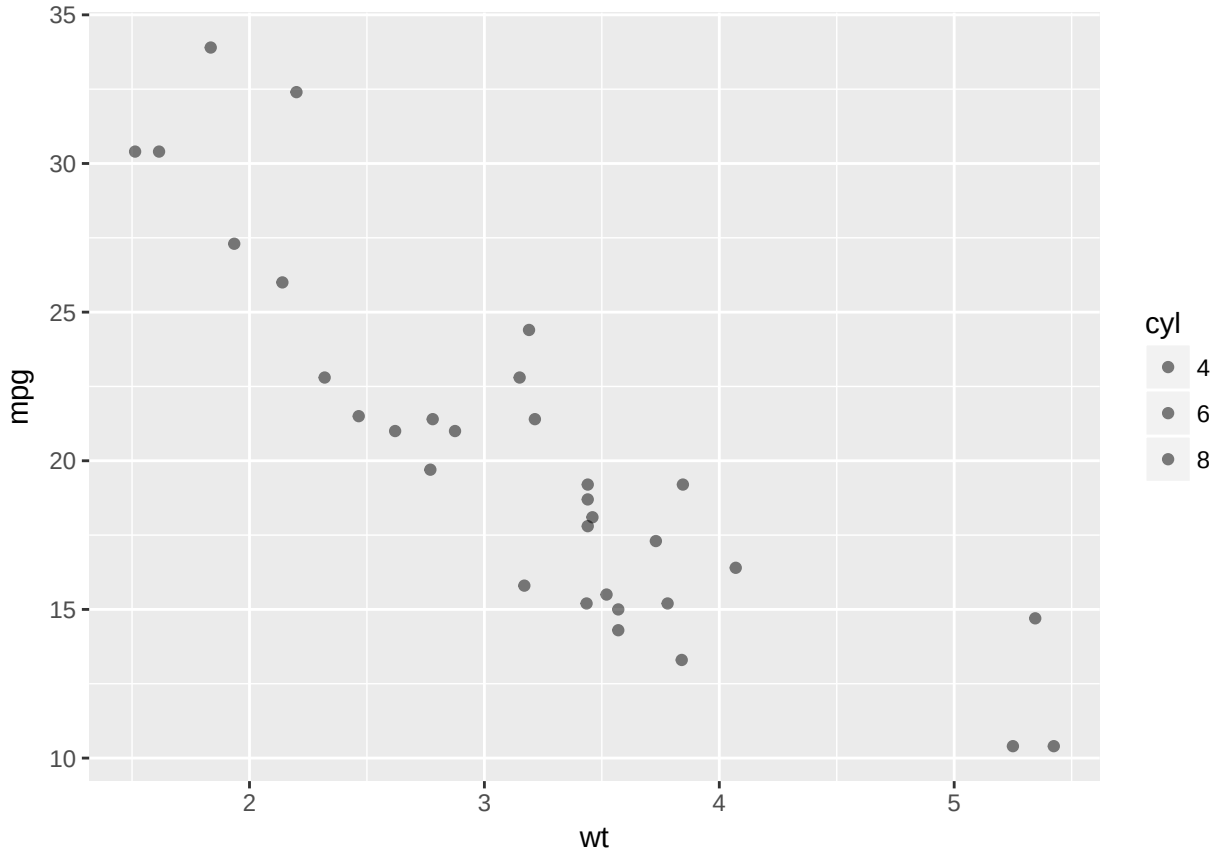
Add to the first command: draw points with `alpha` set to `0.5`.

Add to the second command: draw points of shape `24` in the color `yellow`.

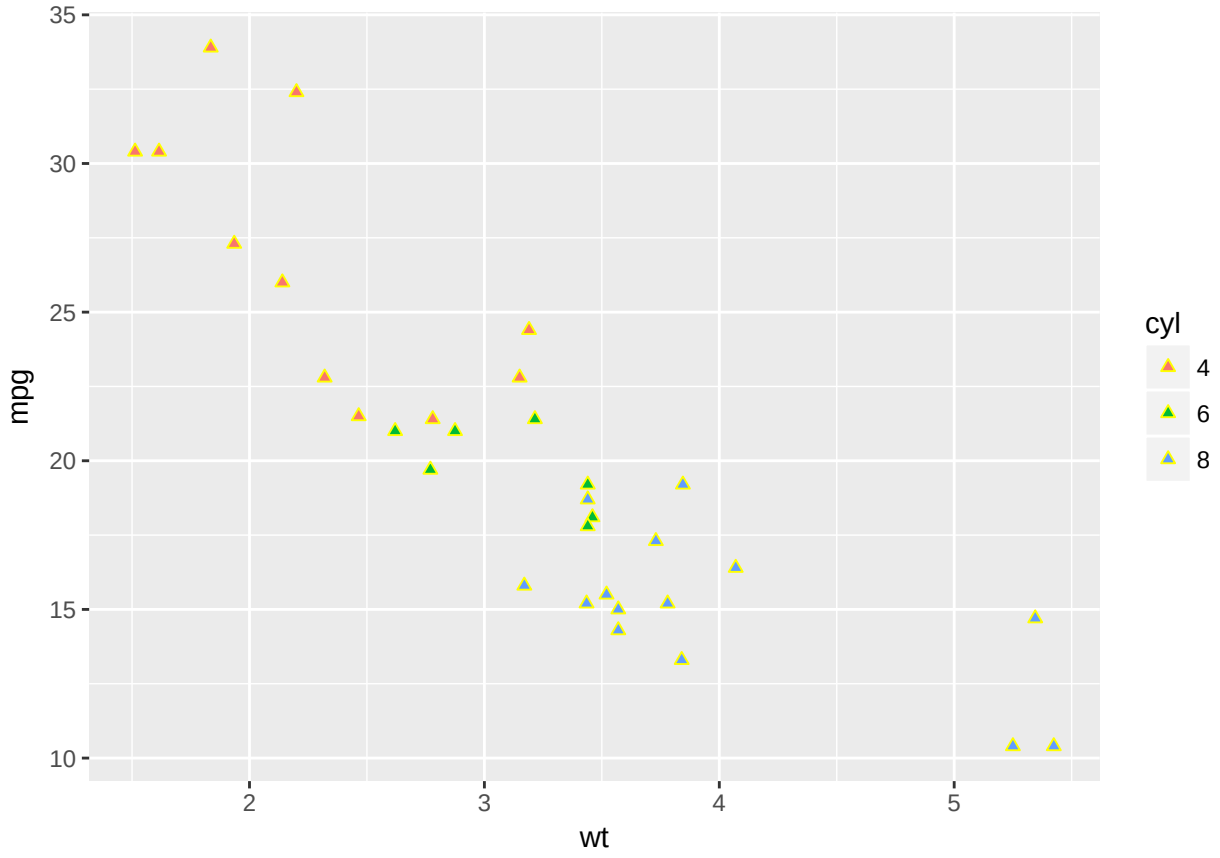
Add to the third command: draw text with label `rownames(mtcars)` in the color `red`. Don't use `geom_point()` here! You should get a scatter plot with the names of the cars instead of points.

Note: Remember to specify characters with quotation marks (`"yellow"`, not `yellow`).

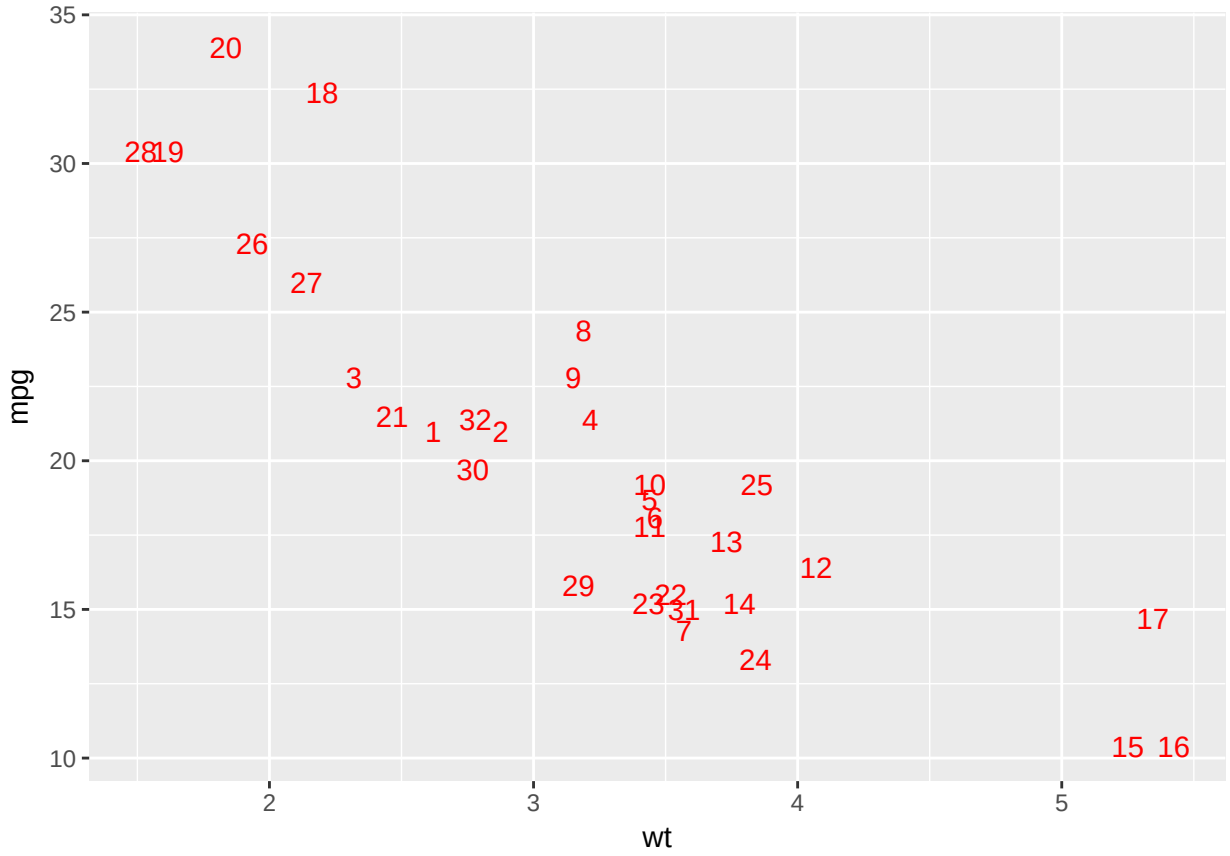
```
# Expand to draw points with alpha 0.5
ggplot(mtcars, aes(x = wt, y = mpg, fill = cyl)) +
  geom_point(alpha = 0.5)
```




```
# Expand to draw points with shape 24 and color yellow
ggplot(mtcars, aes(x = wt, y = mpg, fill = cyl)) +
  geom_point(shape = 24, color = "yellow")
```




```
# Expand to draw text with label rownames(mtcars) and color red
ggplot(mtcars, aes(x = wt, y = mpg, fill = cyl, label = rownames(mtcars))) +
  geom_text(color = "red")
```



Going all out

In this exercise, you will gradually add more aesthetics layers to the plot. You're still working with the `mtcars` dataset, but this time you're using more features of the cars. For completeness, here is a list of all the features of the observations in `mtcars`:

- `mpg` – Miles/(US) gallon
- `cyl` – Number of cylinders
- `disp` – Displacement (cu.in.)
- `hp` – Gross horsepower
- `drat` – Rear axle ratio
- `wt` – Weight (lb/1000)
- `qsec` – 1/4 mile time
- `vs` – V/S engine.
- `am` – Transmission (0 = automatic, 1 = manual)
- `gear` – Number of forward gears
- `carb` – Number of carburetors

Notice that adding more aesthetics to your plot is not always a good idea. Adding aesthetic mappings to a plot will increase its complexity, and thus decrease its readability.

INSTRUCTIONS

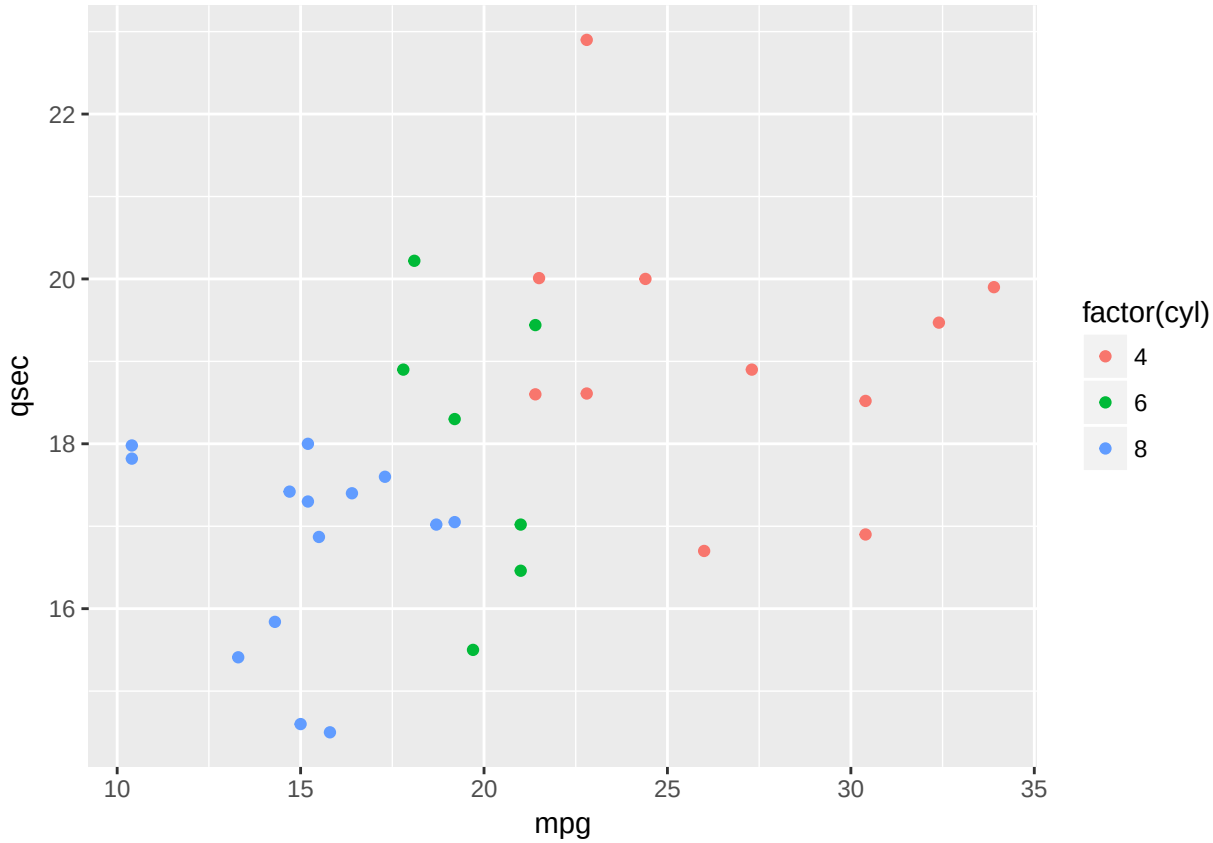
Note: In this chapter you saw aesthetics and attributes. Variables in a data frame are mapped to aesthetics in `aes()`. (e.g. `aes(col = cyl)`) within `ggplot()`. Visual elements are set by attributes in specific geom layers (`geom_point(col = "red")`). Don't confuse these two things - here you're focusing on aesthetic mappings.

Draw a scatter plot of `mtcars` with `mpg` on the x-axis, `qsec` on the y-axis and `factor(cyl)` as colors.

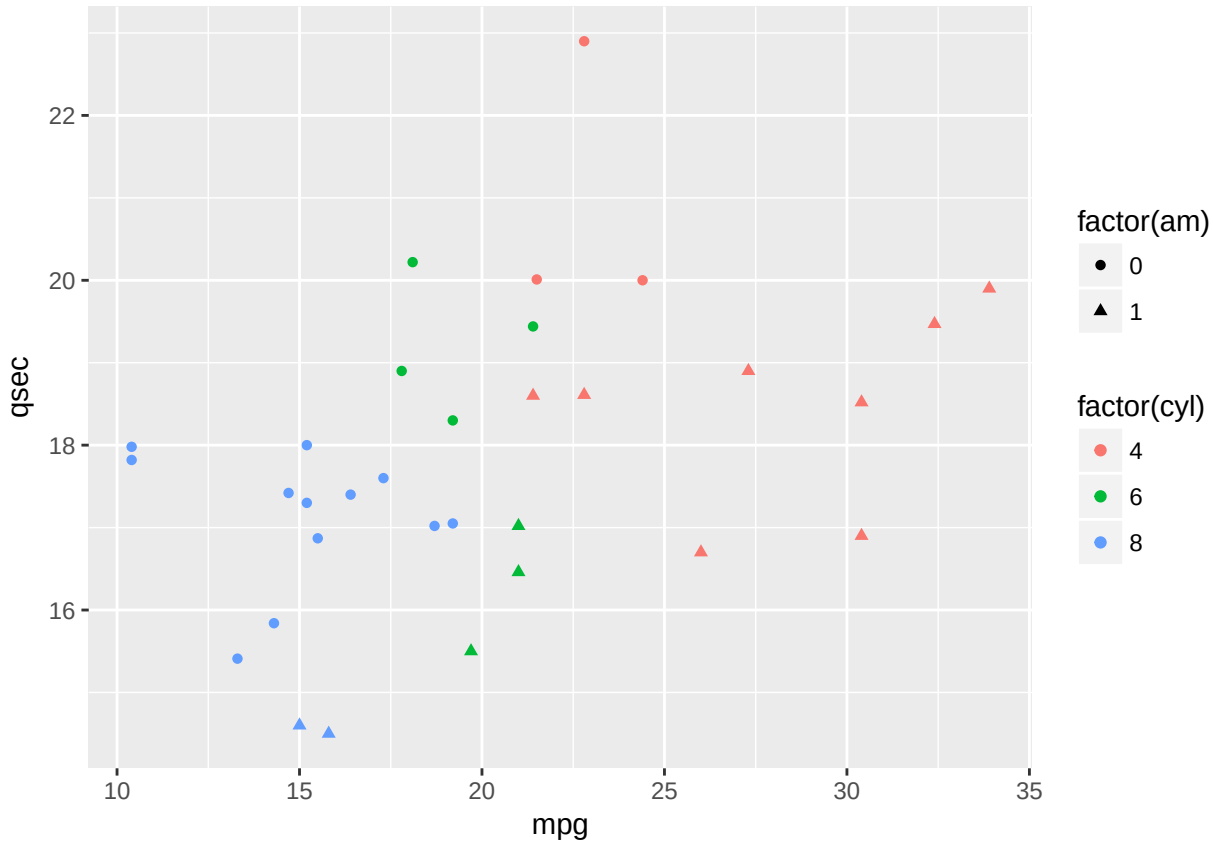
Expand the previous plot to include `factor(am)` as the `shape` of the points.

Expand the previous plot to include the ratio of horsepower to weight (i.e. `(hp/wt)`) as the size of the points.

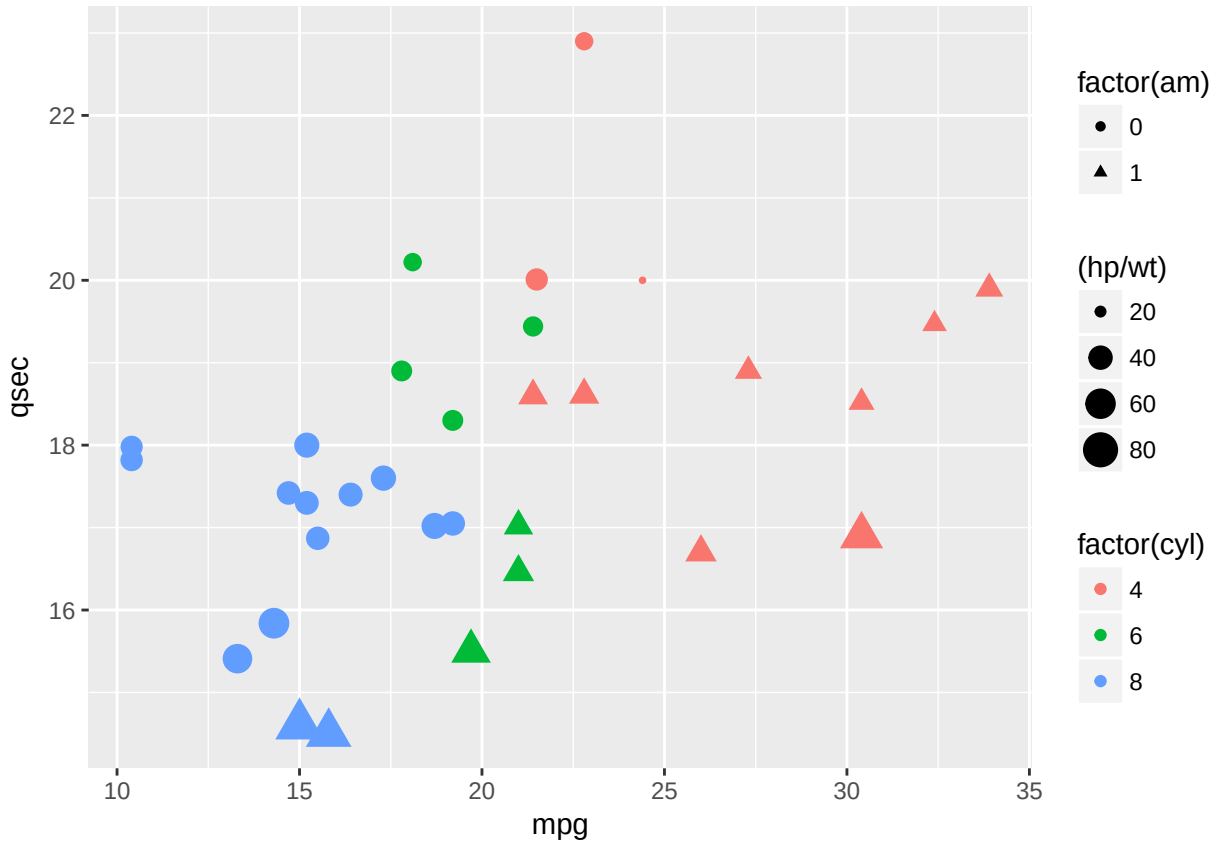
```
# Map mpg onto x, qsec onto y and factor(cyl) onto col
ggplot(mtcars, aes(x = mpg, y = qsec, col = factor(cyl))) +
  geom_point()
```




```
# Add mapping: factor(am) onto shape
ggplot(mtcars, aes(x = mpg, y = qsec, col = factor(cyl), shape = factor(am))) +
  geom_point()
```




```
# Add mapping: (hp/wt) onto size
ggplot(mtcars, aes(x = mpg, y = qsec, col = factor(cyl), shape = factor(am), size = (hp/wt))) +
  geom_point()
```



Aesthetics for categorical and continuous variables

Many of the aesthetics can be mapped onto continuous or categorical variables, but some are restricted to categorical data. Which aesthetics are they?

INSTRUCTIONS

Possible Answers (Correct answer is **Bolded**)

color & fill

alpha & size

label & shape

alpha & label

x & y

Position

You saw how jittering worked in the video, but bar plots suffer from their own issues of overplotting, as you'll see here. Use the `"stack"`, `"fill"` and `"dodge"` positions to reproduce the plot in the viewer.

The `ggplot2` base layers (data and aesthetics) have already been coded; they're stored in a variable `cyl.am`. It looks like this:

```
cyl.am <- ggplot(mtcars, aes(x = factor(cyl), fill = factor(am)))
```

INSTRUCTIONS

Add a `geom_bar()` call to `cyl.am`. By default, the `position` will be set to `"stack"`.

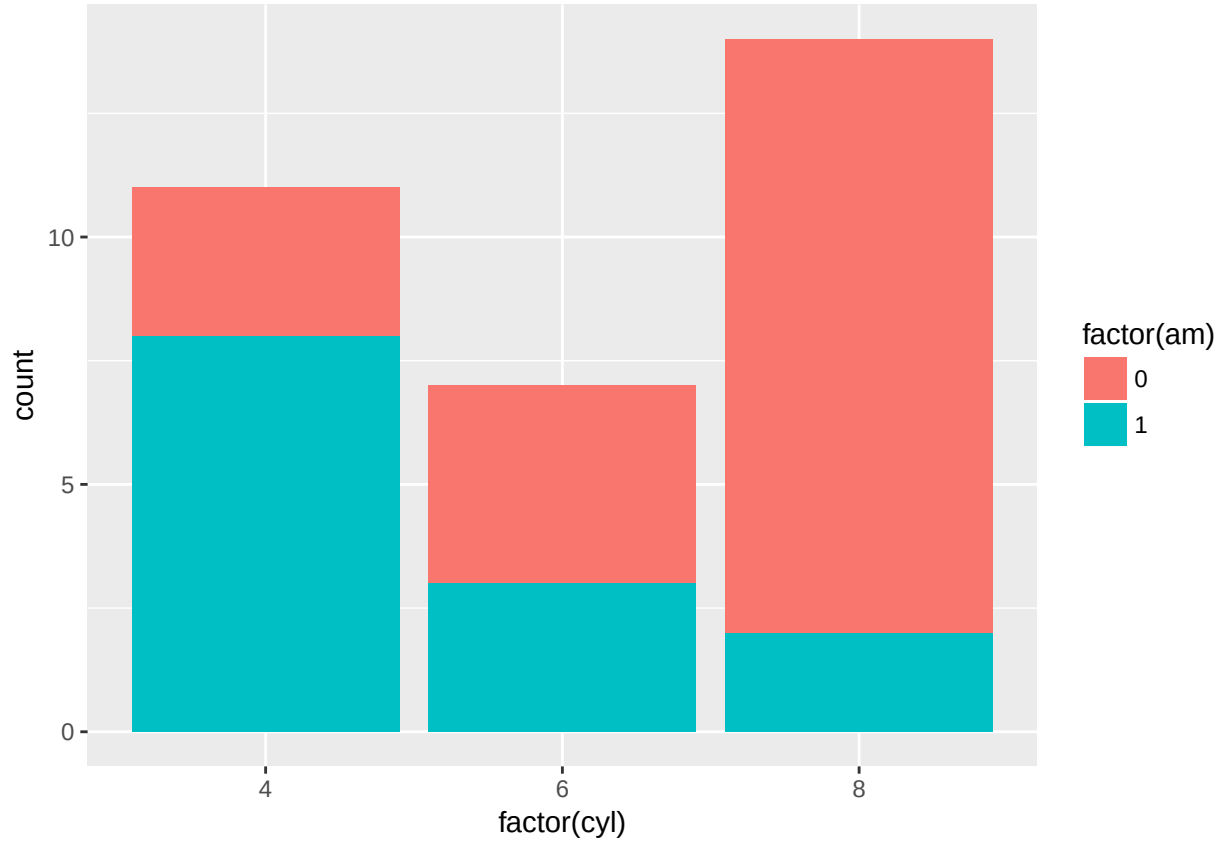
Fill in the second `ggplot` command. Explicitly set position to `"fill"` inside `geom_bar()`.

Fill in the third `ggplot` command. Set `position` to `"dodge"`.

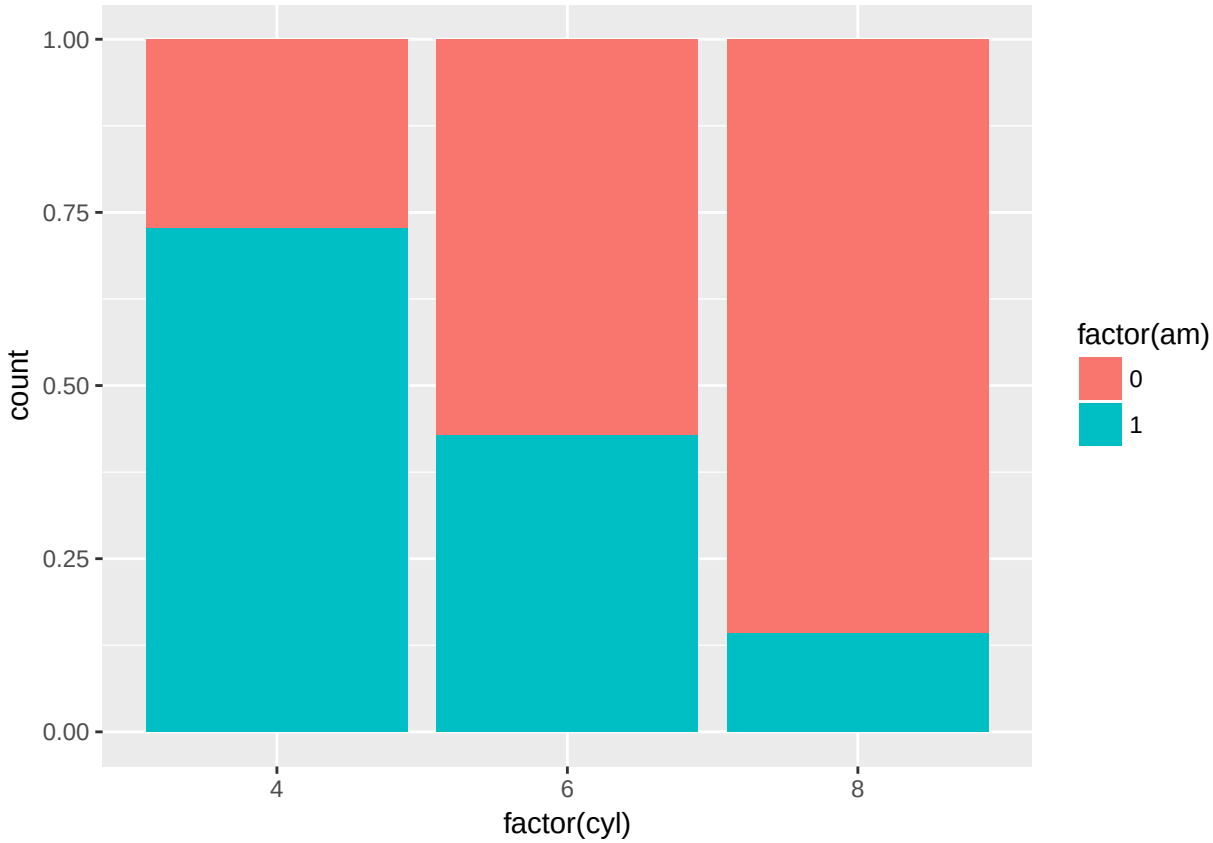
The `position = "dodge"` version seems most appropriate. Finish off the fourth `ggplot` command by completing the three `scale_` functions:

- `scale_x_discrete()`takes as its only argument the x-axis label: `"Cylinders"`.
- 1scale_y_continuous()`takes as its only argument the y-axis label:`"Number"‘.
- `scale_fill_manual()` fixes the legend. The first argument is the title of the legend: `"Transmission"`. Next, `values` and `labels` are set to predefined values for you. These are the colors and the labels in the legend.

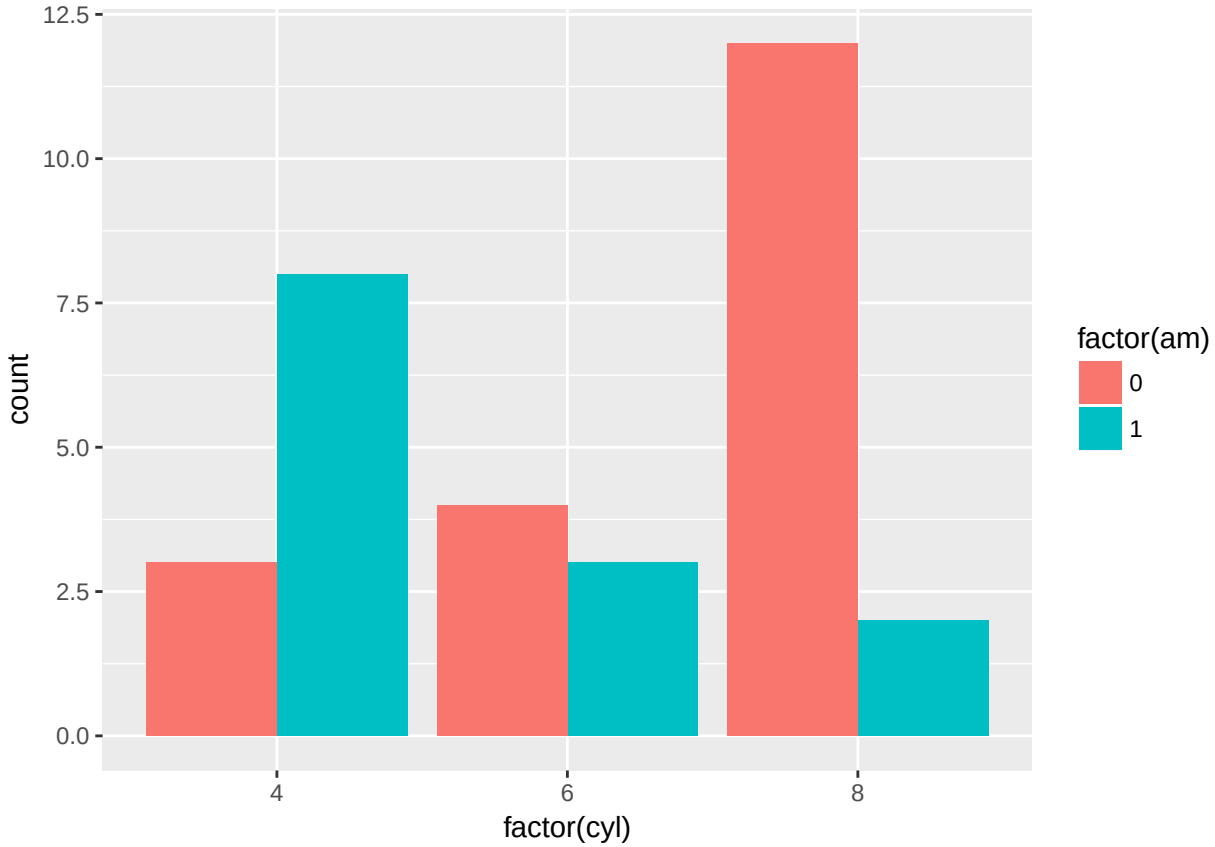
```
# The base layer, cyl.am, is available for you
# Add geom (position = "stack" by default)
cyl.am +
  geom_bar()
```



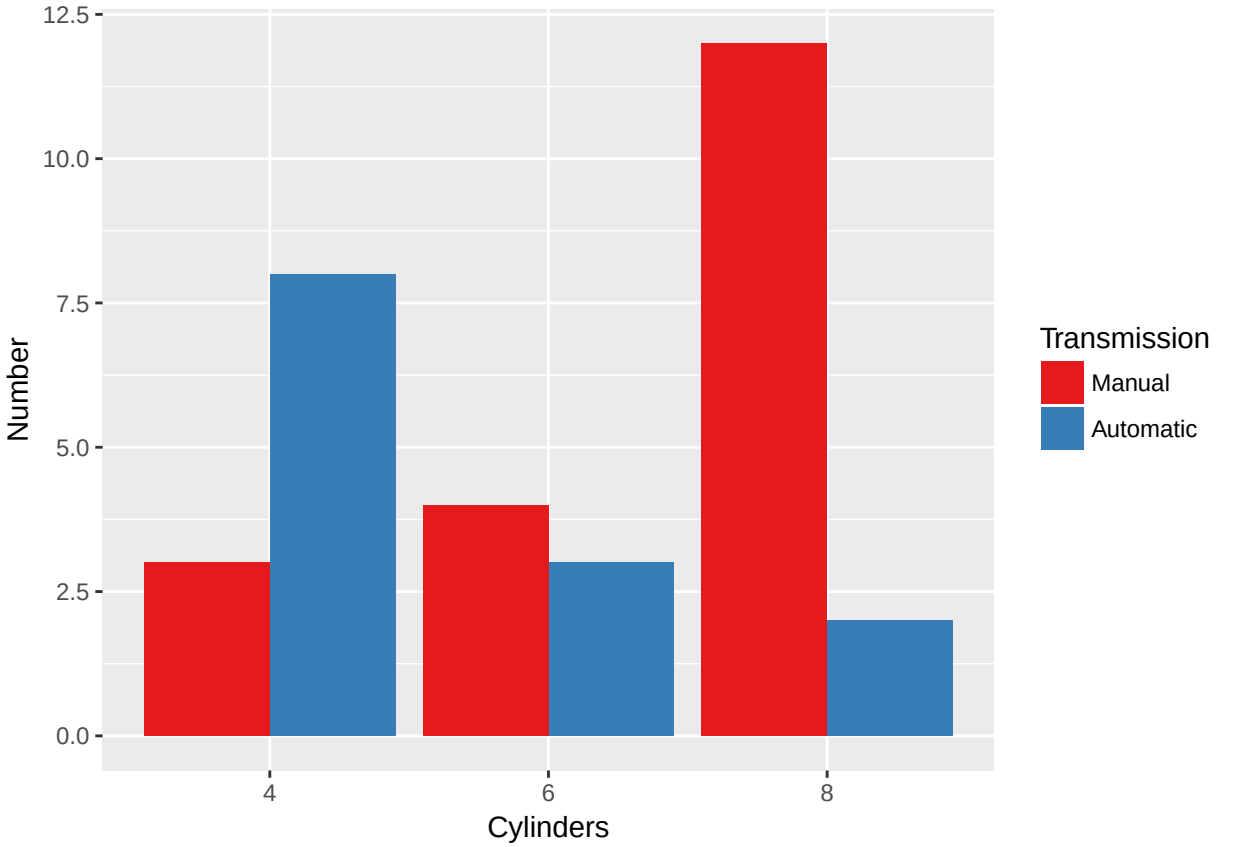

```
# Fill - show proportion
cyl.am +
  geom_bar(position = "fill")
```




```
# Dodging - principles of similarity and proximity
cyl.am +
  geom_bar(position = "dodge")
```




```
# Clean up the axes with scale_ functions
val = c("#E41A1C", "#377EB8")
lab = c("Manual", "Automatic")
cyl.am +
  geom_bar(position = "dodge") +
  scale_x_discrete("Cylinders") +
  scale_y_continuous("Number") +
  scale_fill_manual("Transmission",
                    values = val,
                    labels = lab)
```



Setting a dummy aesthetic

In the last chapter you saw that all the visible aesthetics can serve as attributes and aesthetics, but I very conveniently left out x and y. That's because although you can make univariate plots (such as histograms, which you'll get to in the next chapter), a y-axis will always be provided, even if you didn't ask for it.

In the `base` package you can make univariate plots with `stripchart()` (shown in the viewer) directly and it will take care of a fake y axis for us. Since this is univariate data, there is no real y axis.

You can get the same thing in `ggplot2`, but it's a bit more cumbersome. The only reason you'd really want to do this is if you were making many plots and you wanted them to be in the same style, or you wanted to take advantage of an aesthetic mapping (e.g. colour).

INSTRUCTIONS

Try to run

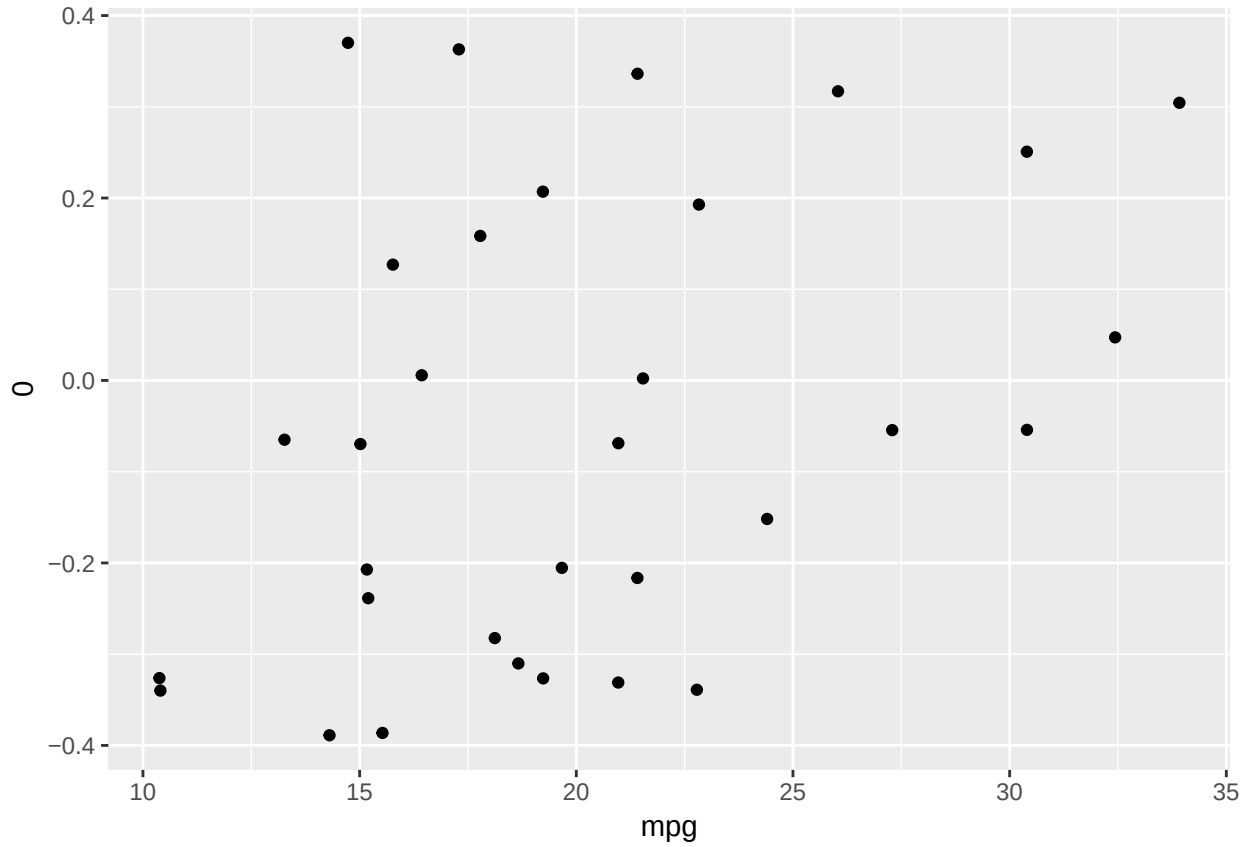
```
ggplot(mtcars, aes(x = mpg)) + geom_point()
```

in the console. `x` is only one of the two essential aesthetics for `geom_point()`, which is why you get an error message.

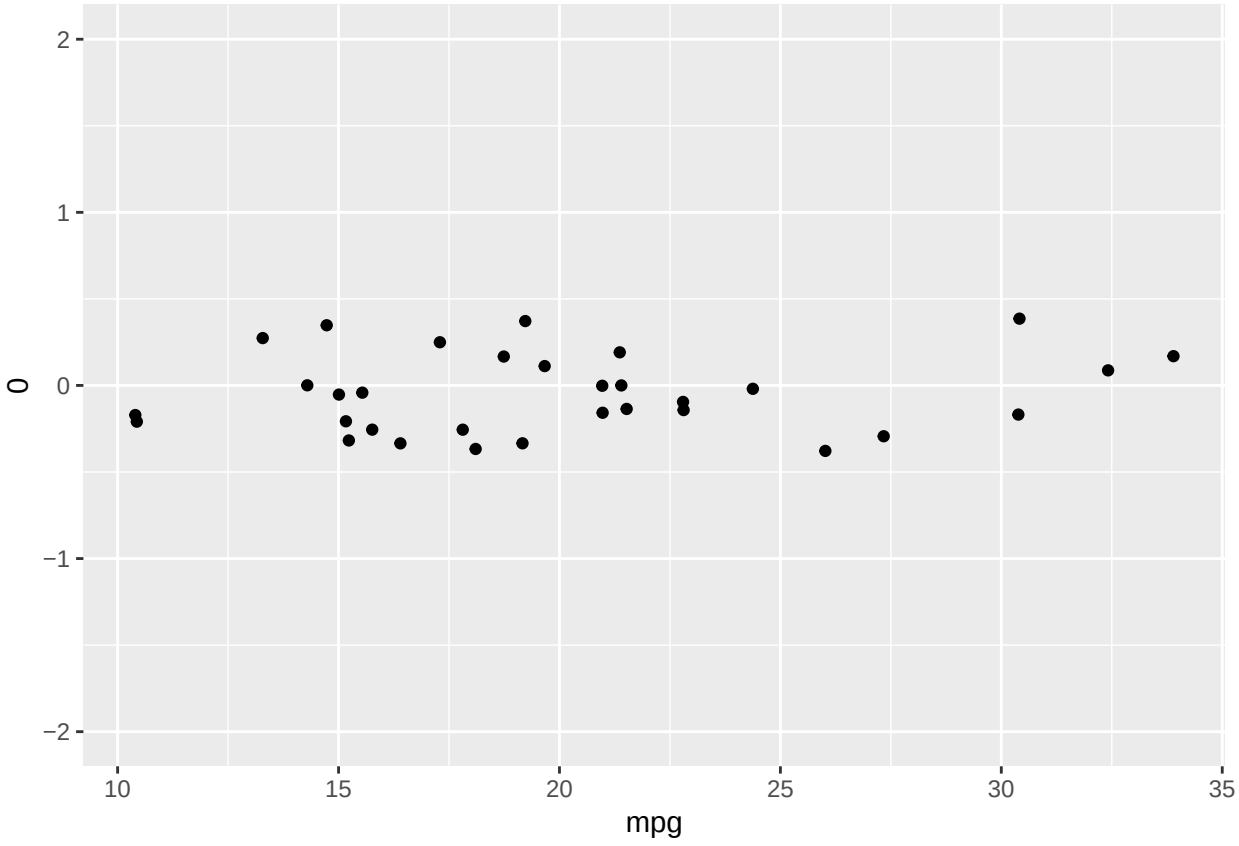
1 - To fix this, map a value, e.g. `0`, instead of a variable, onto `y`. Use `geom_jitter()` to avoid having all the points on a horizontal line.

2 - To make everything look nicer, copy & paste the code for plot 1 and change the limits of the y axis using the appropriate `scale_y_...()` function. Set the `limits argument` to `c(-2, 2)`.

```
# 1 - Create jittered plot of mtcars, mpg onto x, 0 onto y
ggplot(mtcars, aes(x = mpg, y = 0)) +
  geom_jitter()
```




```
# 2 - Add function to change y axis limits
ggplot(mtcars, aes(x = mpg, y = 0)) +
  geom_jitter() +
  scale_y_continuous(limits = c(-2,2))
```



Overplotting 1 - Point shape and transparency

In the previous section you saw that there are lots of ways to use aesthetics. Perhaps too many, because although they are possible, they are not all recommended. Let's take a look at what works and what doesn't.

So far you've focused on scatter plots since they are intuitive, easily understood and very common. A major consideration in any scatter plot is dealing with overplotting. You'll encounter this topic again in the geometries layer, but you can already make some adjustments here.

You'll have to deal with overplotting when you have:

- 1. Large datasets,
- 2. Imprecise data and so points are not clearly separated on your plot (you saw this in the video with the `iris` dataset),
- 3. Interval data (i.e. data appears at fixed values), or
- 4. Aligned data values on a single axis.

One very common technique that I'd recommend to always use when you have solid shapes it to use alpha blending (i.e. adding transparency). An alternative is to use hollow shapes. These are adjustments to make before even worrying about positioning. This addresses the first point as above, which you'll see again in the next exercise.

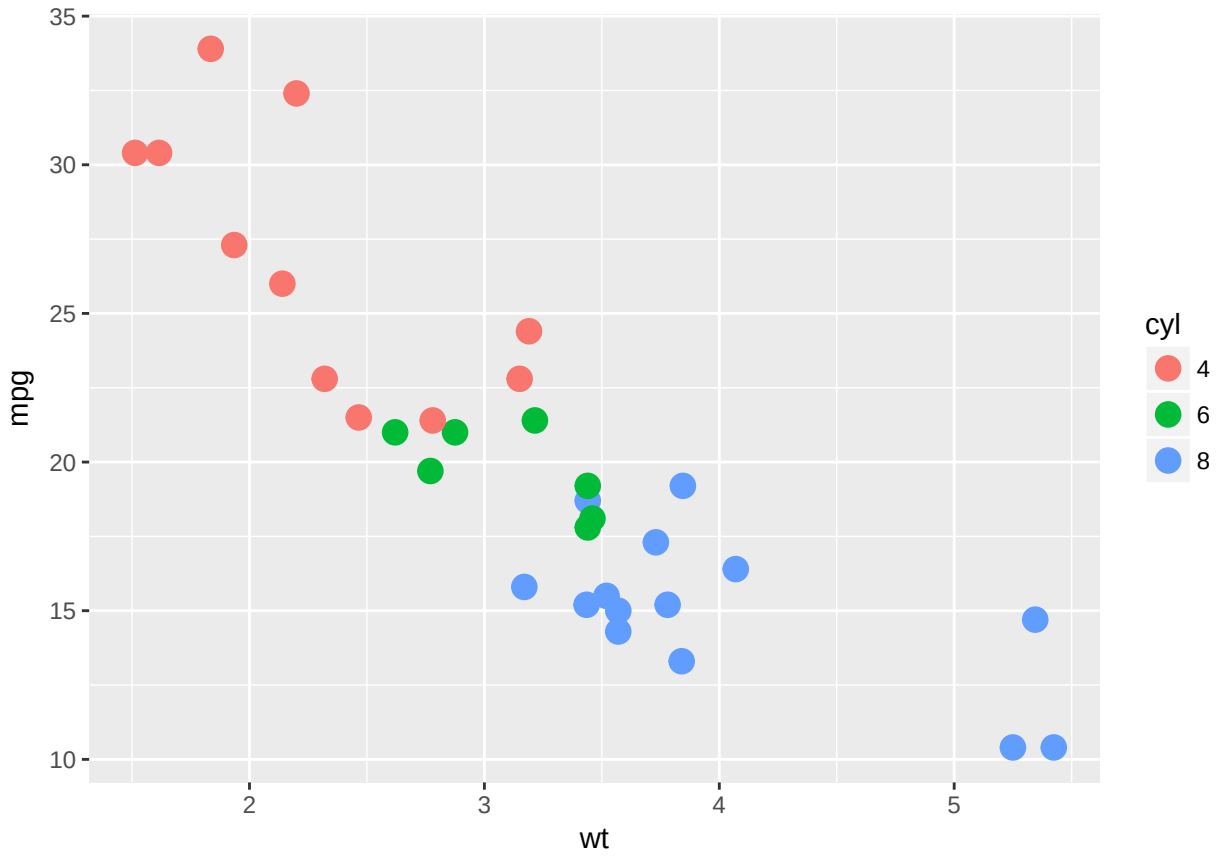
INSTRUCTIONS

Begin by making a basic scatter plot of `mpg` (y) vs. `wt` (x), map `cyl` to `color` and make the `size = 4`. `cyl` has already been converted to a factor variable for you.

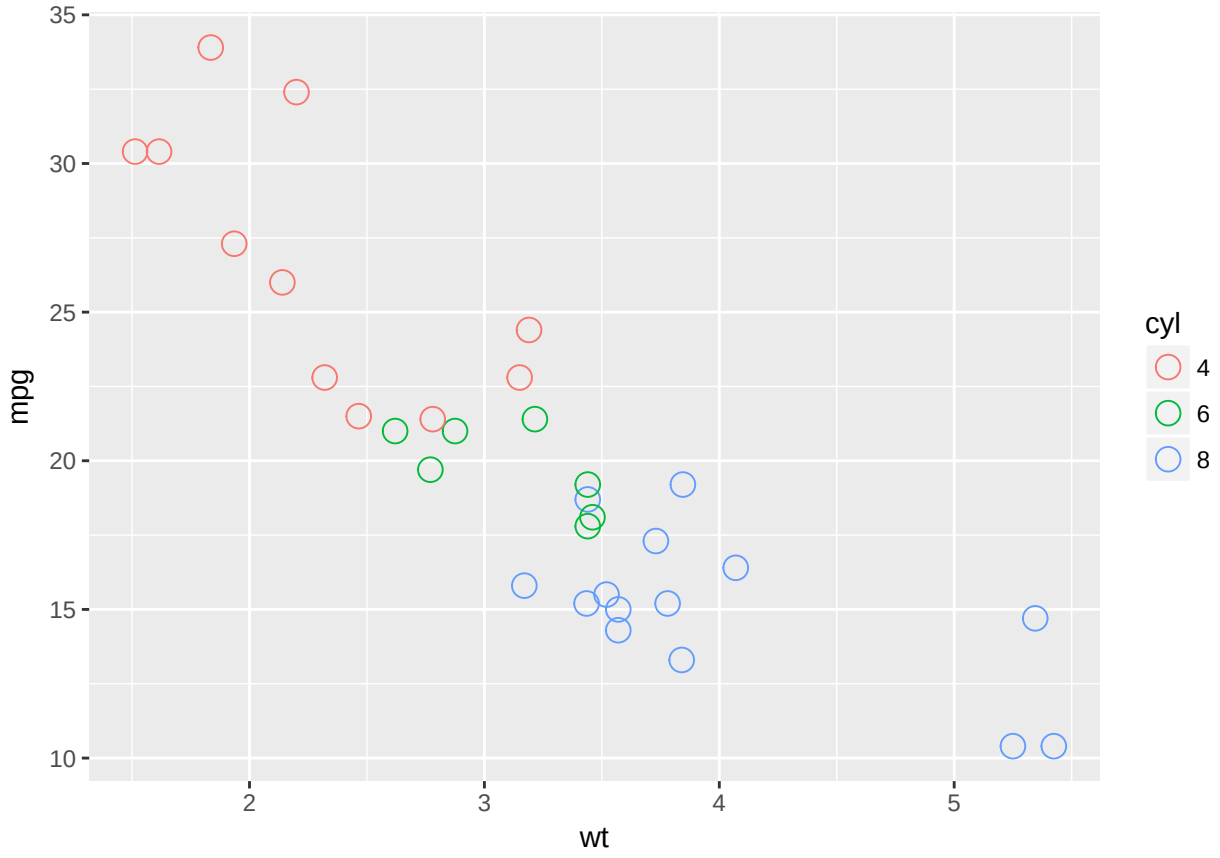
Modify the above plot to set `shape` to `1`. This allows for hollow circles.

Modify the first plot to set `alpha` to `0.6`.

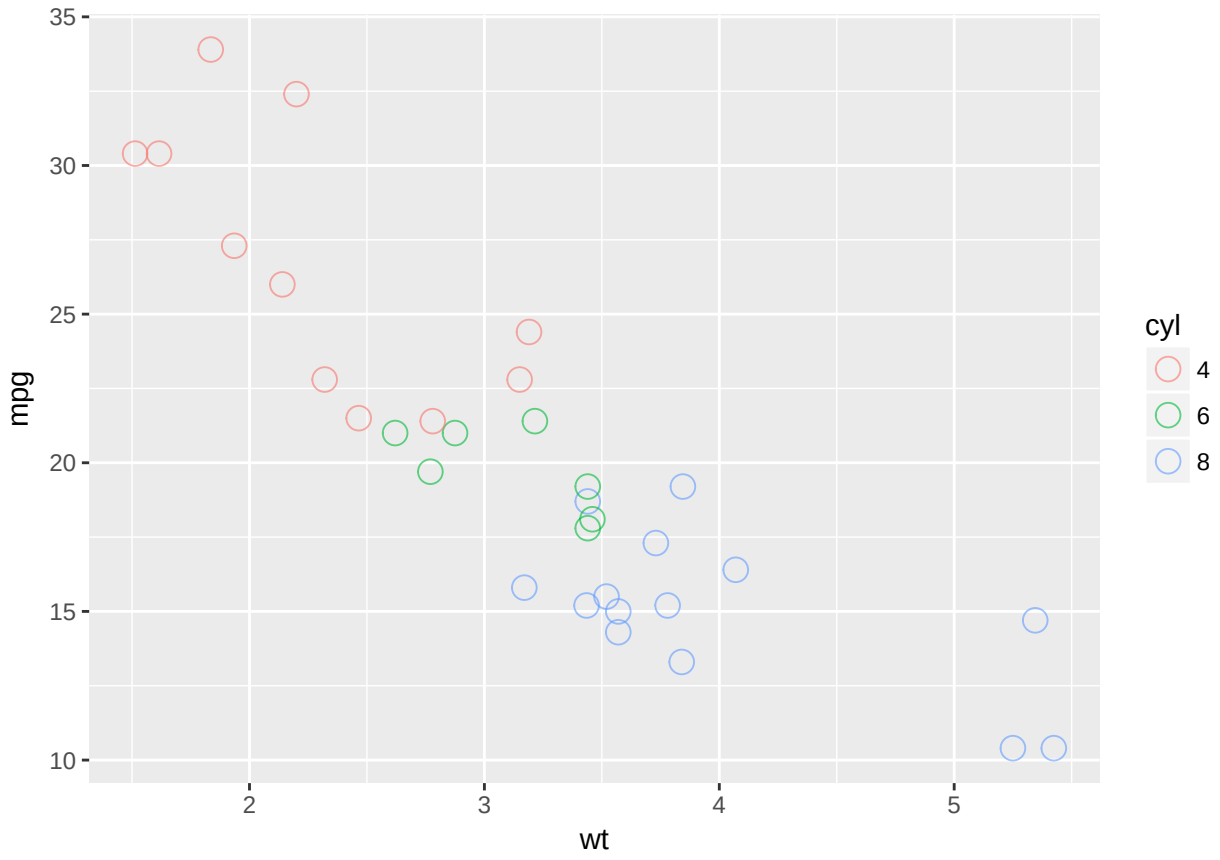
```
# Basic scatter plot: wt on x-axis and mpg on y-axis; map cyl to col
ggplot(mtcars, aes(x = wt, y = mpg, color = cyl)) +
  geom_point(size = 4)
```




```
# Hollow circles - an improvement
ggplot(mtcars, aes(x = wt, y = mpg, color = cyl)) +
  geom_point(size = 4, shape = 1)
```




```
# Add transparency - very nice
ggplot(mtcars, aes(x = wt, y = mpg, color = cyl)) +
  geom_point(size = 4, shape = 1, alpha = 0.6)
```



Overplotting 2 - alpha with large datasets

In a previous exercise we defined four situations in which you'd have to adjust for overplotting. You'll consider the last two here with the `diamonds` dataset:

- 1. Large datasets.
- 2. Aligned data values on a single axis

INSTRUCTIONS

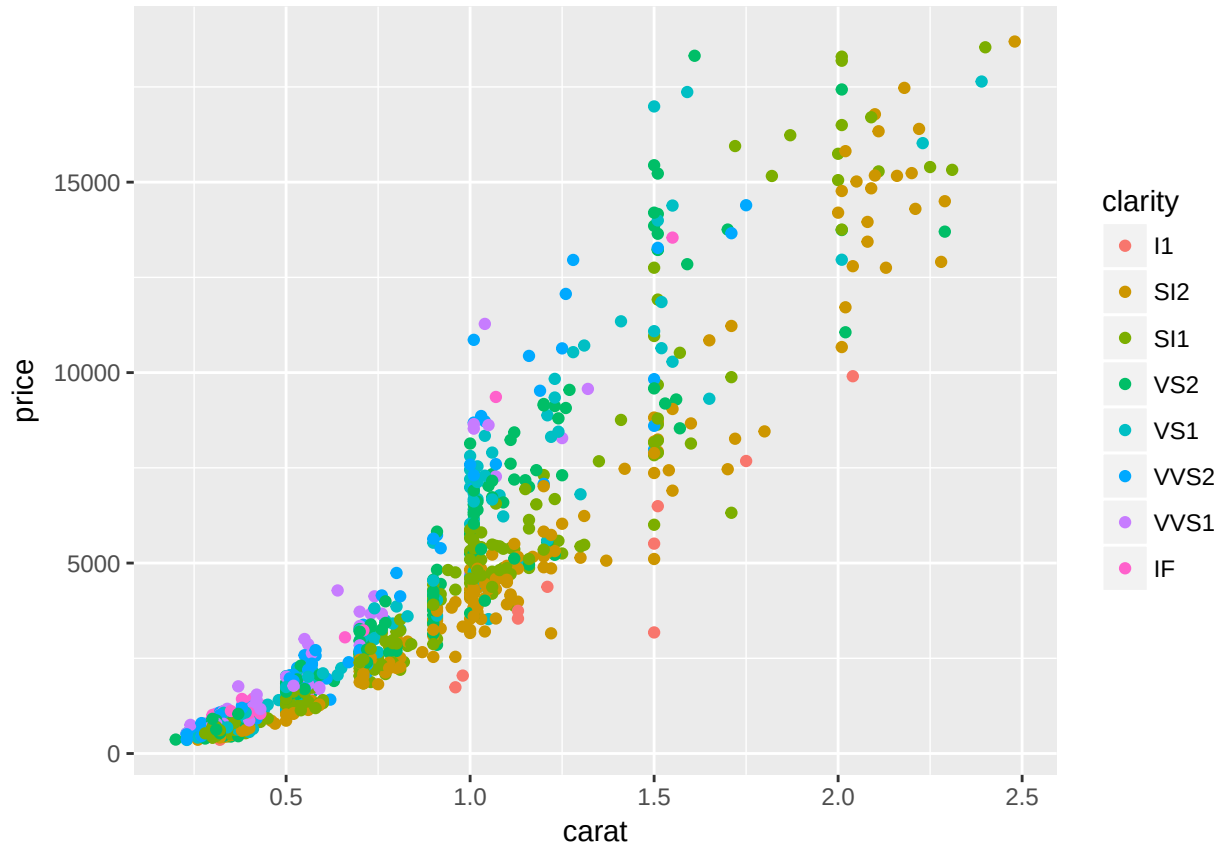
The `diamonds` data frame is available in the `ggplot2()` package. Begin by making a basic scatter plot of `price` (y) vs. `carat` (x) and map `clarity` onto `color`.

Copy the above functions and set the `alpha` to `0.5`. This is a good start to dealing with the large dataset.

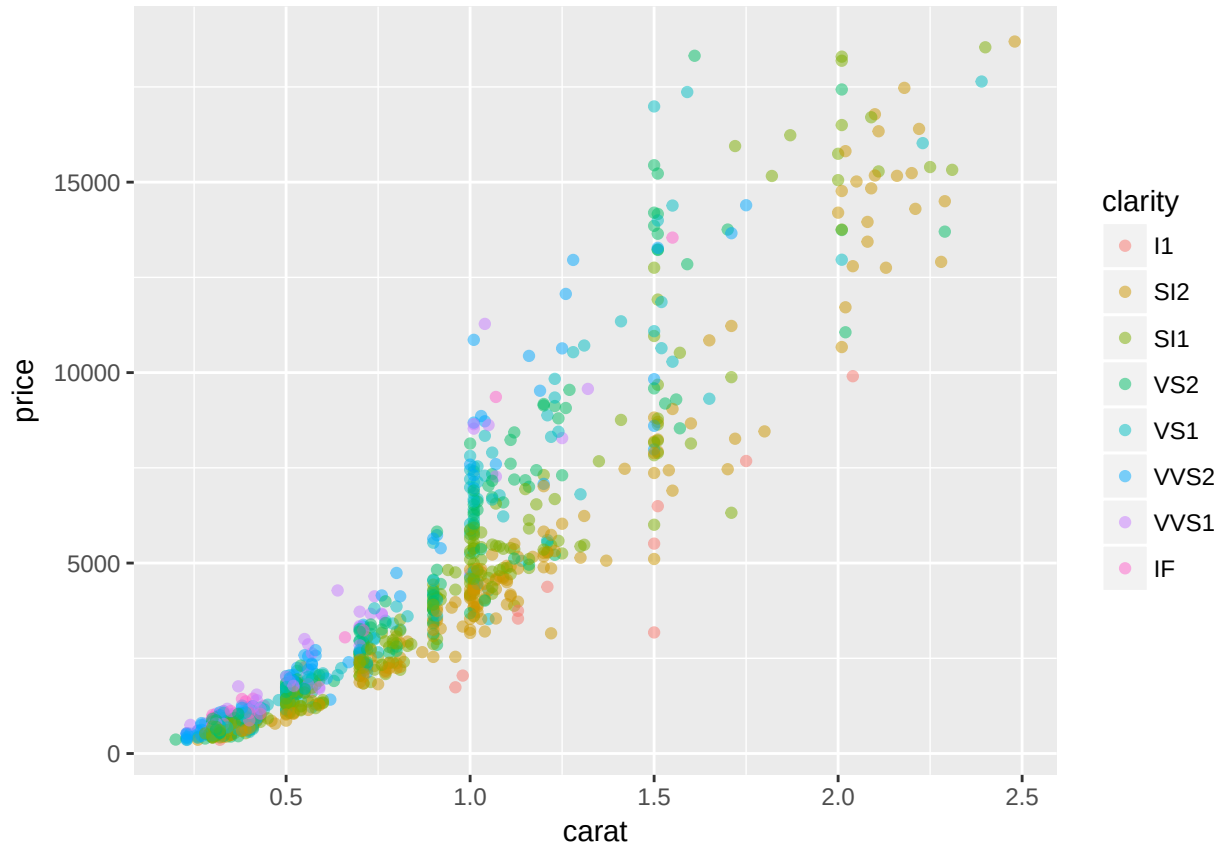
Align all the diamonds within a `clarity` class, by plotting `carat` (y) vs. `clarity` (x). Map `price` onto `color`. `alpha` should still be `0.5`.

In the previous plot, all the individual values line up on a single axis within each `clarity` category, so you have not overcome overplotting. Modify the above plot to use the `position = "jitter"` inside `geom_point()`.

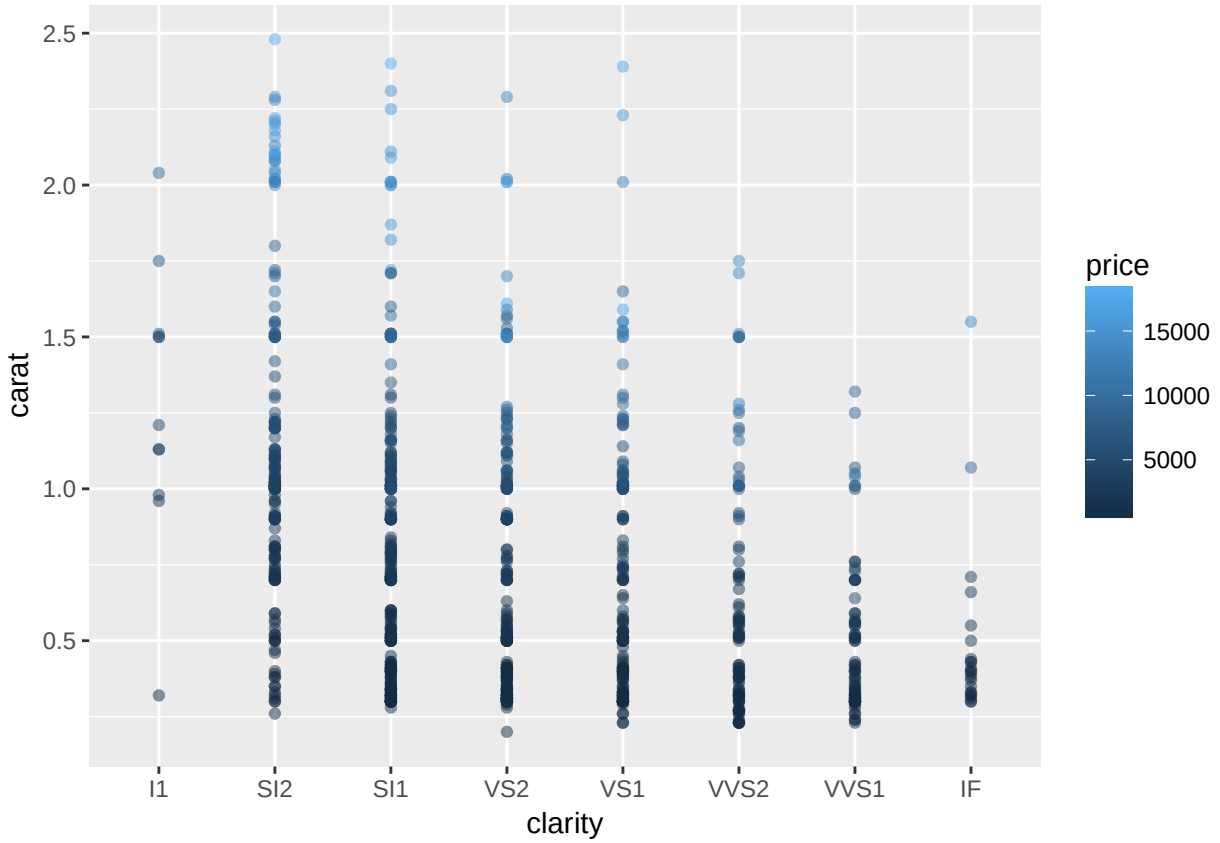
```
# Scatter plot: carat (x), price (y), clarity (color)
ggplot(diamonds, aes(x = carat, y = price, color = clarity)) +
  geom_point()
```



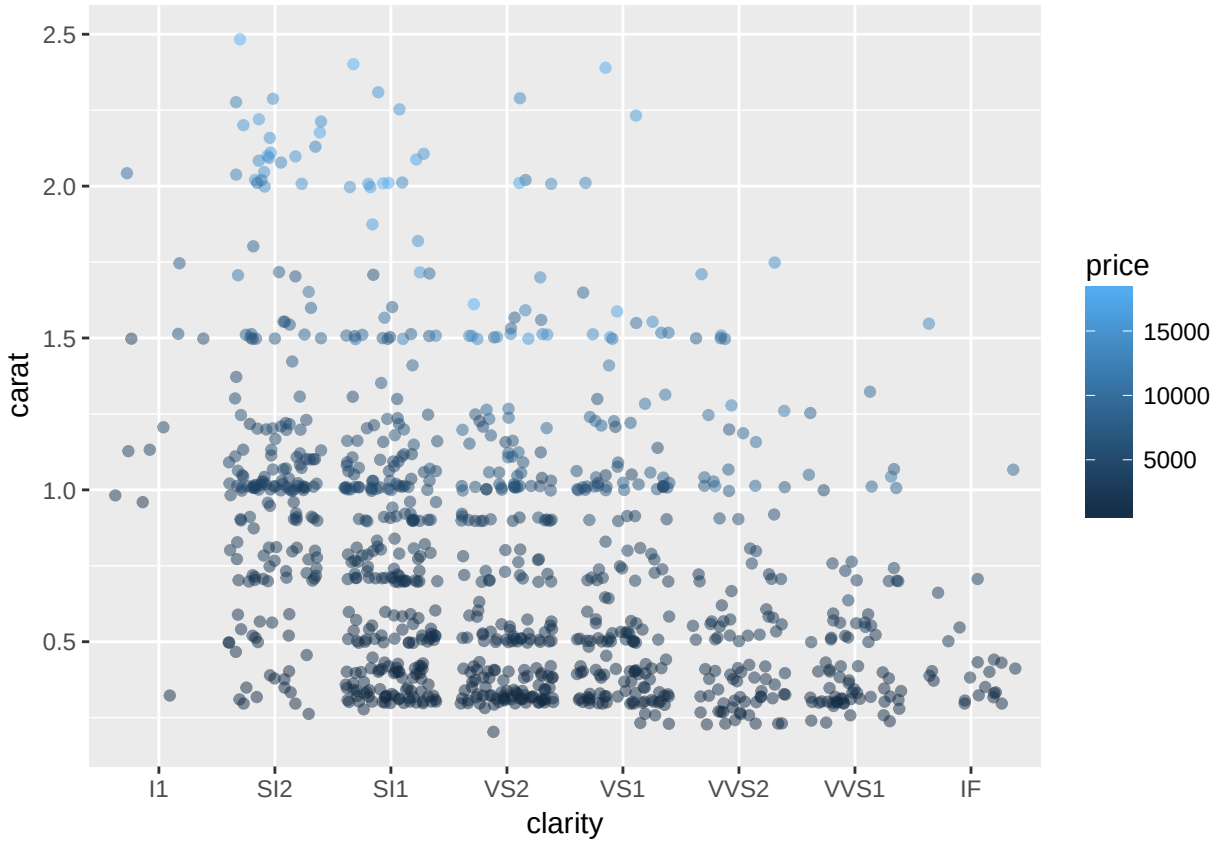

```
# Adjust for overplotting
ggplot(diamonds, aes(x = carat, y = price, color = clarity)) +
  geom_point(alpha = 0.5)
```




```
# Scatter plot: clarity (x), carat (y), price (color)
ggplot(diamonds, aes(x = clarity, y = carat, color = price)) +
  geom_point(alpha = 0.5)
```




```
# Dot plot with jittering
ggplot(diamonds, aes(x = clarity, y = carat, color = price)) +
  geom_point(alpha = 0.5, position = "jitter")
```



Chapter 4: Geometries

A plot's geometry dictates what visual elements will be used. In this chapter, we'll familiarize you with the geometries used in the three most common plot types you'll encounter - scatter plots, bar charts and line plots. We'll look at a variety of different ways to construct these plots.

Scatter plots and jittering (1)

You already saw a few examples using `geom_point()` where the result was not a scatter plot. For example, in the plot shown in the viewer a continuous variable, `wt`, is mapped to the `y` aesthetic, and a categorical variable, `cyl`, is mapped to the `x` aesthetic. This also leads to over-plotting, since the points are arranged on a single x position. You previously dealt with overplotting by setting the `position = jitter` inside `geom_point()`. Let's look at some other solutions here.

INSTRUCTIONS

Beginning with the code for the plot in the viewer (given), make these modifications

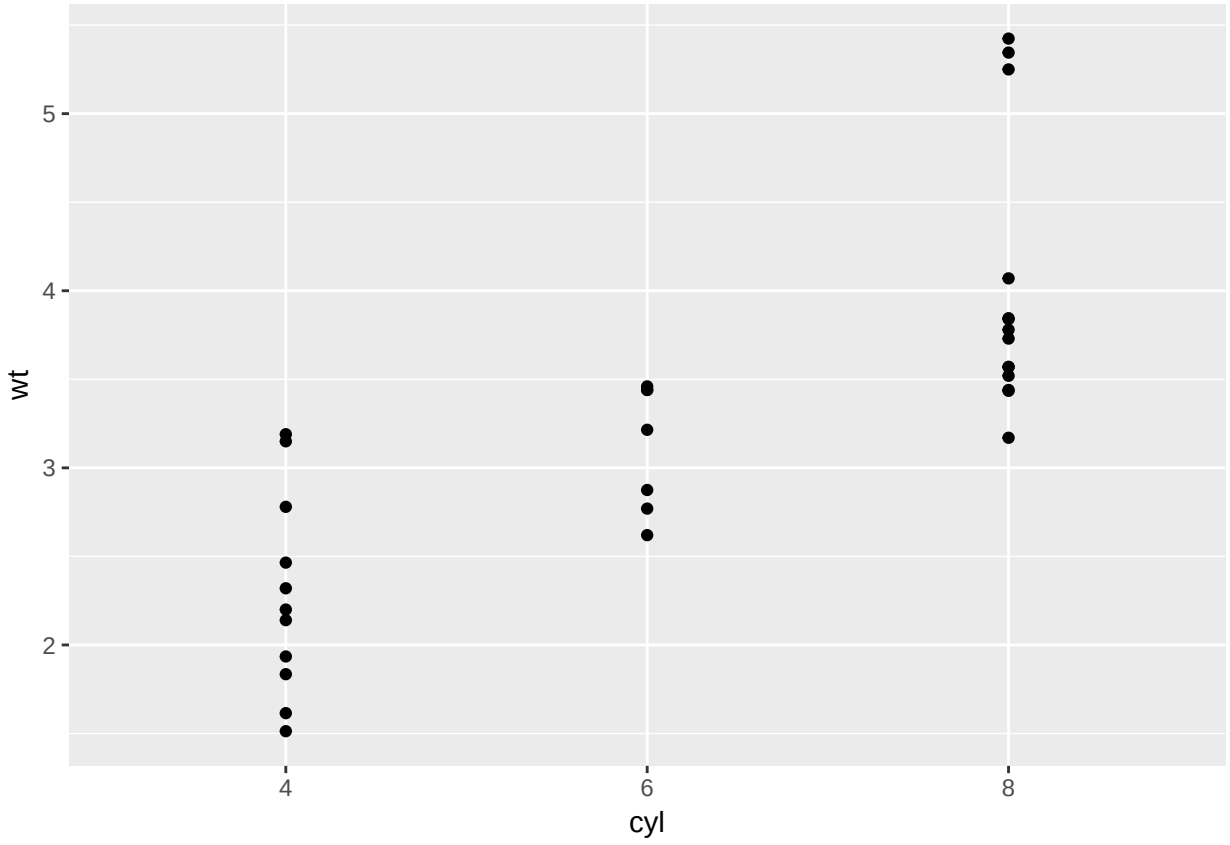
1 - Use a shortcut geom, `geom_jitter()`, instead of `geom_point()`.

2 - Unfortunately, the width of the jitter is a bit too wide to be useful. Adjust this by setting the argument width = 0.1 inside `geom_jitter()`.

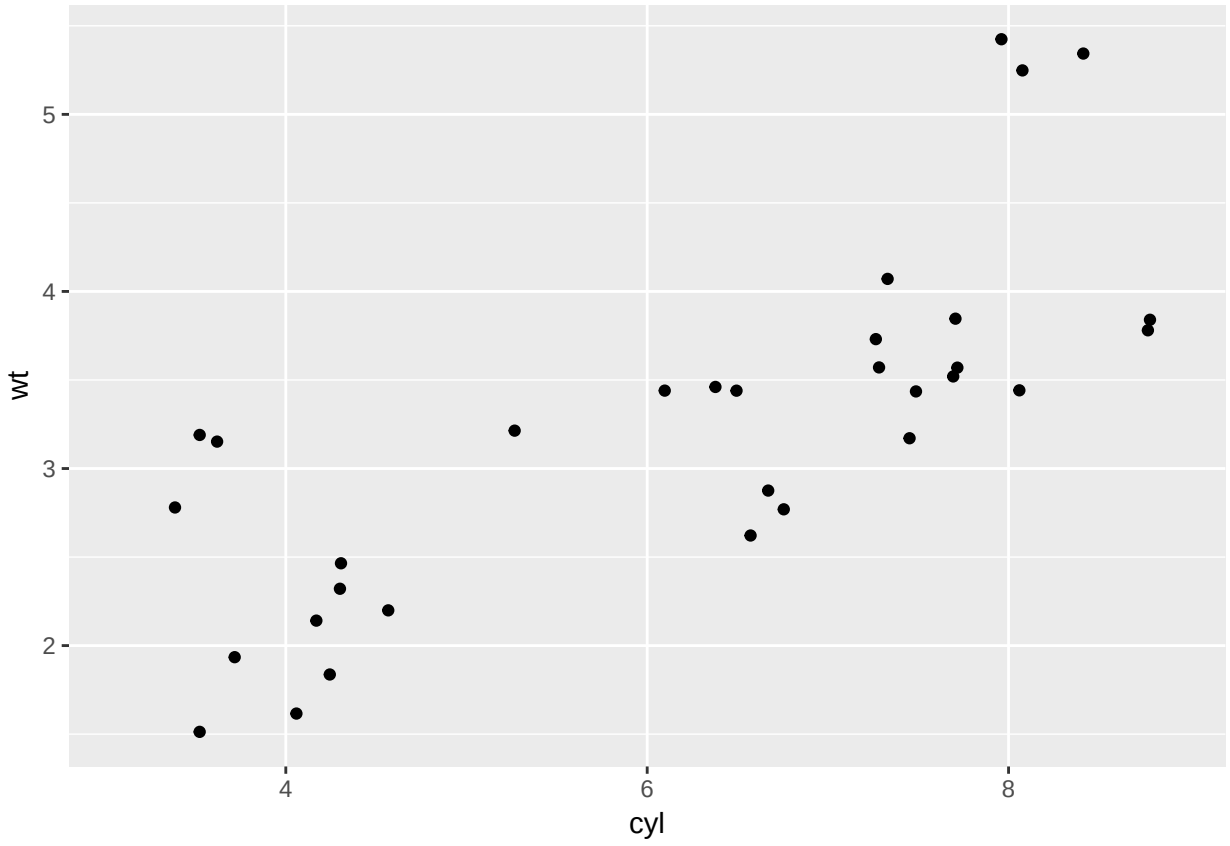
3 - Finally, return to `geom_point()` and set the position argument here to `position_jitter(0.1)`, which will set the jittering width directly inside a points layer.

Note: For convenience, you could have saved the data and aesthetic layers as a `ggplot2` object and re-used it in all solutions. We've made each plot explicit so that you can see all plotting instructions.

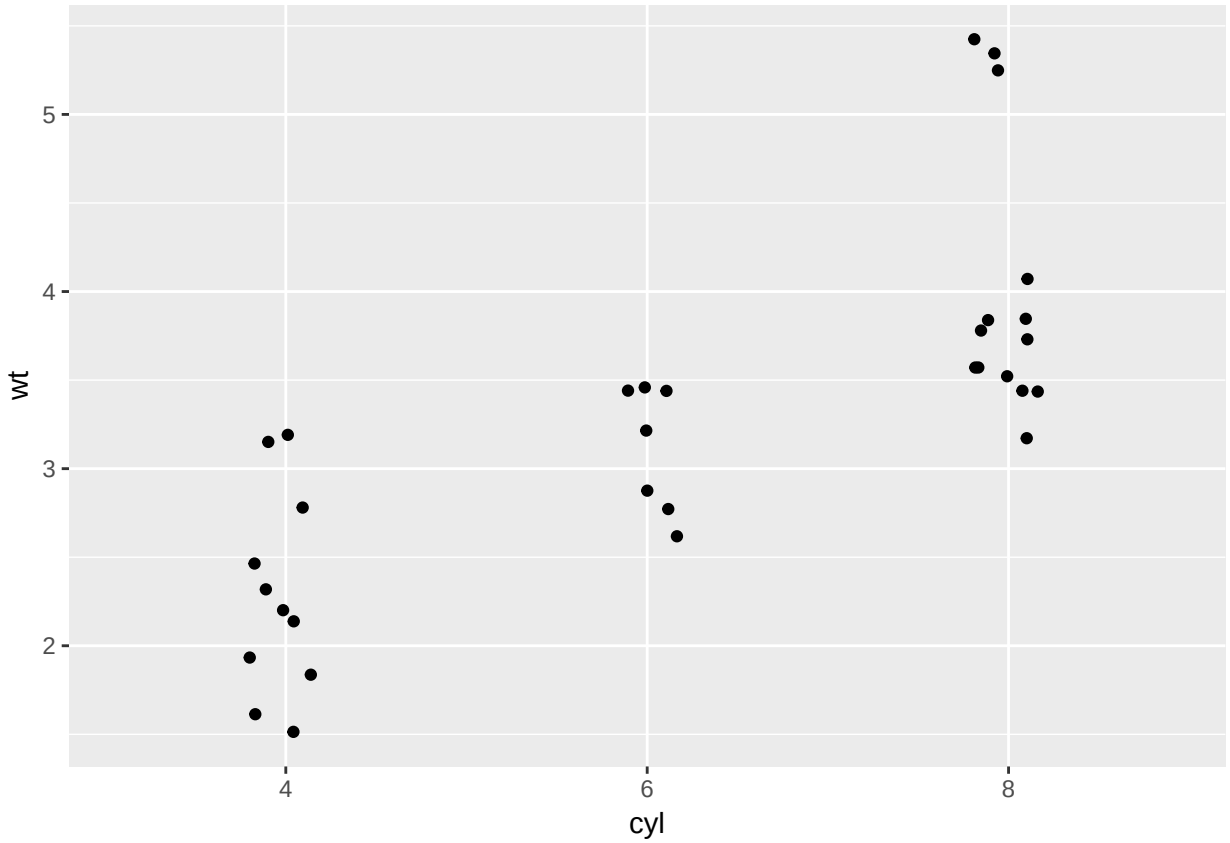
```
# Shown in the viewer:
ggplot(mtcars, aes(x = cyl, y = wt)) +
  geom_point()
```



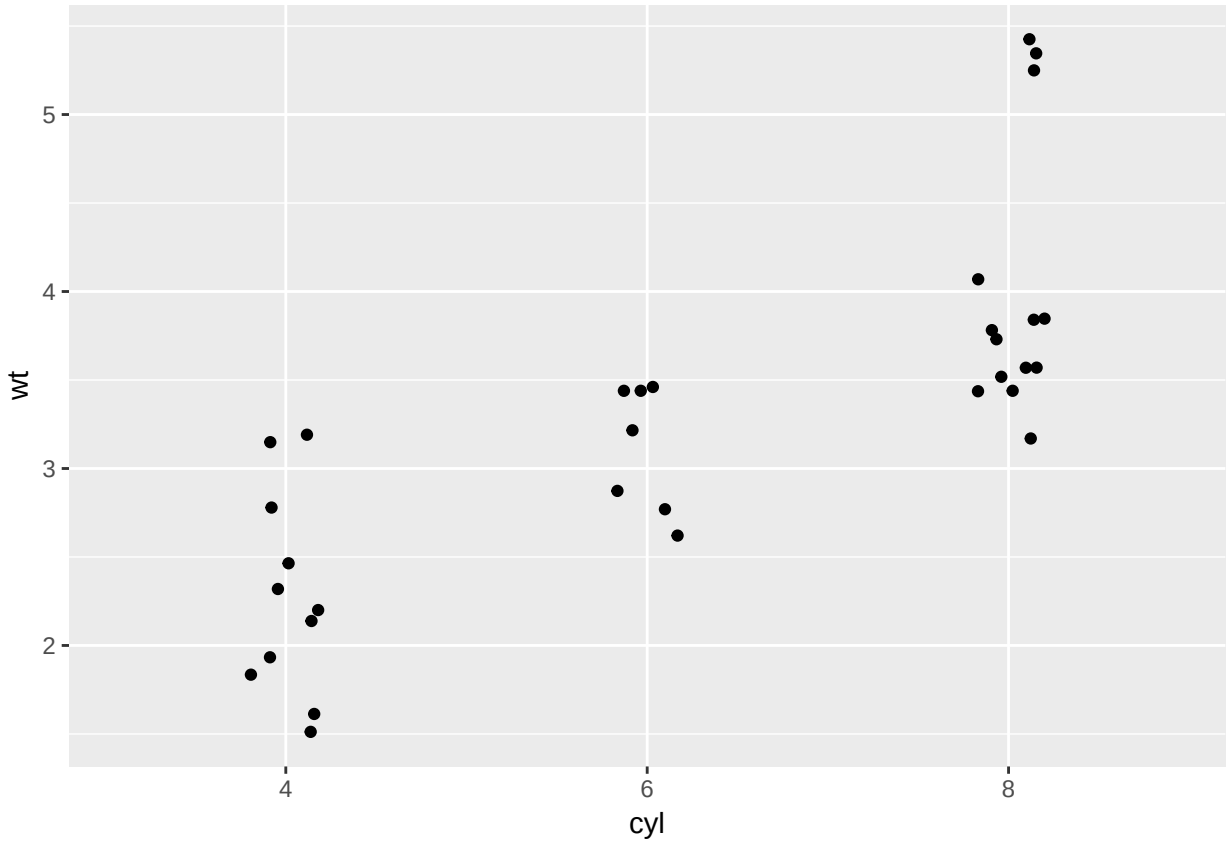

```
# Solutions:
# 1 - With geom_jitter()
ggplot(mtcars, aes(x = cyl, y = wt)) +
  geom_jitter()
```




```
# 2 - Set width in geom_jitter()
ggplot(mtcars, aes(x = cyl, y = wt)) +
  geom_jitter(width = 0.1)
```




```
# 3 - Set position = position_jitter() in geom_point() ()
ggplot(mtcars, aes(x = cyl, y = wt)) +
  geom_point(position = position_jitter(0.1))
```



Scatter plots and jittering (2)

In the chapter on aesthetics you saw different ways in which you will have to compensate for overplotting. In the video you saw a dataset that suffered from overplotting because of the precision of the dataset.

Another example you saw is when you have interval data. This can be continuous data measured on an `interval` (i.e. 1 ,2, 3 ...), as opposed to `numeric` (i.e. 1.1, 1.4, 1.5, ...), scale, or two categorical (e.g. `factor`) variables, which are just type `interval` under-the-hood.

In such a case you'll have a small, defined number of intersections between the two variables.

You will be using the `Vocab` dataset. The `Vocab` dataset contains information about the years of education and integer score on a vocabulary test for over 21,000 individuals based on US General Social Surveys from 1972-2004.

INSTRUCTIONS

The `Vocab` data frame has been loaded for you. Both the `education` and `vocabulary` variables are classified as integers. You can imagine these as factor variables, but here, integers are more convenient to work with. First, get familiar with the dataset by looking at its structure with `str()`.

Make a basic scatter plot of `vocabulary` (y) vs. `education` (x). Here it becomes apparent that you have issues with overplotting because of the integer scales.

Use `geom_jitter()` instead of `geom_point()`.

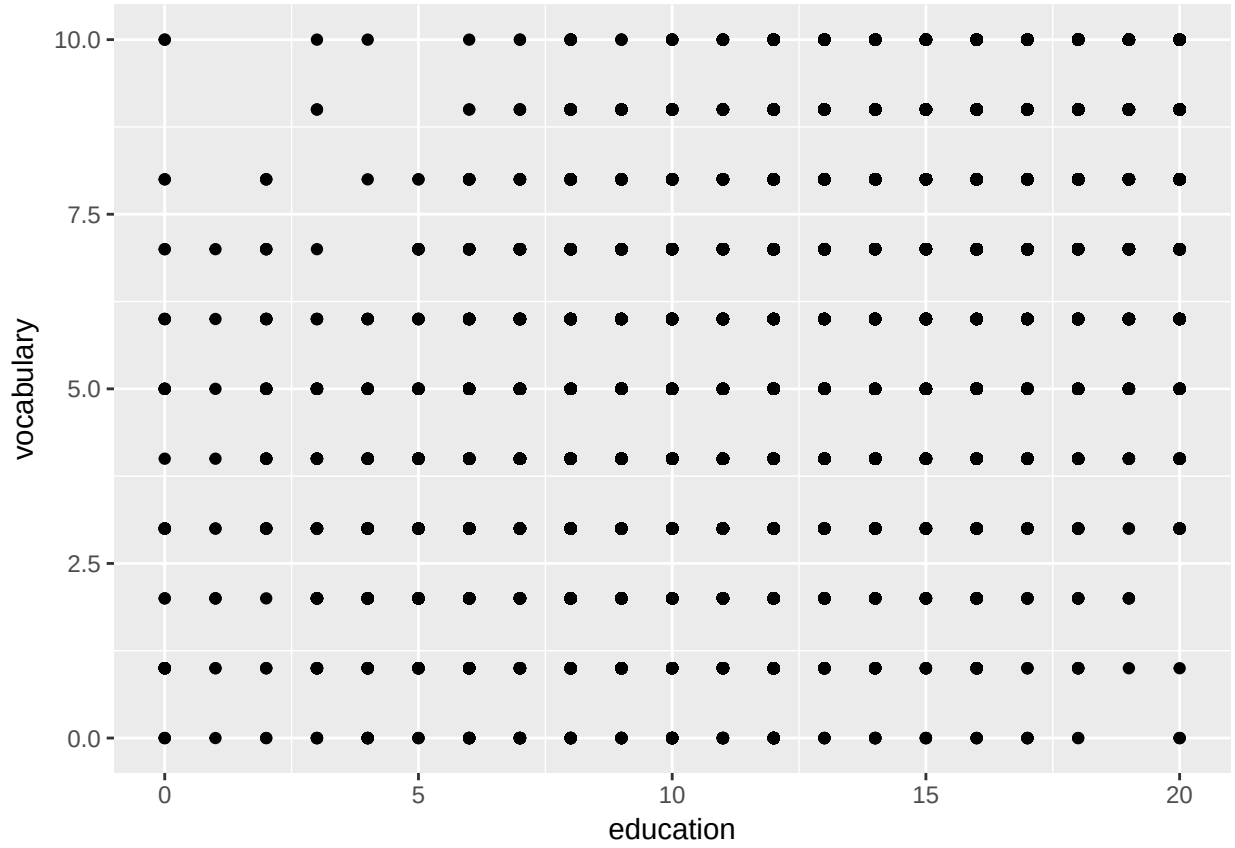
Using the jittered plot, set `alpha` to `0.2` (very low).

Using the jittered plot, set `shape` to `1`.

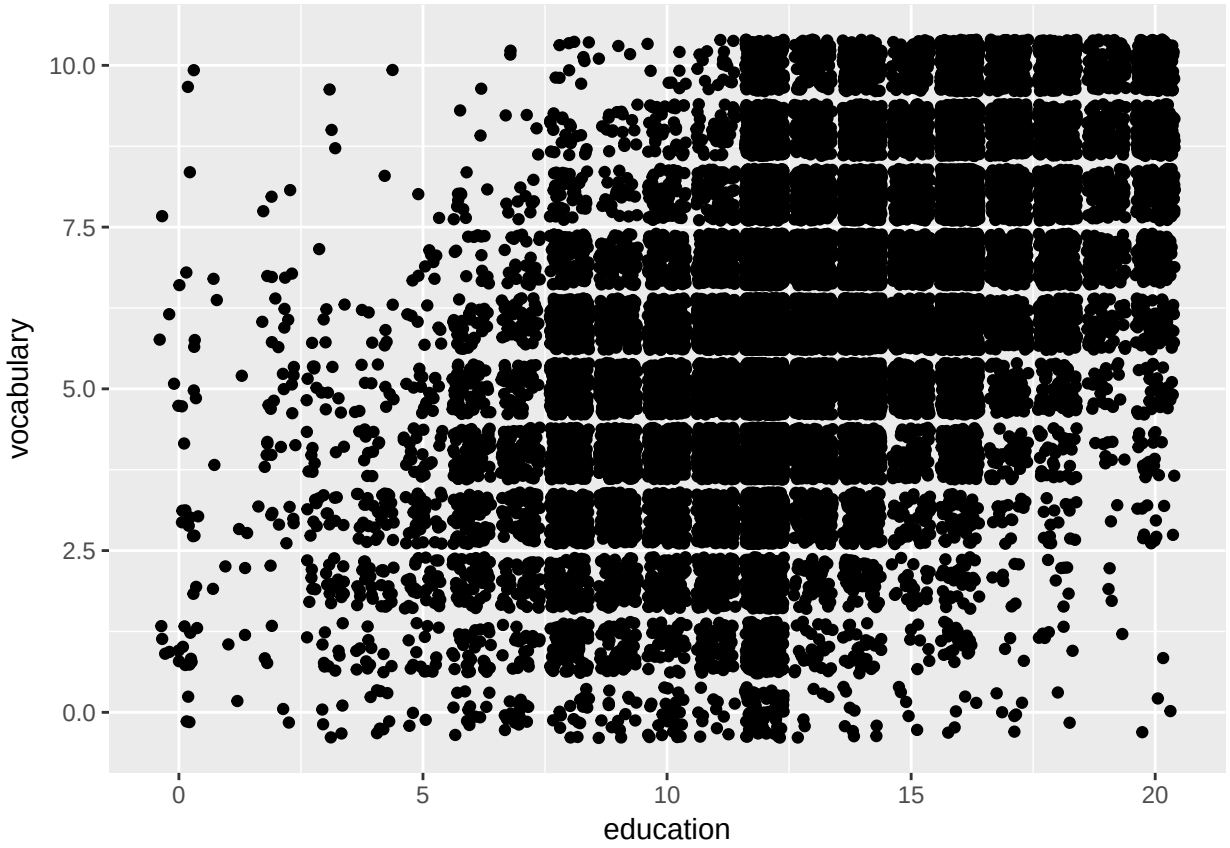
```
# Examine the structure of Vocab
str(Vocab)
```

```
## 'data.frame':    30351 obs. of  4 variables:
##  $ year      : num  1974 1974 1974 1974 1974 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 2 2 1 1 1 2 2 2 1 1 ...
##  $ education : num  14 16 10 10 12 16 17 10 12 11 ...
##  $ vocabulary: num  9 9 9 5 8 8 9 5 3 5 ...
##  - attr(*, "na.action")=Class 'omit'  Named int [1:32115] 1 2 3 4 5 6 7 8 9 10 ...
##   .. ..- attr(*, "names")= chr [1:32115] "19720001" "19720002" "19720003" "19720004" ...
```

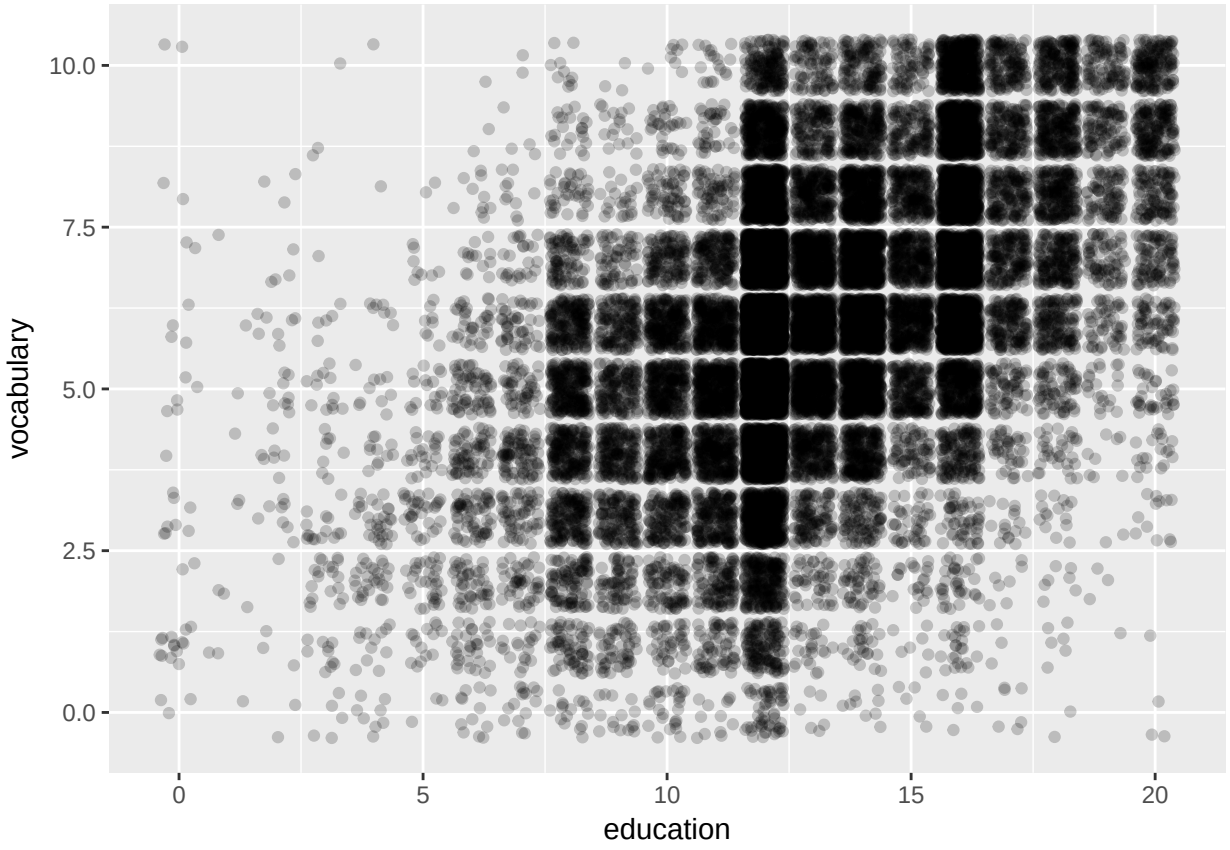
```
# Basic scatter plot of vocabulary (y) against education (x). Use geom_point()
ggplot(Vocab, aes(x = education, y = vocabulary)) +
  geom_point()
```



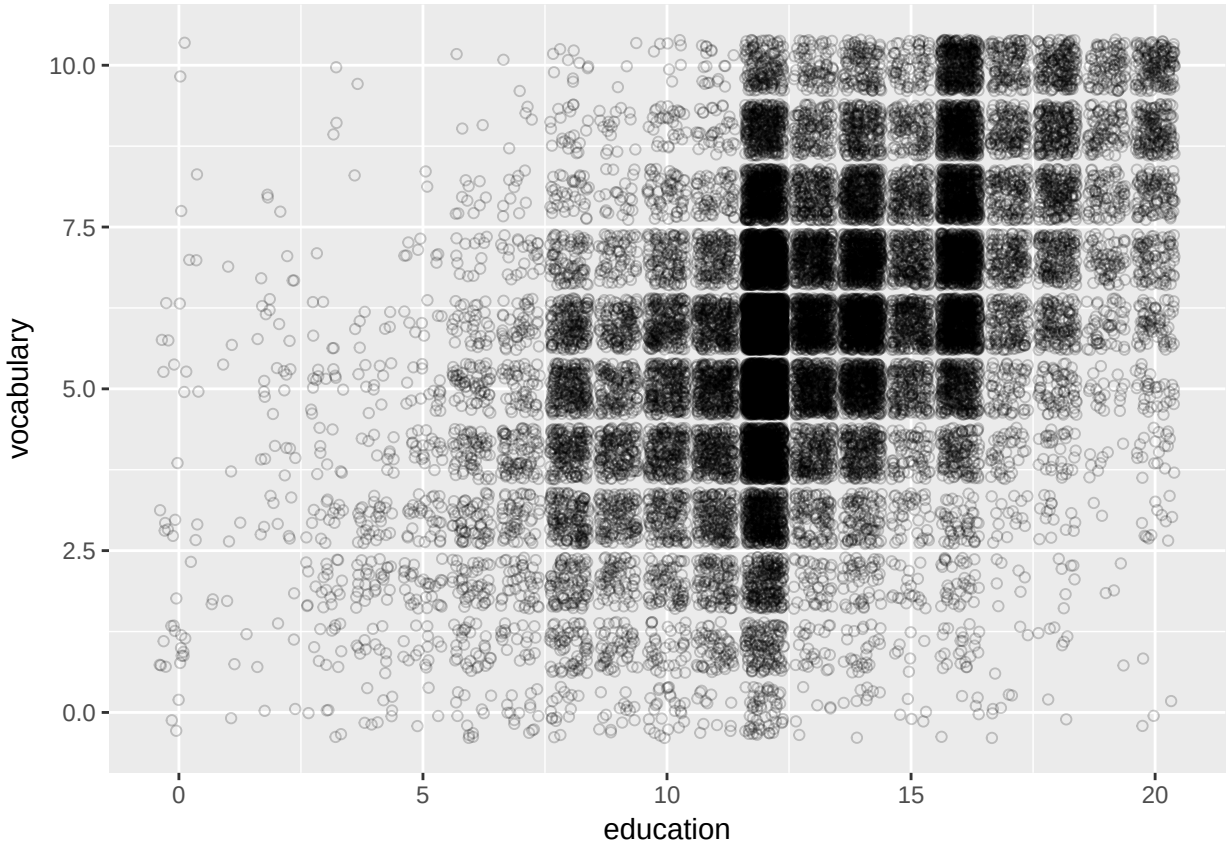

```
# Use geom_jitter() instead of geom_point()
ggplot(Vocab, aes(x = education, y = vocabulary)) +
  geom_jitter()
```




```
# Using the above plotting command, set alpha to a very low 0.2
ggplot(Vocab, aes(x = education, y = vocabulary) )+
  geom_jitter(alpha = 0.2)
```




```
# Using the above plotting command, set the shape to 1
ggplot(Vocab, aes(x = education, y = vocabulary)) +
  geom_jitter(alpha = 0.2, shape = 1)
```



Histograms

Histograms are one of the most common and intuitive ways of showing distributions. In this exercise you'll use the `mtcars` data frame to explore typical variations of simple histograms. But first, some background:

The x axis/aesthetic: The documentation for `geom_histogram()` states the argument `stat = "bin"` as a default. Recall that histograms cut up a continuous variable into discrete bins - thats what the stat "bin" is doing. You always get 30 evenly-sized bins by default, which is specified with the default argument `binwidth = range/30`. This is a pretty good starting point if you don't know anything about the variable being ploted and want to start exploring.

The y axis/aesthetic: `geom_histogram()` only requires one aesthetic: `x`. But there is clearly a `y` axis on your plot, so where does it come from? Actually, there is a variable mapped to the `y` aesthetic, it's called `..count...` When `geom_histogram()` executed the binning statistic (see above), it not only cut up the data into discrete bins, but it also counted how many values are in each bin. So there is an internal data frame where this information is stored. The `..` calls the variable count from this internal data frame. This is what appears on the y aesthetic. But it gets better! The density has also been calculated. This is the proportional frequency of this bin in relation to the whole data set. You use `..density..` to access this information.

INSTRUCTIONS

1 - Use the `mtcars` data frame and make a univariate histogram by mapping `mpg` onto the `x` aesthetic. Use `geom_histogram()` for the geom layer.

2 - Take plot 1 and manually create 1-unit wide bins with the `binwidth = 1` argument in `geom_histogram()`.

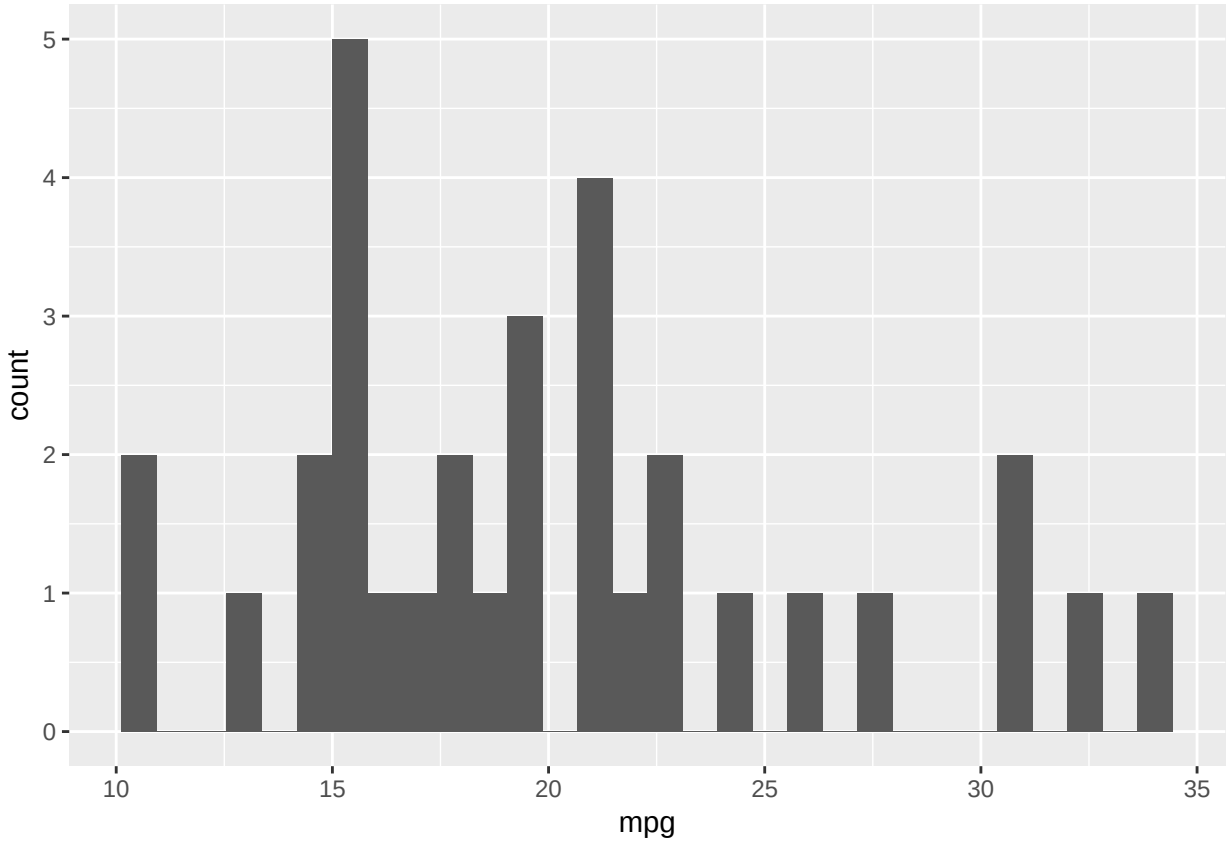
3 - Take plot 2, and map `..density..` onto the `y` aesthetic (i.e. inside an `aes()`) inside `geom_histogram()`. You'll have two `aes()` functions: one inside `ggplot()` and another inside `geom_histogram()`. (See the intro

text for a discussion of `..density..`).

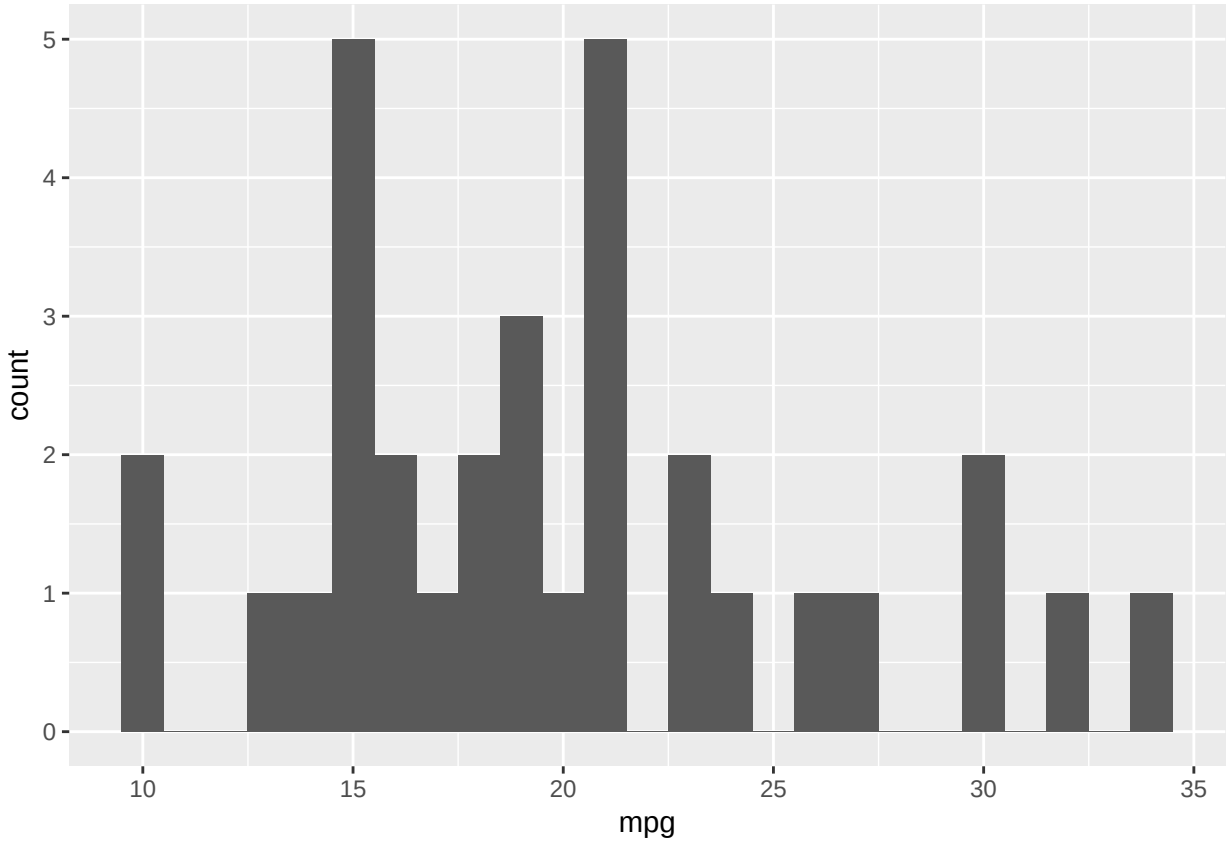
4 - Take plot 3 and set the attribute `fill`, the inside of the bars, to the value `"#377EB8"` in `geom_histogram()`. This should not appear in `aes()`, since it's an attribute, not an aesthetic mapping.

```
# 1 - Make a univariate histogram
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram()
```

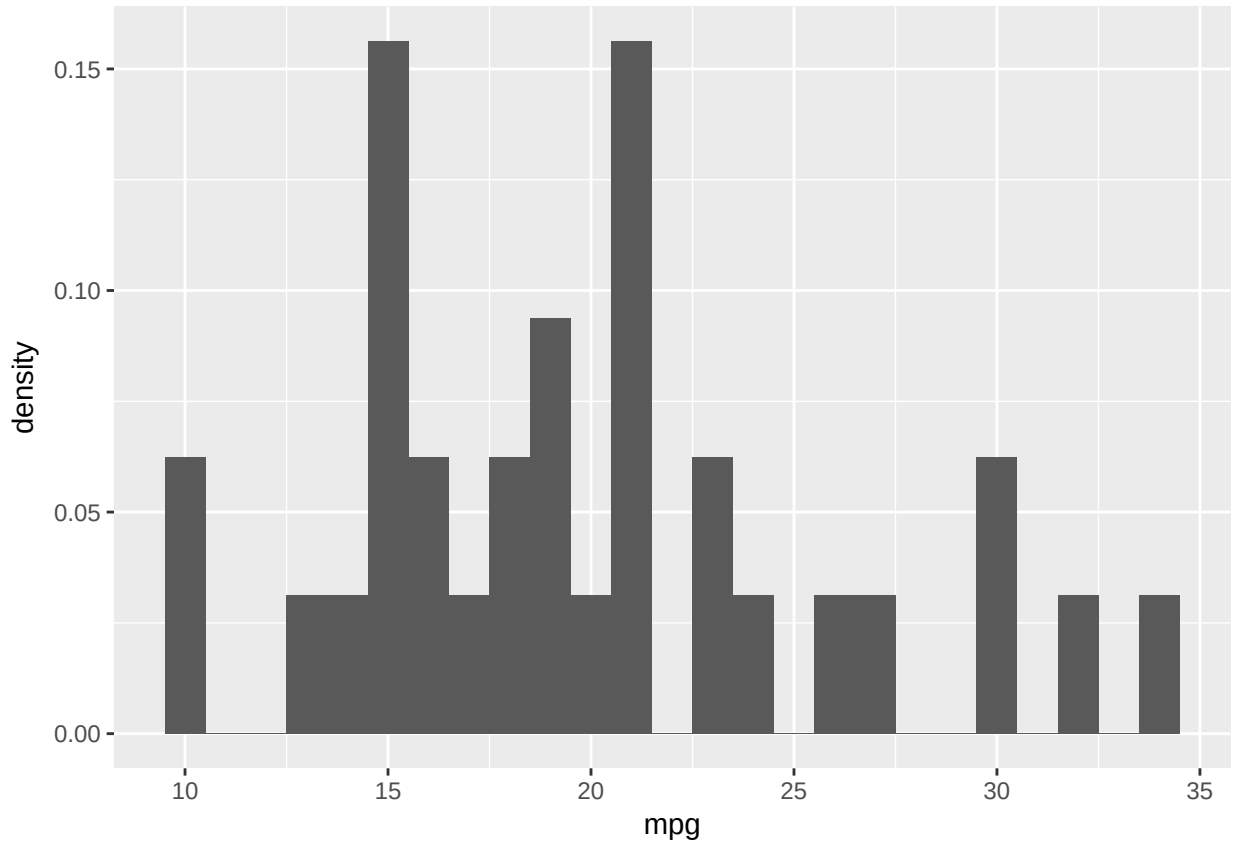
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



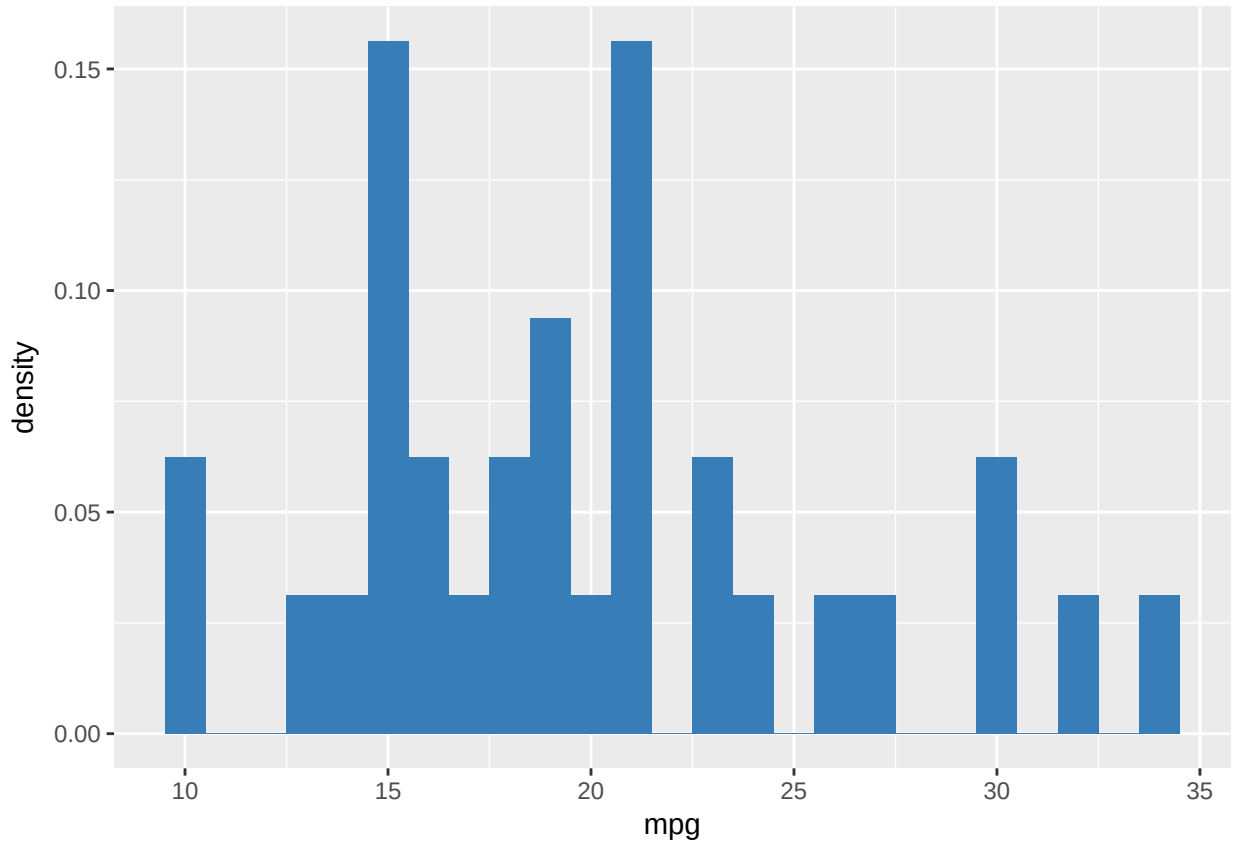

```
# 2 - Plot 1, plus set binwidth to 1 in the geom layer
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 1)
```




```
# 3 - Plot 2, plus MAP ..density.. to the y aesthetic (i.e. in a second aes() function)
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 1, aes(y = ..density..))
```




```
# 4 - plot 3, plus SET the fill attribute to "#377EB8"
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 1, fill = "#377EB8", aes(y = ..density..))
```



Position

In the previous chapter you saw that there are lots of ways to position scatter plots. Likewise, the `geom_bar()` and `geom_histogram()` geoms also have a `position` argument, which you can use to specify how to draw the bars of the plot.

Three `position` arguments will be introduced here:

- 1. `stack`: place the bars on top of each other. Counts are used. This is the default position.
- 2. `fill`: place the bars on top of each other, but this time use proportions.
- 3. `dodge`: place the bars next to each other. Counts are used.

In this exercise you'll draw the total count of cars having a given number of cylinders (`cyl`), according to manual or automatic transmission type (`am`) - as shown in the viewer.

Since, in the built-in `mtcars` data set, `cyl` and `am` are integers, they have already been converted to factor variables for you.

INSTRUCTIONS

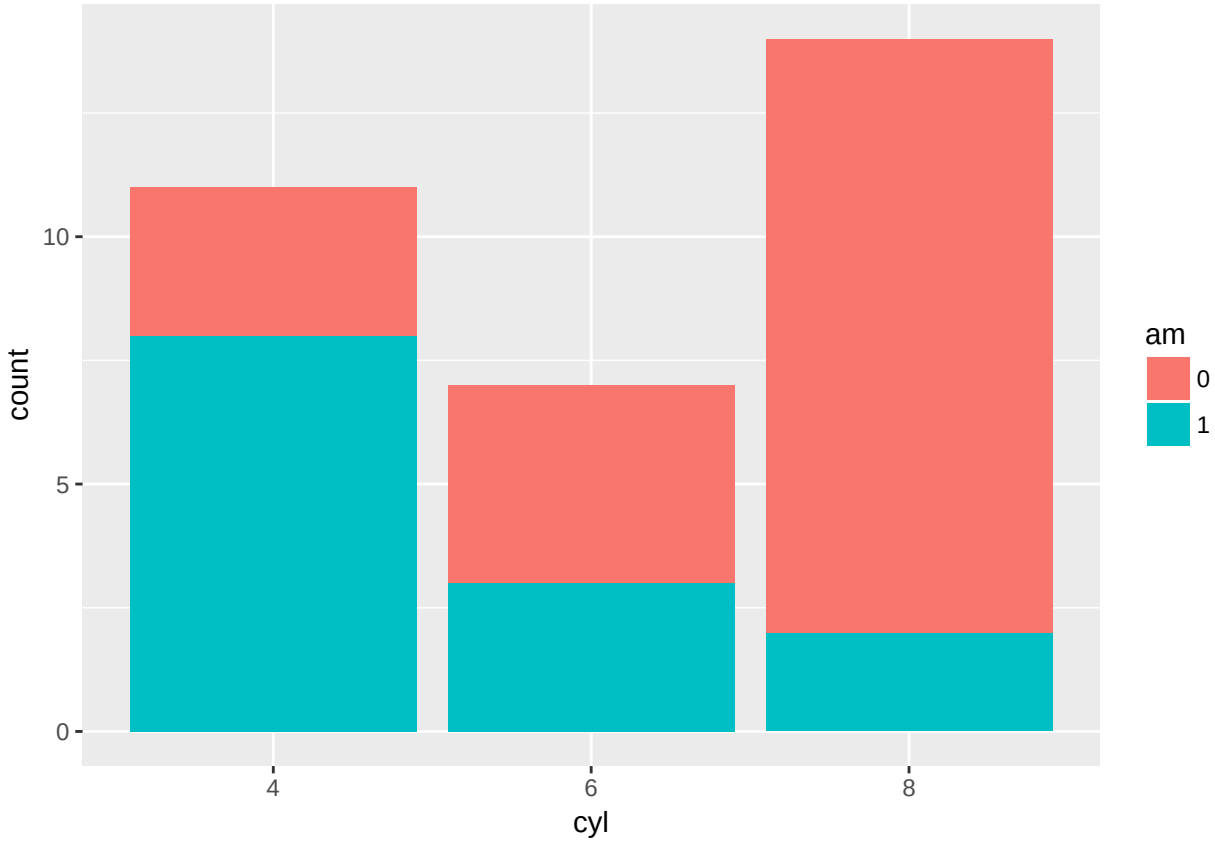
1 - Using `mtcars`, map `cyl` onto the `x` aesthetic and `am` onto `fill`. Use `geom_bar()` to make a bar plot.

2 - Take plot 1 and explicitly set `position = stack"` in `geom_bar()`. This doesn't change anything, does it? It was mentioned above that `"stack"` is the default.

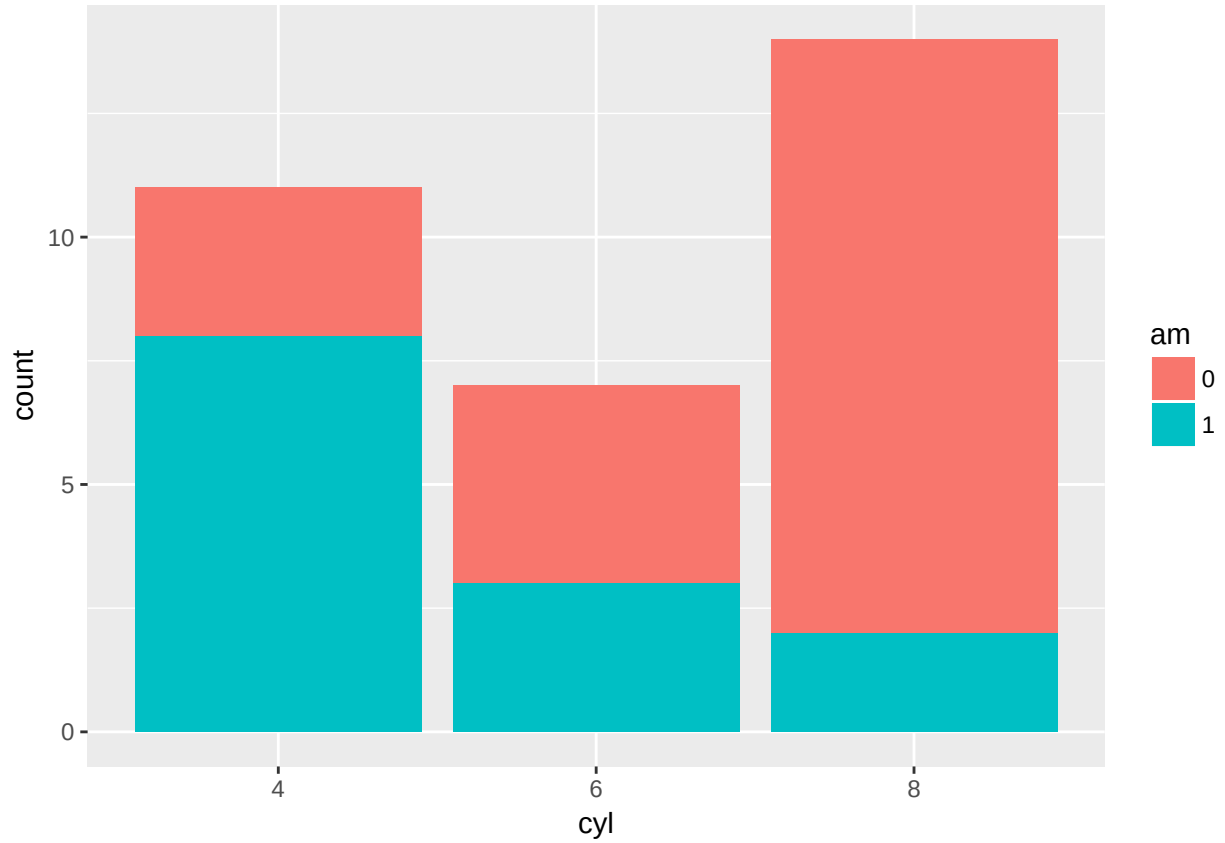
3 - Take plot 2 and set `position = "fill"` in `geom_bar()`.

4 - Take plot 3 and set `position = "dodge"` in `geom_bar()`.

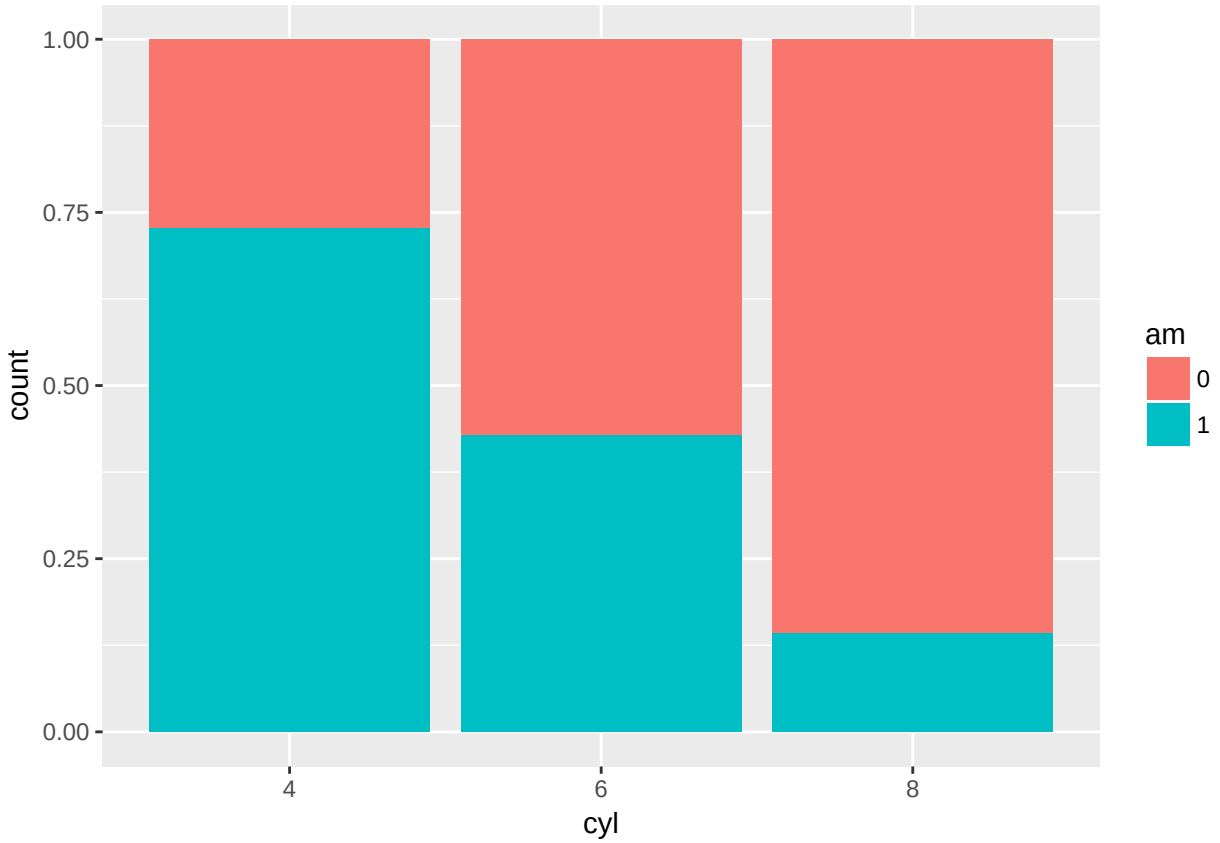
```r
# Draw a bar plot of cyl, filled according to am
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar()
```



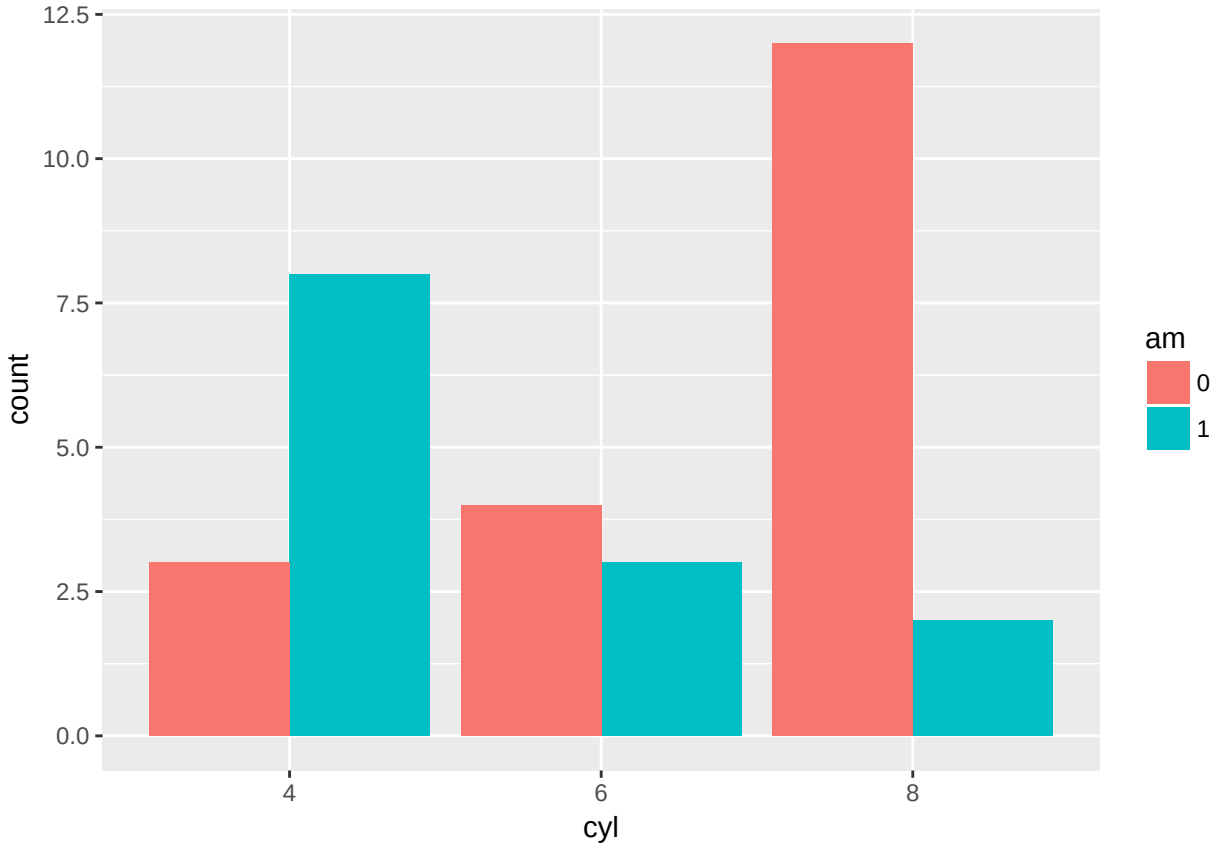

```r
# Change the position argument to stack
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar(position = "stack")
```




```
# Change the position argument to fill
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar(position = "fill")
```




```
# Change the position argument to dodge
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar(position = "dodge")
```



Overlapping bar plots

So far you've seen three different positions for bar plots: `stack` (the default), `dodge`(preferred), and `fill` (to show proportions).

However, you can go one step further by adjusting the dodging, so that your bars partially overlap each other. For this example you'll again use the `mtcars` dataset. Like last time `cyl` and `am` are already available as factors inside `mtcars`.

Instead of using `position = "dodge"` you're going to use `position_dodge()`, like you did with `position_jitter()` in the **Scatter plots and jittering (1)** exercise. Here, you'll save this as an object, `posn_d`, so that you can easily reuse it.

Remember, the reason you want to use `position_dodge()` (and `position_jitter()`) is to specify how much dodging (or jittering) you want.

INSTRUCTIONS

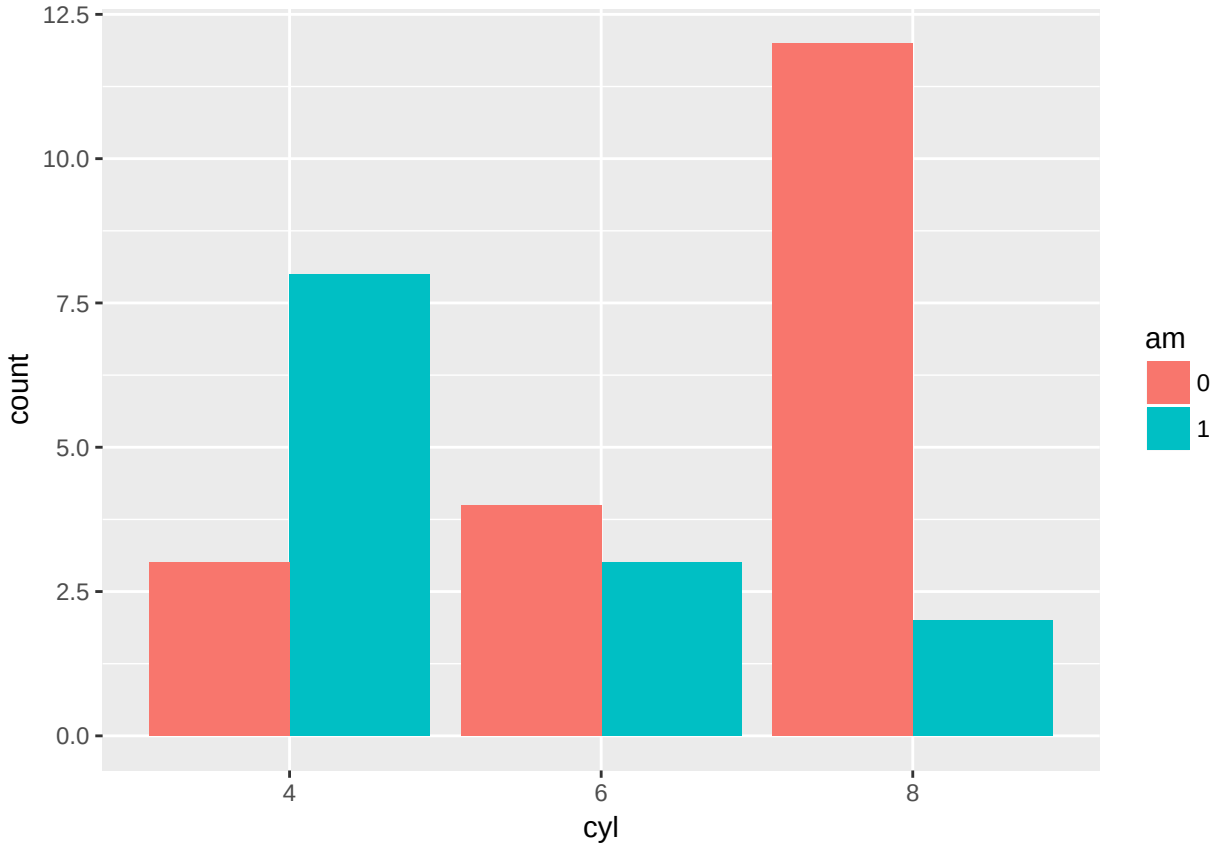
1 - The last plot from the last exercise has been provided for you.

2 - Define a new object called `posn_d` by calling `position_dodge()` with the argument `width = 0.2`.

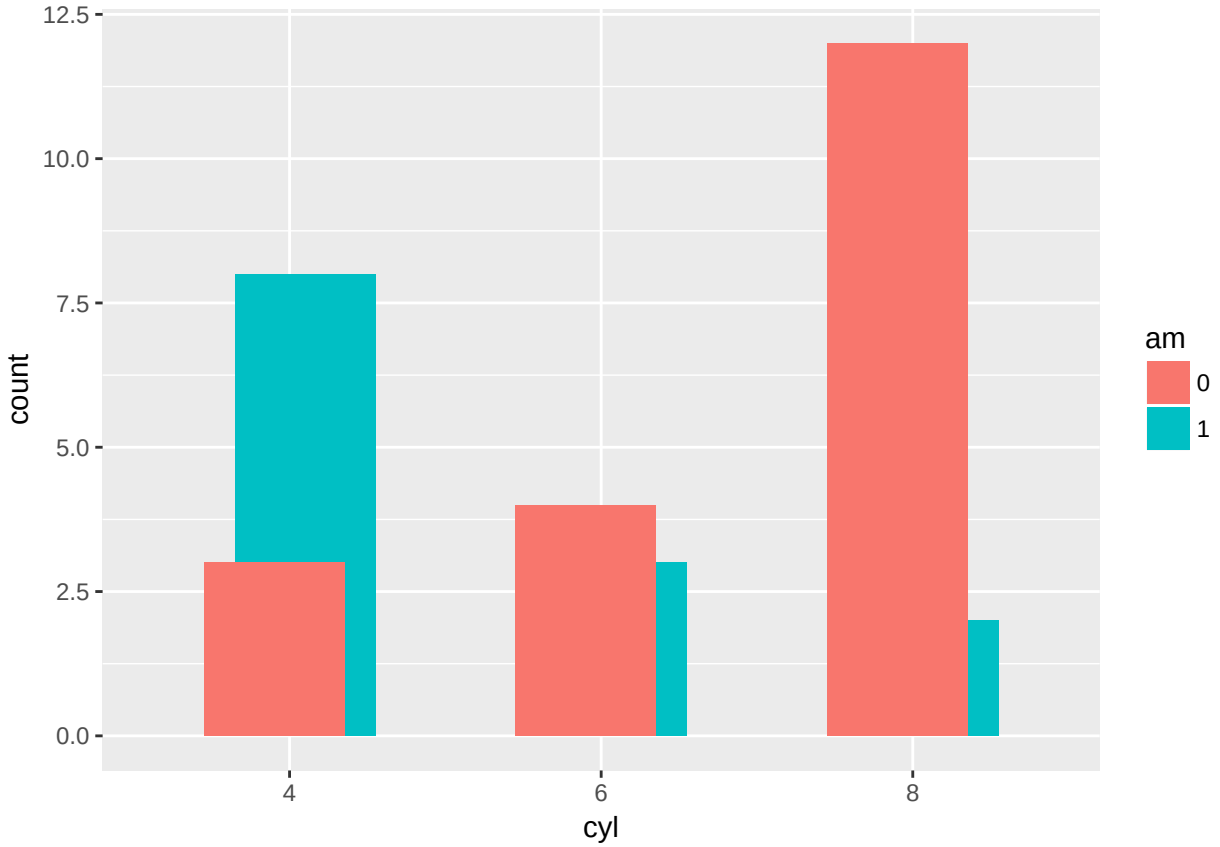
3 - Take plot 1 and make slightly overlapping bars by using the `position = posn_d` argument.

4 - Take plot 3 and `set alpha = 0.6` to see the overlap in bars.

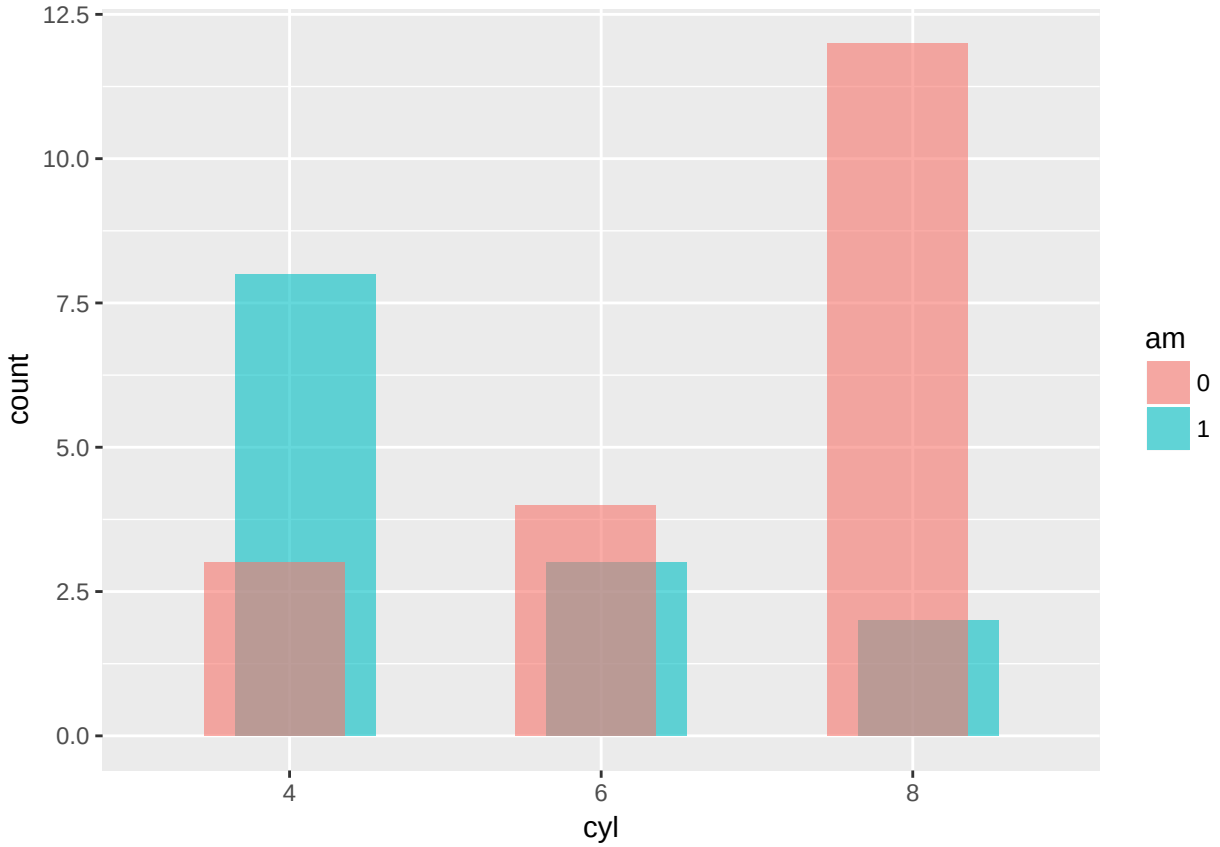
```
# 1 - The last plot form the previous exercise
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar(position = "dodge")
```




```
# 2 - Define posn_d with position_dodge()
posn_d <- position_dodge(width = 0.2)

# 3 - Change the position argument to posn_d
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar(position = posn_d)
```




```
# 4 - Use posn_d as position and adjust alpha to 0.6
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar(position = posn_d, alpha = 0.6)
```



Icon exercise interactive

Overlapping histograms pose similar problems to overlapping bar plots, but there is a unique solution here: a frequency polygon.

This is a geom specific to binned data that draws a line connecting the value of each bin. Like `geom_histogram()`, it takes a binwidth argument and by default `stat = "bin"` and `position = "identity"`.

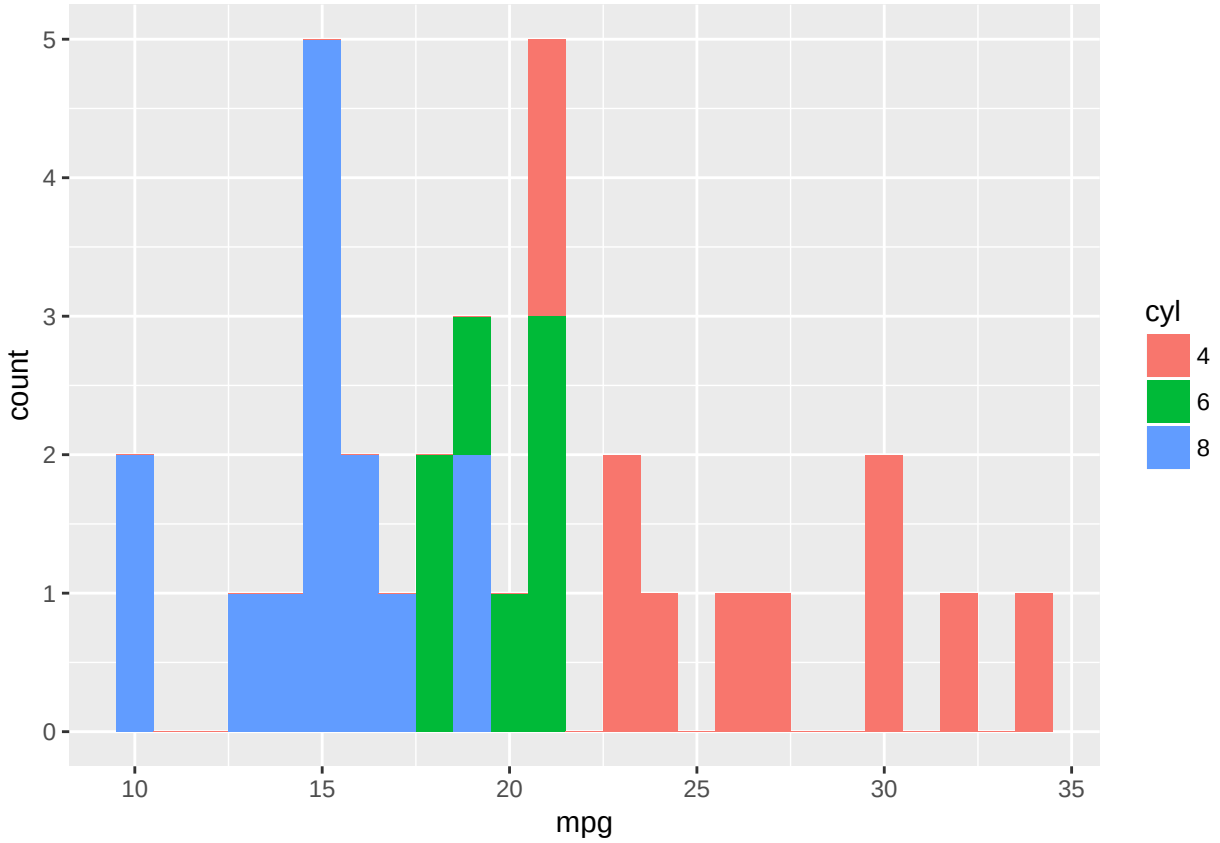
INSTRUCTIONS

The code for a basic histogram of `mpg`, which you've already seen, is provided. Extend the code to map `cyl` onto `fill` inside `aes()`.

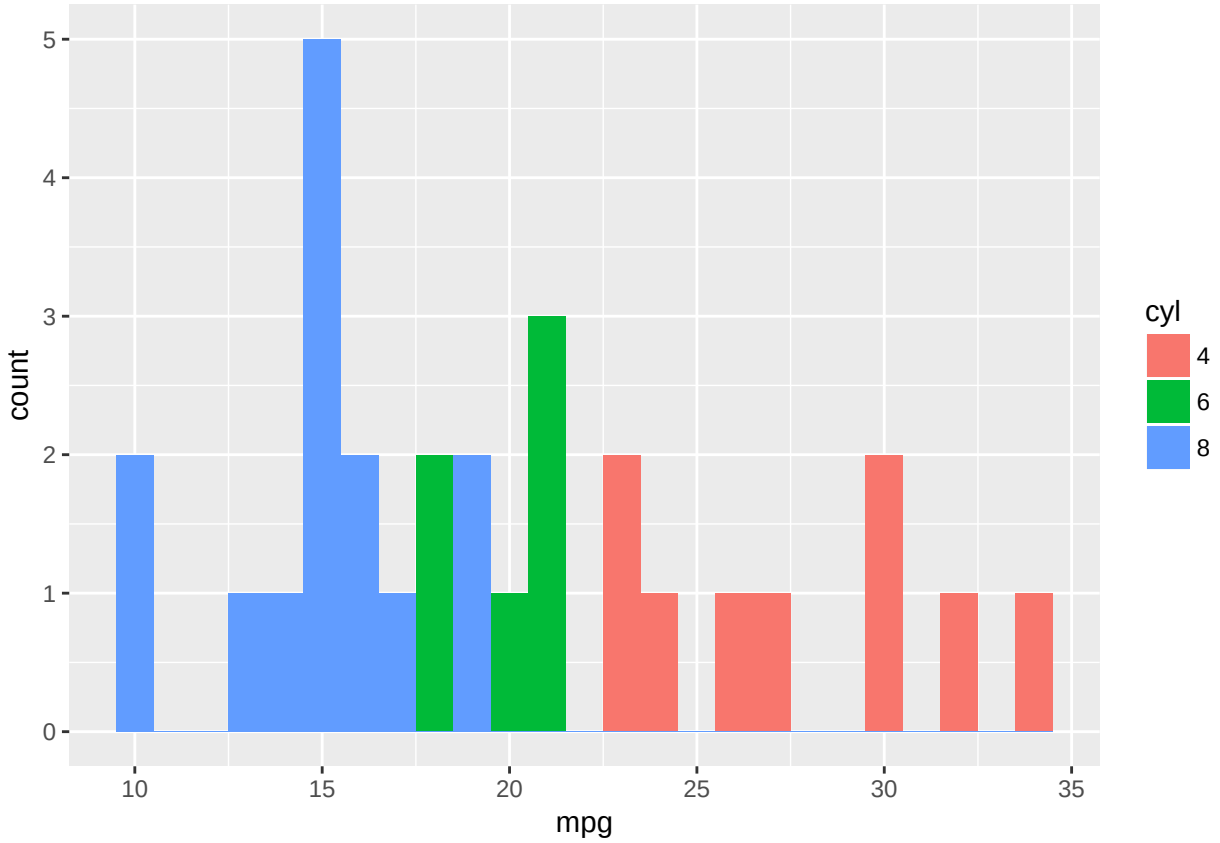
The default position for histograms is `"stack"`. Copy your solution to the first exercise and set the position for the histogram bars to `"identity"`.

Using the same data and base layers as in the previous two plots, create a plot with a `geom_freqpoly()`. Because you're no longer working with bars, change the `aes()` function: `cyl` should be mapped onto `col`, not onto `fill`. This will correctly color the geom.

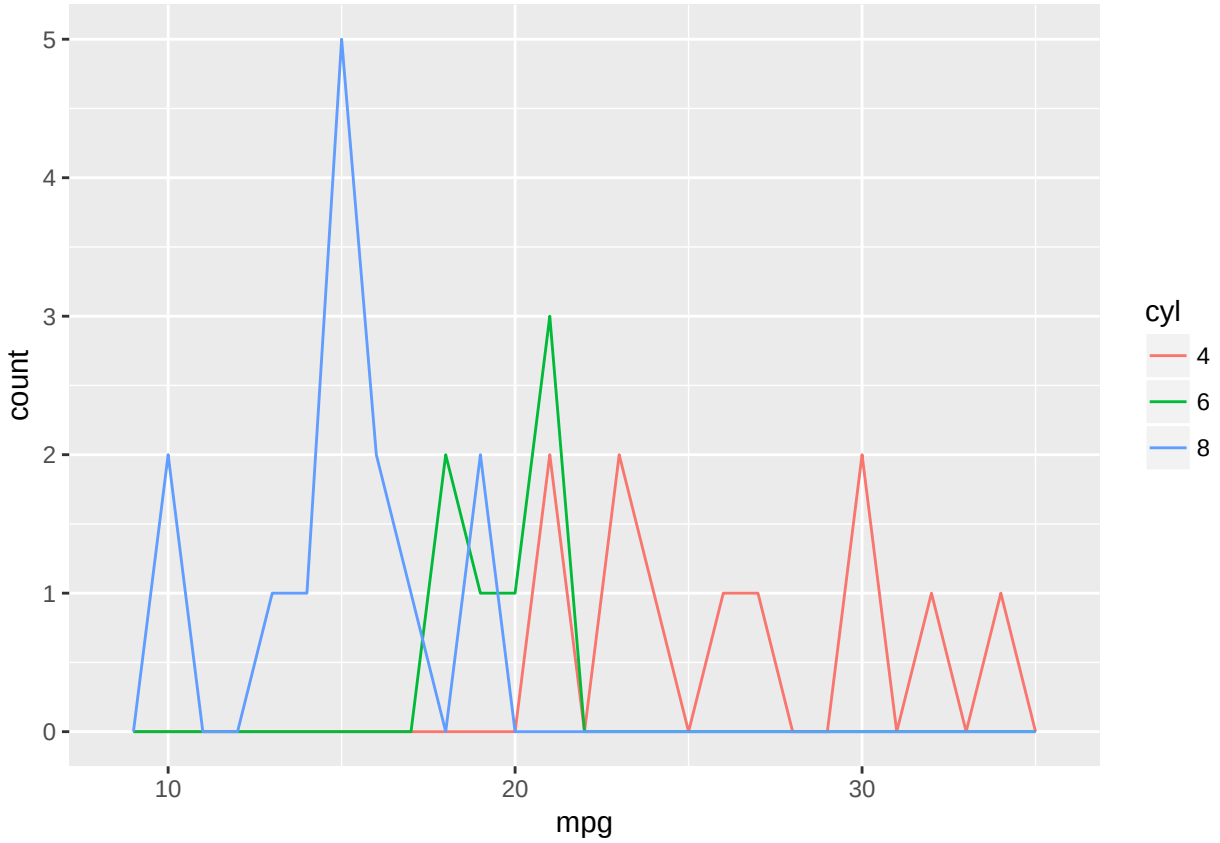
```
# A basic histogram, add coloring defined by cyl
ggplot(mtcars, aes(mpg, fill = cyl)) +
  geom_histogram(binwidth = 1)
```




```
# Change position to identity
ggplot(mtcars, aes(mpg, fill = cyl)) +
  geom_histogram(position = "identity", binwidth = 1)
```




```
# Change geom to freqpoly (position is identity by default)
ggplot(mtcars, aes(mpg, col = cyl)) +
  geom_freqpoly(position = "identity", binwidth = 1)
```



Overlapping histograms

In this example of a bar plot, you'll fill each segment according to an ordinal variable. The best way to do that is with a sequential color series.

You'll be using the `Vocab` dataset from earlier. Since this is a much larger dataset with more categories, you'll also compare it to a simpler dataset, `mtcars`. Both datasets are ordinal.

INSTRUCTIONS

The bar plot from the previous exercise is provided - `cyl` is on the x-axis and filled according to transmission type, `am`. Notice how you can set the color palette used to fill the bars with `scale_fill_brewer()`. For a full list of possible color sets, have a look at `?brewer.pal`.

Explore `Vocab` with `str()`. Notice that the `education` and `vocabulary` variables have already been converted to factor variables for you.

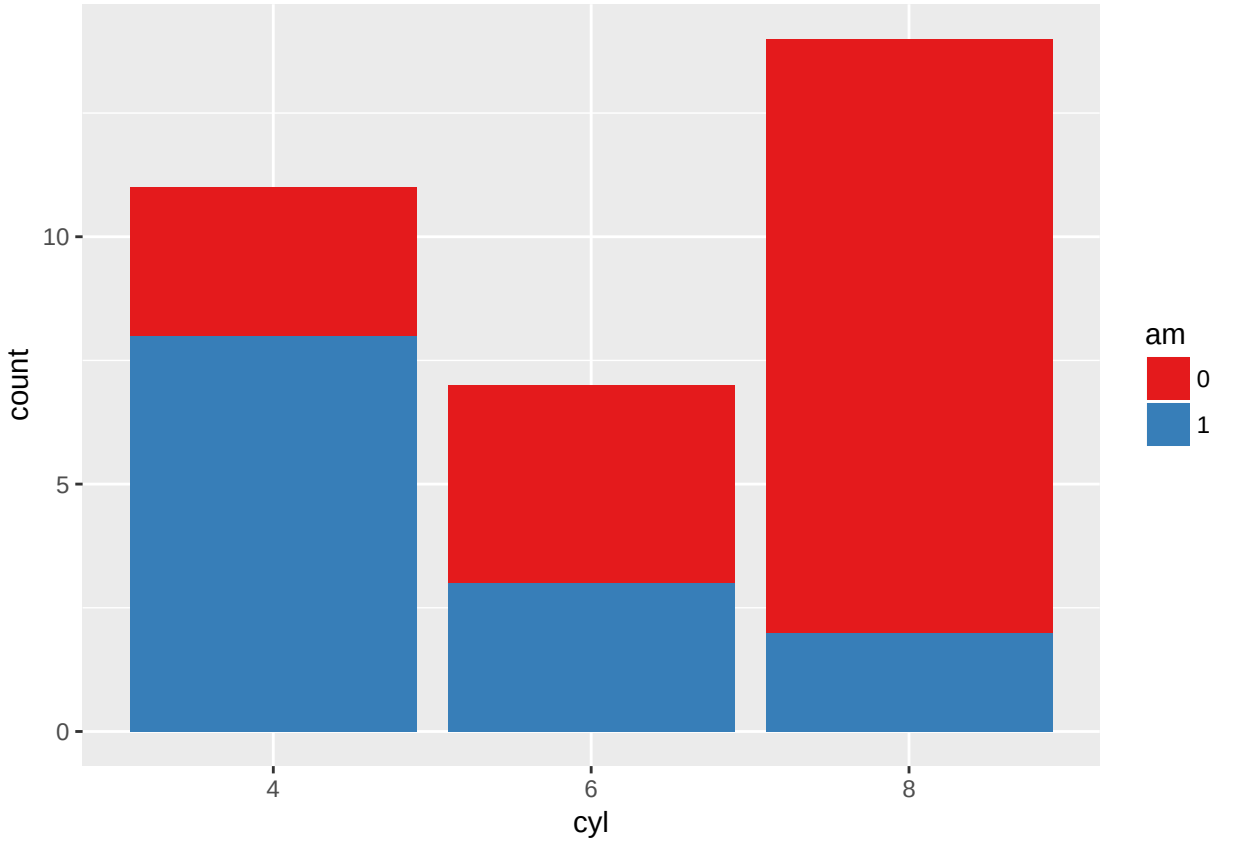
Make a filled bar chart with the Vocab dataset.

- Map `education` to `x` and `vocabulary` to `fill`.
- Inside `geom_bar()`, make sure to set position = "fill"‘.

Allow color brewer to choose a default color palette by using the appropriate scale function, without arguments. Notice how this generates a warning message and an incomplete plot.

```
Vocab <- Vocab %>%
  mutate(education = factor(education),
         vocabulary = factor(vocabulary))
```

```
# Example of how to use a brewed color palette
ggplot(mtcars, aes(x = cyl, fill = am)) +
  geom_bar() +
  scale_fill_brewer(palette = "Set1")
```

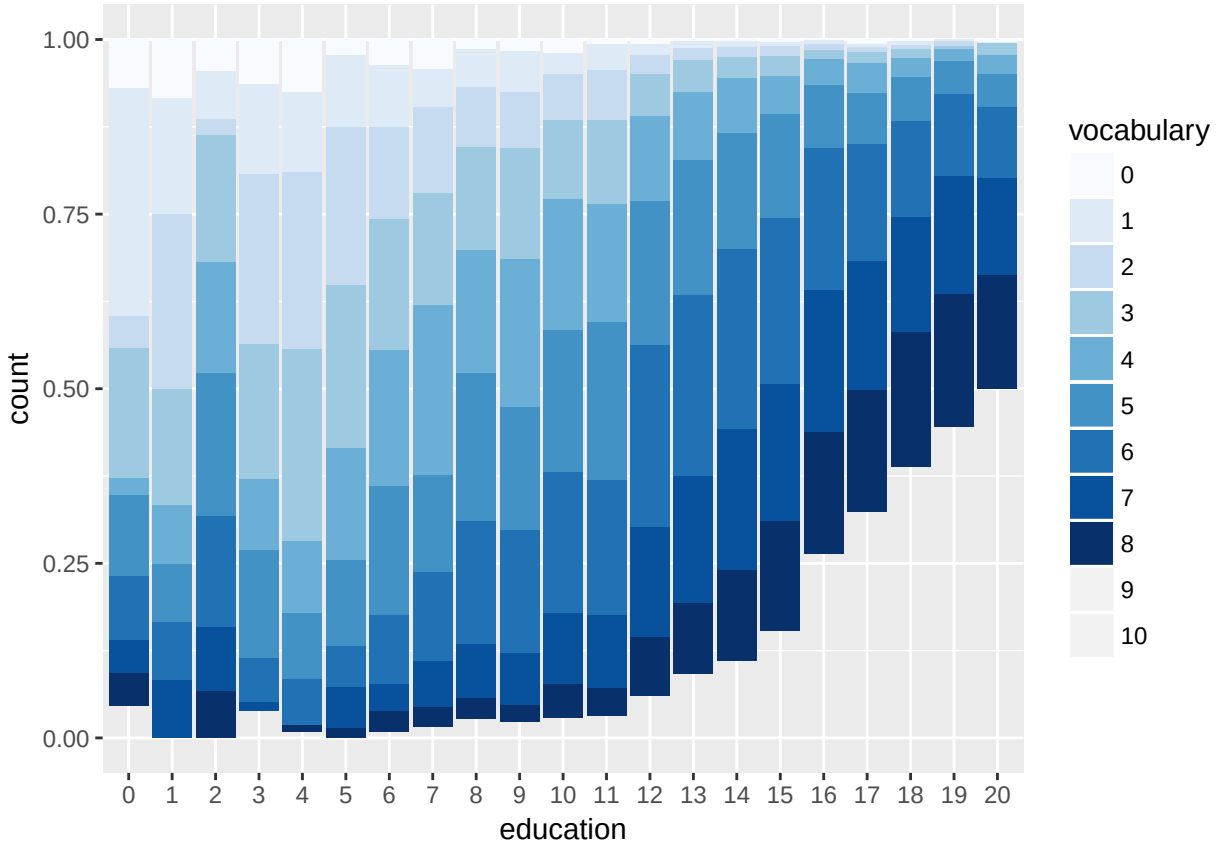



```
# Use str() on Vocab to check out the structure
str(Vocab)
```

```
## 'data.frame':    30351 obs. of  4 variables:
##  $ year      : num  1974 1974 1974 1974 1974 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 2 2 1 1 1 2 2 2 1 1 ...
##  $ education : Factor w/ 21 levels "0","1","2","3",..: 15 17 11 11 13 17 18 11 13 12 ...
##  $ vocabulary: Factor w/ 11 levels "0","1","2","3",..: 10 10 10 6 9 9 10 6 4 6 ...
```

```
# Plot education on x and vocabulary on fill
# Use the default brewed color palette
ggplot(Vocab, aes(x = education, fill = vocabulary)) +
  geom_bar(position = "fill") +
  scale_fill_brewer()
```

```
## Warning in RColorBrewer::brewer.pal(n, pal): n too large, allowed maximum for palette Blues is 9
## Returning the palette you asked for with that many colors
```



Bar plots with color ramp, part 1

In the previous exercise, you ended up with an incomplete bar plot. This was because for continuous data, the default `RColorBrewer` palette that `scale_fill_brewer()` calls is `"Blues"`. There are only 9 colours in the palette, and since you have 11 categories, your plot looked strange.

In this exercise, you'll manually create a color palette that can generate all the colours you need. To do this you'll use a function called `colorRampPalette()`.

The input is a character vector of 2 or more colour values, e.g. `"#FFFFFF"` (white) and `"#0000FF"` (pure blue). (See **All about attributes, part 1** exercise for a discussion on hexadecimal codes).

The output is itself a function! So when you assign it to an object, that object should be used as a function. To see what we mean, execute the following three lines in the console:

```
new_col <- colorRampPalette(c("#FFFFFF", "#0000FF"))
new_col(4) # the newly extrapolated colours
munsell::plot_hex(new_col(4)) # Quick and dirty plot
```

`new_col()` is a function that takes one argument: the number of colours you want to extrapolate. You want to use nicer colours, so we've assigned the entire `"Blues"` colour palette from the `RColorBrewer` package to the character vector `blues`.

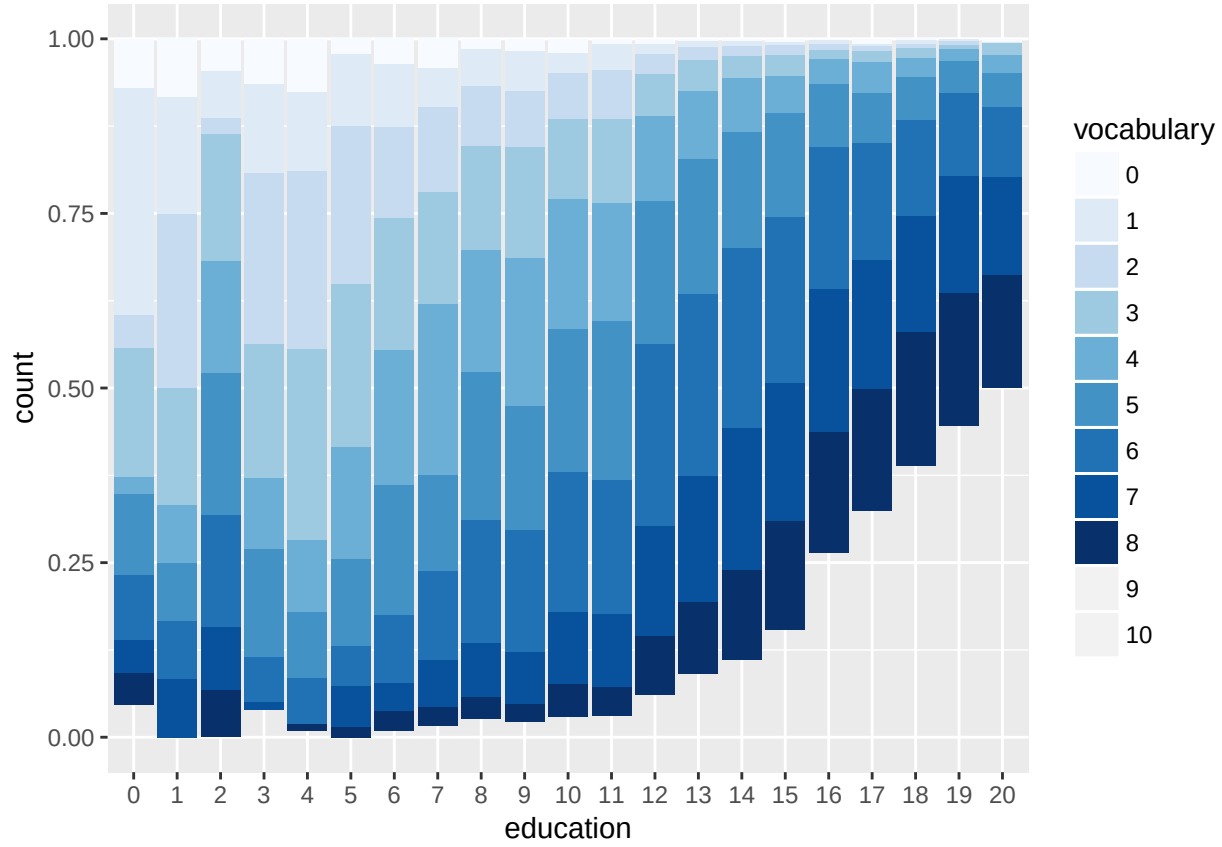
INSTRUCTIONS

1 - Like in the example code above, create a new function called `blue_range` that uses `colorRampPalette()` to extrapolate over all 9 values of the `blues` character vector.

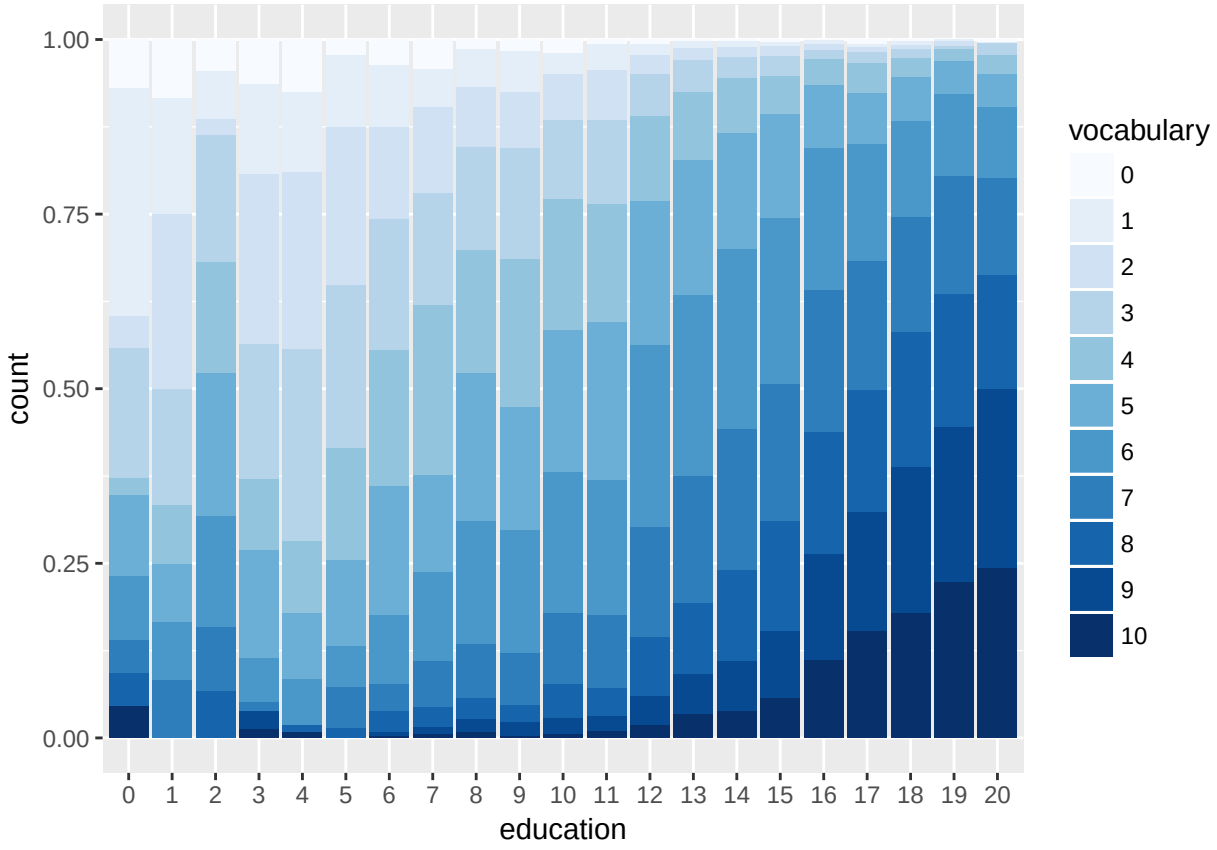
2 - Take the plot code from the last exercise (provided), and change `scale_fill_brewer()` to be `scale_fill_manual()`. Set the argument `values = blue_range(11)` inside `scale_fill_manual()`.

```
# Final plot of last exercise
ggplot(Vocab, aes(x = education, fill = vocabulary)) +
  geom_bar(position = "fill") +
  scale_fill_brewer()
```

```
## Warning in RColorBrewer::brewer.pal(n, pal): n too large, allowed maximum for palette Blues is 9
## Returning the palette you asked for with that many colors
```




```
# Definition of a set of blue colors
blues <- brewer.pal(9, "Blues") # from the RColorBrewer package

# 1 - Make a color range using colorRampPalette() and the set of blues
blue_range <- colorRampPalette(blues)

# 2 - Use blue_range to adjust the color of the bars, use scale_fill_manual()
ggplot(Vocab, aes(x = education, fill = vocabulary)) +
  geom_bar(position = "fill") +
  scale_fill_manual(values = blue_range(11))
```



Bar plots with color ramp, part 2

As a last example of bar plots, you'll return to histograms (which you now see are just a special type of bar plot). You saw a nice trick in a previous exercise of how to slightly overlap bars, but now you'll see how to overlap them completely. This would be nice for multiple histograms, as long as there are not too many different overlaps!

You'll make a histogram using the `mpg` variable in the `mtcars` data frame.

INSTRUCTIONS

1 - A basic histogram plot is provided.

2 - Take plot 1 and map `am` onto `fill` within the `aes()` function. The default position is `"stack"`.

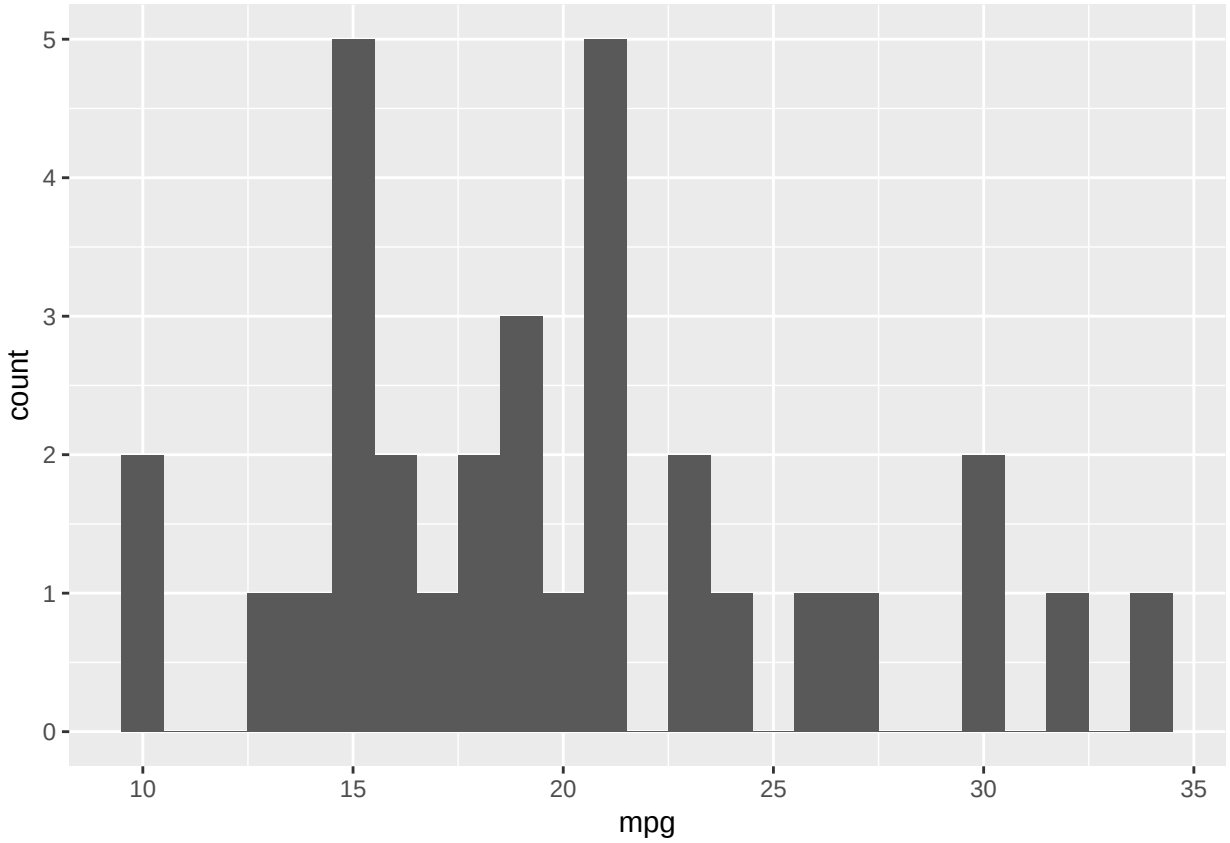
3 - Take plot 2 and add the `position` argument within `geom_histogram()`. Set it to `"dodge"`.

4 - Take plot 3 and change the `position` argument to `"fill"`. In this case, none of these positions really work well, because it's difficult to compare the distributions directly.

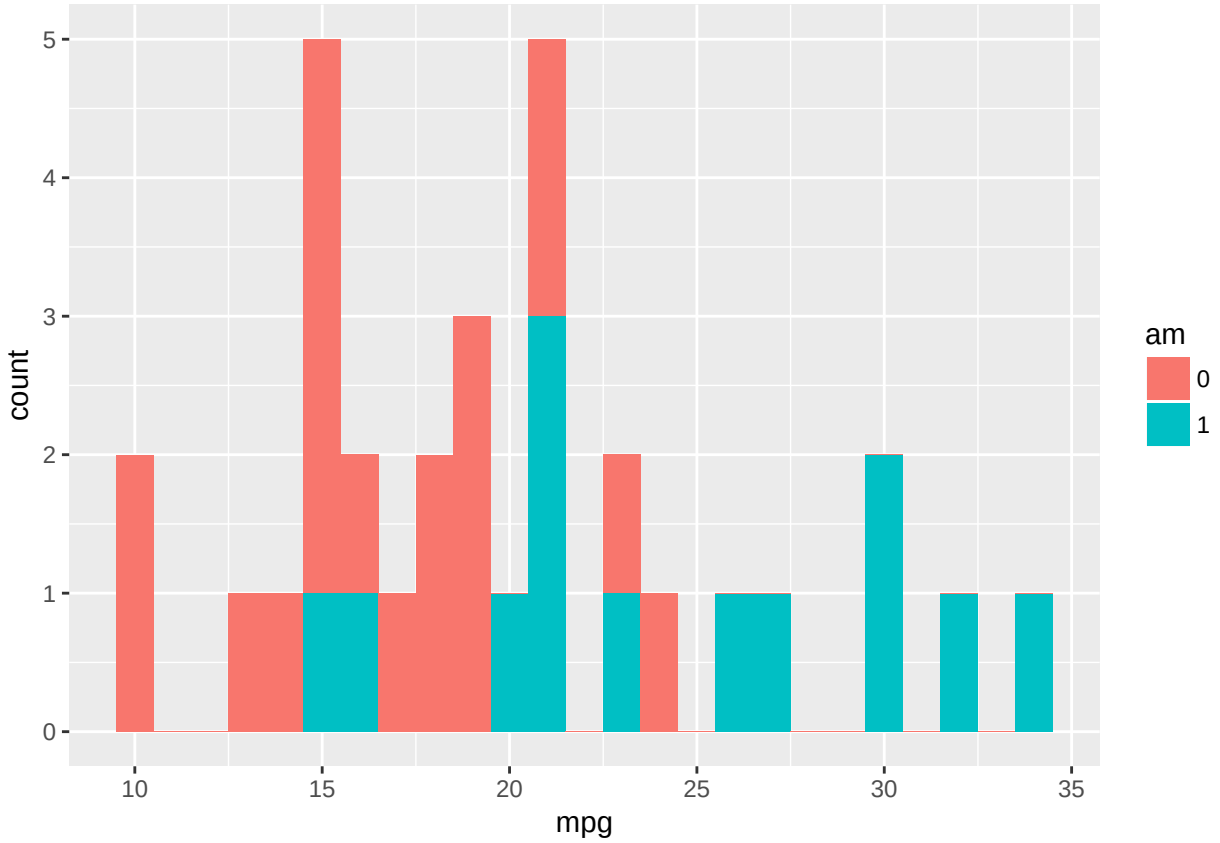
5 - Take plot 4 and change the position argument to `"identity"` and set `alpha = 0.4`. This produces overlapping bars.

6 - Take plot 5 and change the aesthetic mapping. Map `cyl` onto `fill`.

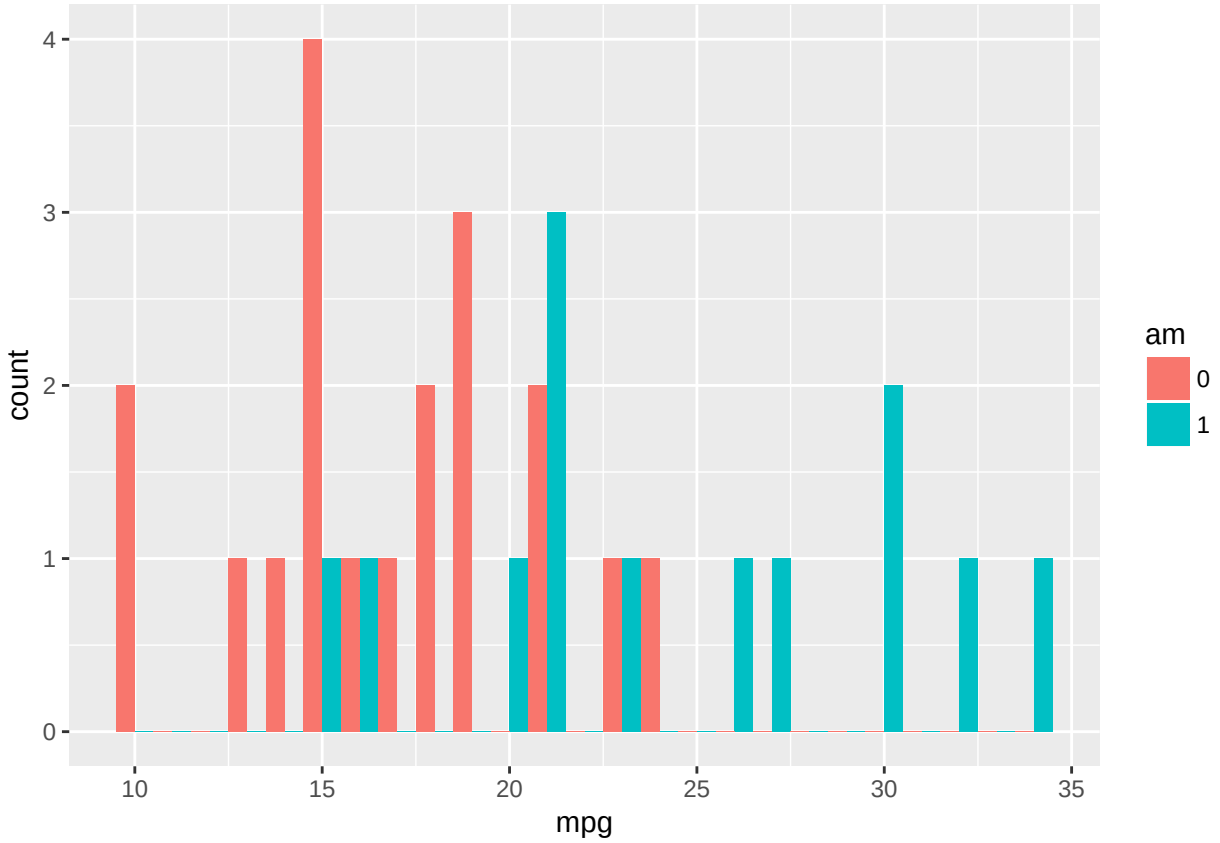
```
# 1 - Basic histogram plot command
ggplot(mtcars, aes(mpg)) +
  geom_histogram(binwidth = 1)
```




```
# 2 - Plot 1, Expand aesthetics: am onto fill
ggplot(mtcars, aes(mpg, fill = am)) +
  geom_histogram(binwidth = 1)
```

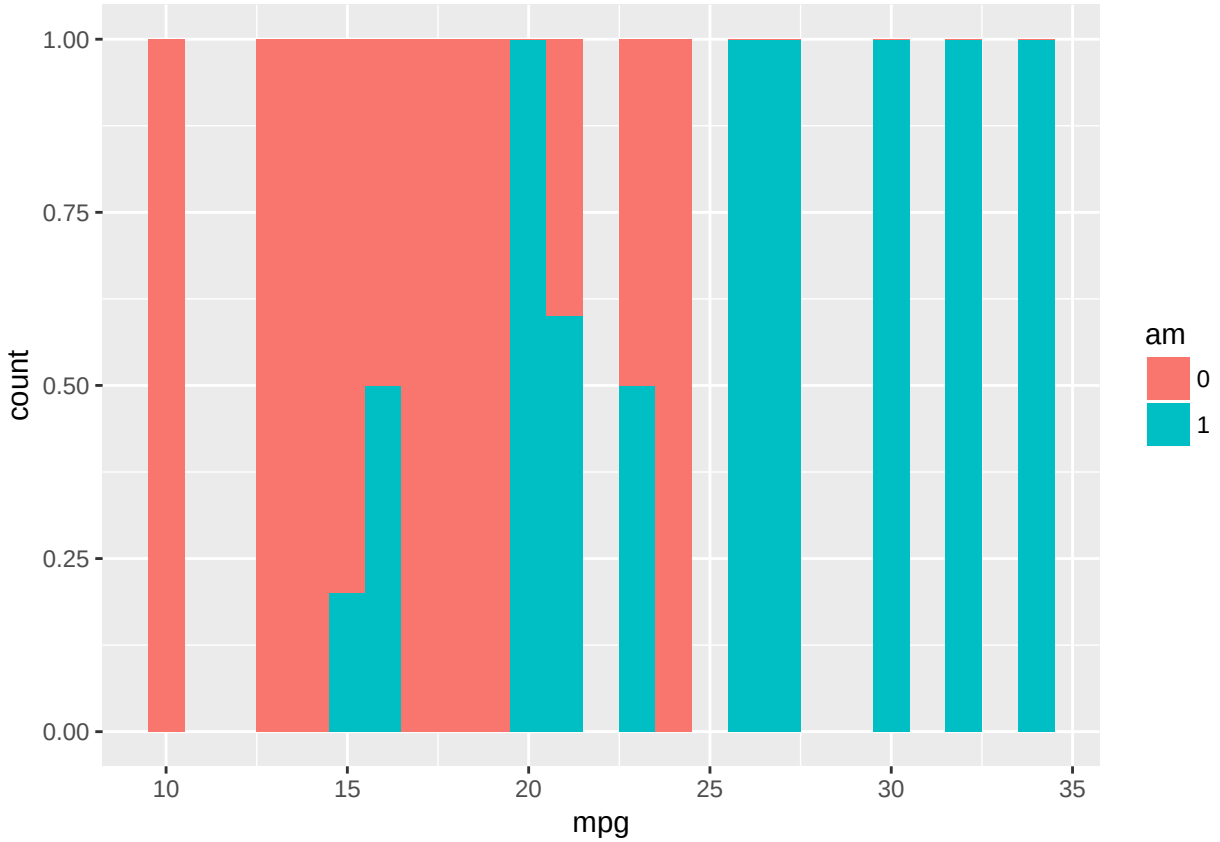



```
# 3 - Plot 2, change position = "dodge"
ggplot(mtcars, aes(mpg, fill = am)) +
  geom_histogram(binwidth = 1, position = "dodge")
```

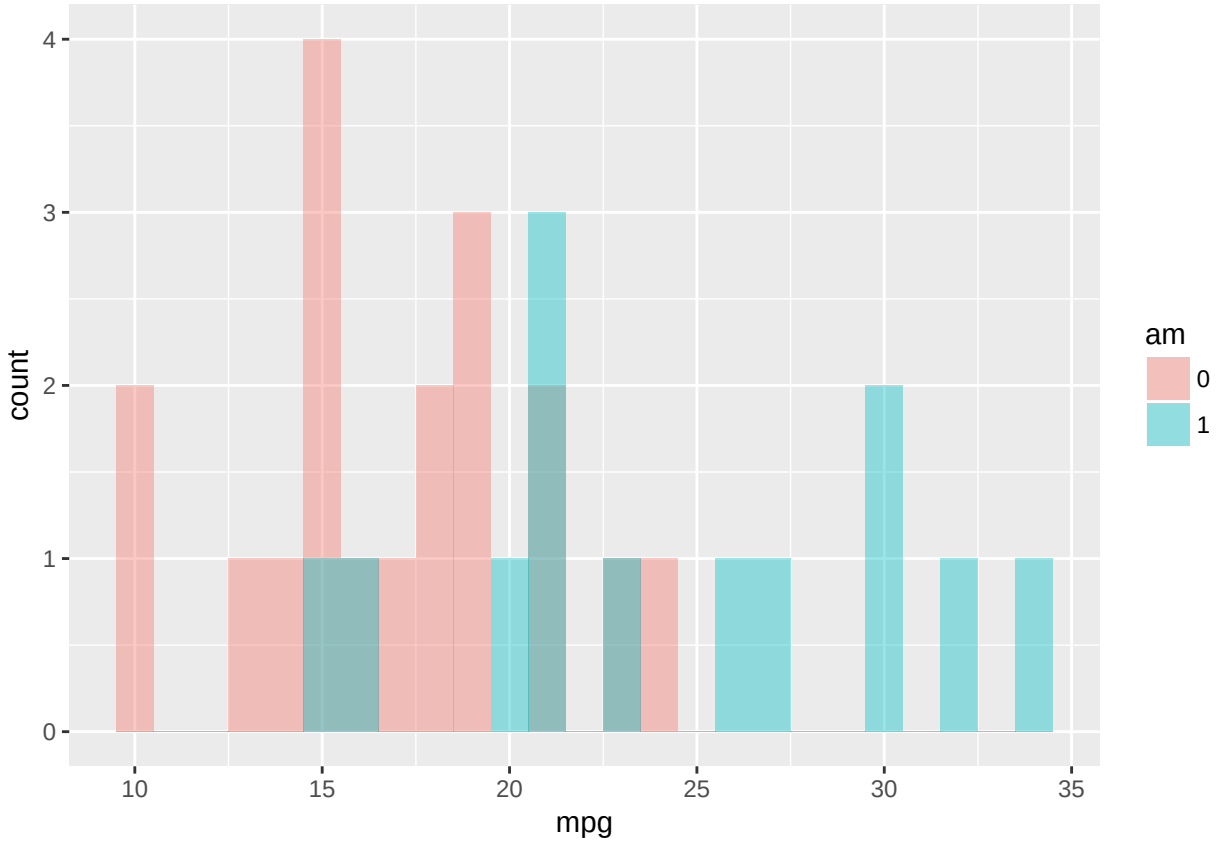



```
# 4 - Plot 3, change position = "fill"
ggplot(mtcars, aes(mpg, fill = am)) +
  geom_histogram(binwidth = 1, position = "fill")
```

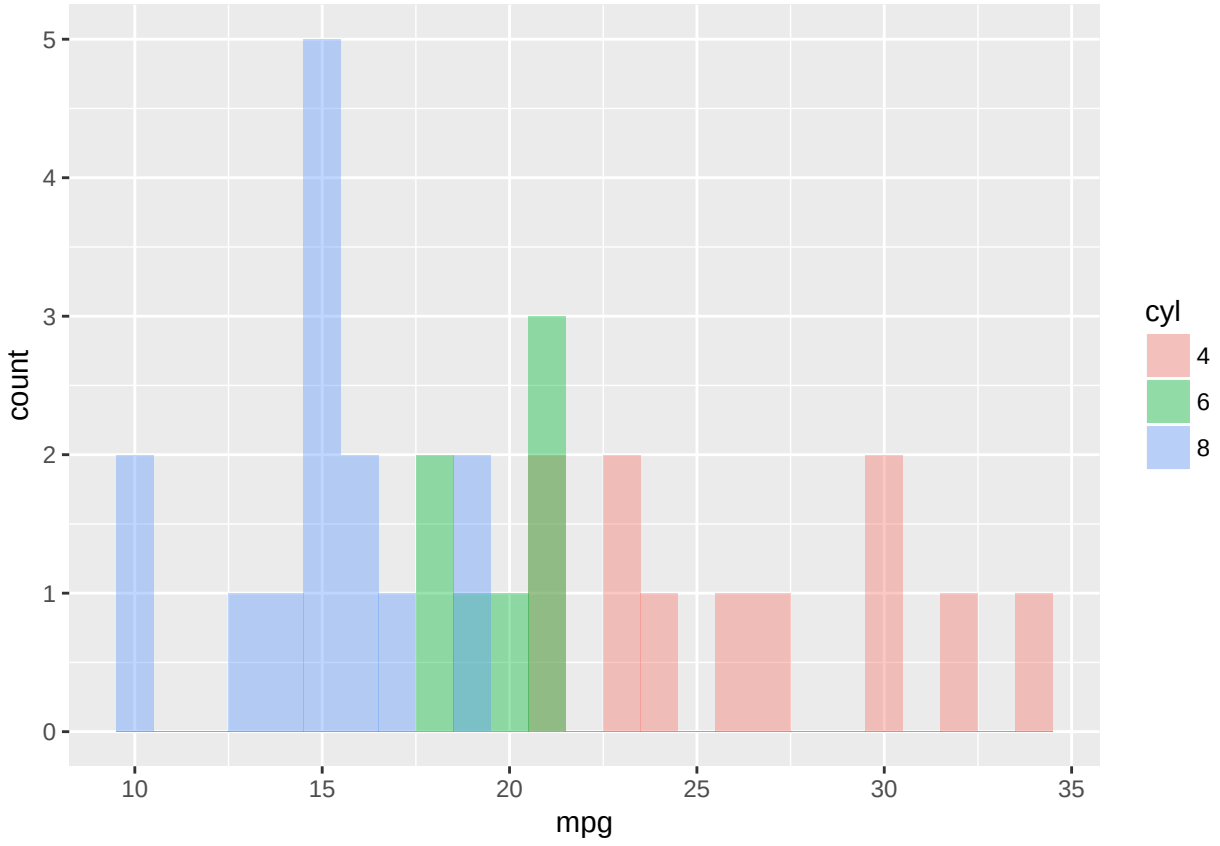
```
## Warning: Removed 16 rows containing missing values (geom_bar).
```




```
# 5 - Plot 4, plus change position = "identity" and alpha = 0.4
ggplot(mtcars, aes(mpg, fill = am)) +
  geom_histogram(binwidth = 1, position = "identity", alpha = 0.4)
```




```
# 6 - Plot 5, plus change mapping: cyl onto fill
ggplot(mtcars, aes(mpg, fill = cyl)) +
  geom_histogram(binwidth = 1, position = "identity", alpha = 0.4)
```



Overlapping histograms (2)

In the video you saw how to make line plots using time series data. To explore this topic, you'll use the `economics` data frame, which contains time series for unemployment and population statistics from the Federal Reserve Bank of St. Louis in the US. The data is contained in the `ggplot2` package.

To begin with, you can look at how the median unemployment time and the unemployment rate (the number of unemployed people as a proportion of the population) change over time.

In the next exercises, you'll explore to how add embellishments to the line plots, such as recession periods.

INSTRUCTIONS

Print out the `head()` of the `economics` data frame.

Use the `economics` data frame to plot date on the x axis and `unemploy` on the y-axis. Use `geom_line()`.

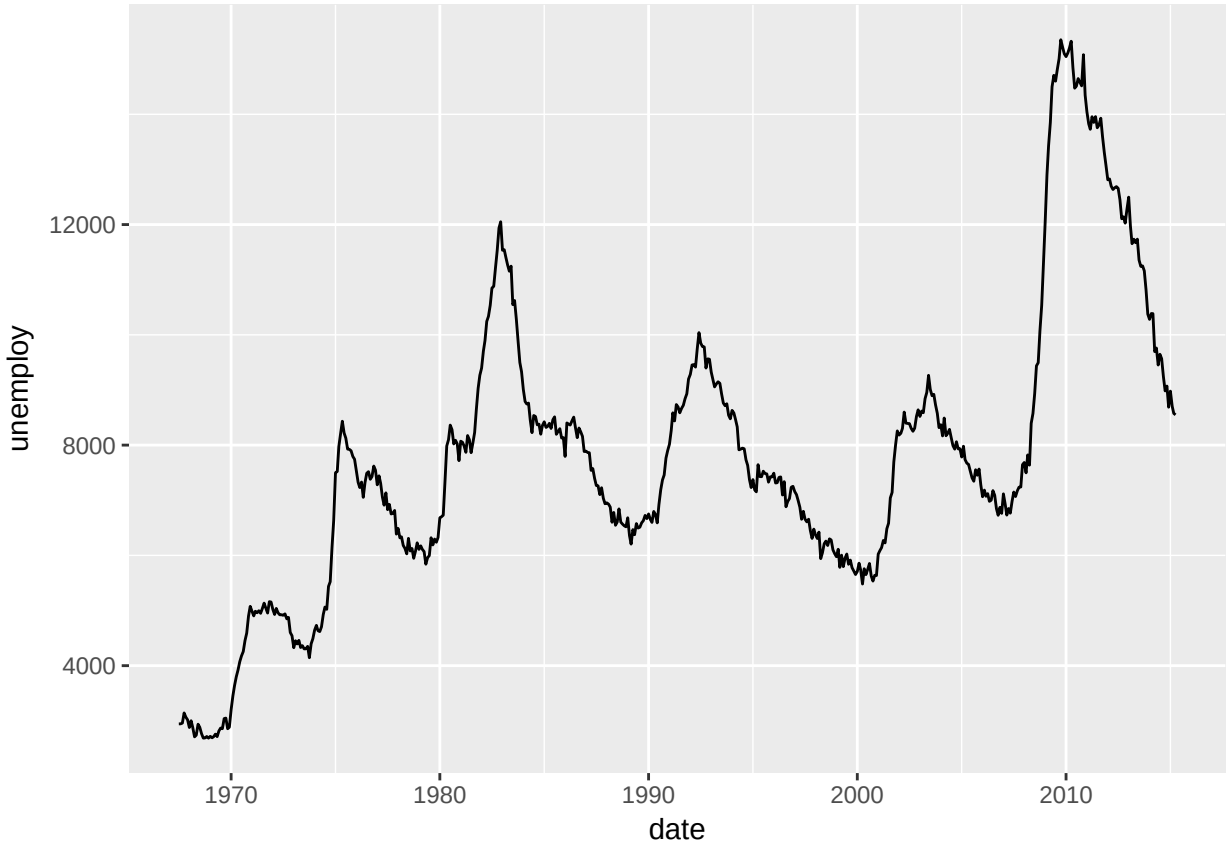
Copy, paste and adjust the code for the previous instruction: instead of `unemploy`, plot `unemploy/pop` to represent the fraction of the total population that is unemployed.

```
# Print out head of economics
head(economics)
```

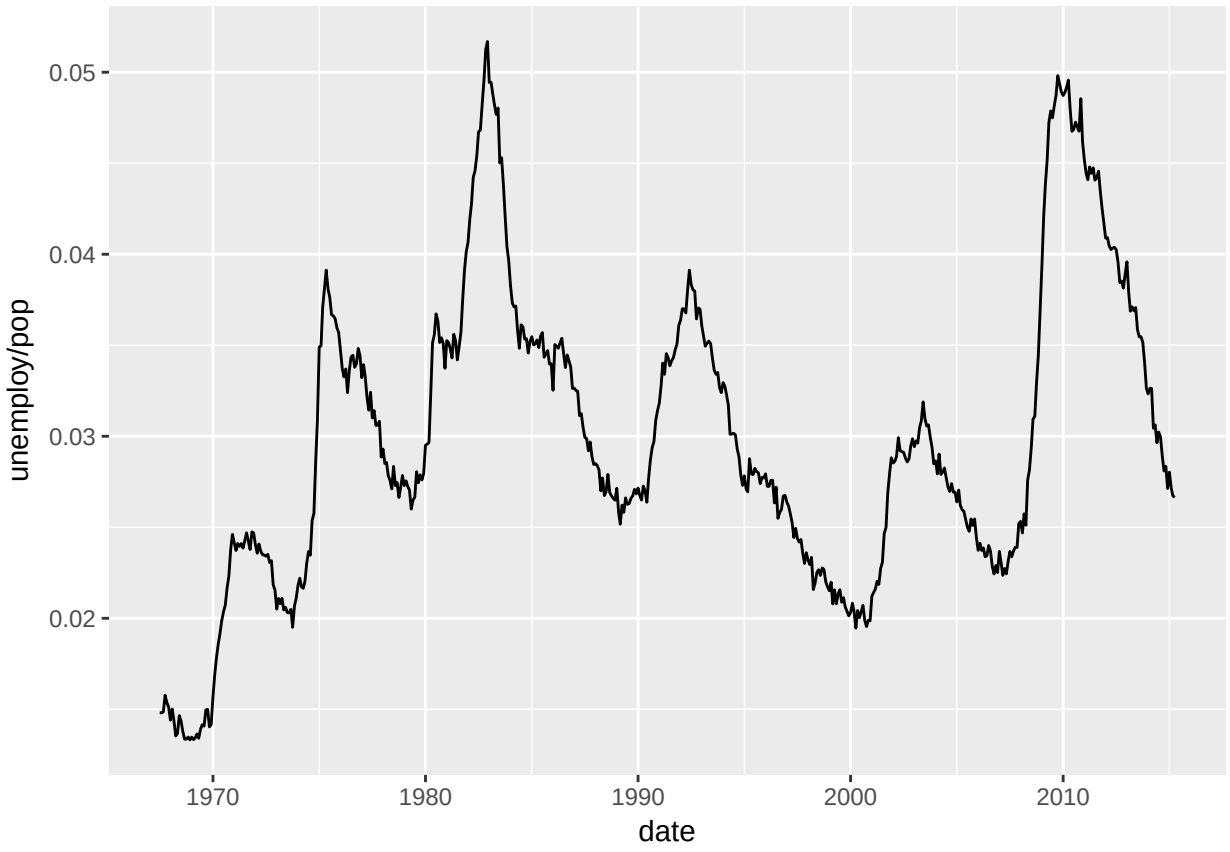
```
## # A tibble: 6 x 6
##   date         pce    pop psavert uempmed unemploy
##   <date>     <dbl>  <int>   <dbl>   <dbl>    <int>
## 1 1967-07-01  507. 198712    12.5    4.50     2944
## 2 1967-08-01  510. 198911    12.5    4.70     2945
## 3 1967-09-01  516. 199113    11.7    4.60     2958
```

```
## 4 1967-10-01  513. 199311    12.5    4.90     3143
## 5 1967-11-01  518. 199498    12.5    4.70     3066
## 6 1967-12-01  526. 199657    12.1    4.80     3018
```

```
# Plot unemploy as a function of date using a line plot
ggplot(economics, aes(x = date, y = unemploy)) +
  geom_line()
```




```
# Adjust plot to represent the fraction of total population that is unemployed
ggplot(economics, aes(x = date, y = unemploy/pop)) +
  geom_line()
```



Periods of recession

By themselves, time series often contain enough valuable information, but you always want to maximize the number of variables you can show in a plot. This allows you (and your viewers) to begin making comparisons between those variables that would otherwise be difficult or impossible.

Here, you'll add shaded regions to the background to indicate recession periods. How do unemployment rate and recession period interact with each other?

In addition to the `economics` dataset from before, you'll also use the `recess` dataset for the periods of recession. The `recess` data frame contains 2 variables: the `begin` period of the recession and the `end`. It's already available in your workspace.

INSTRUCTIONS

Expand the command from the previous exercise with `geom_rect()`. You will use this geom layer to draw rectangles across the recession periods. There a few pitfalls here:

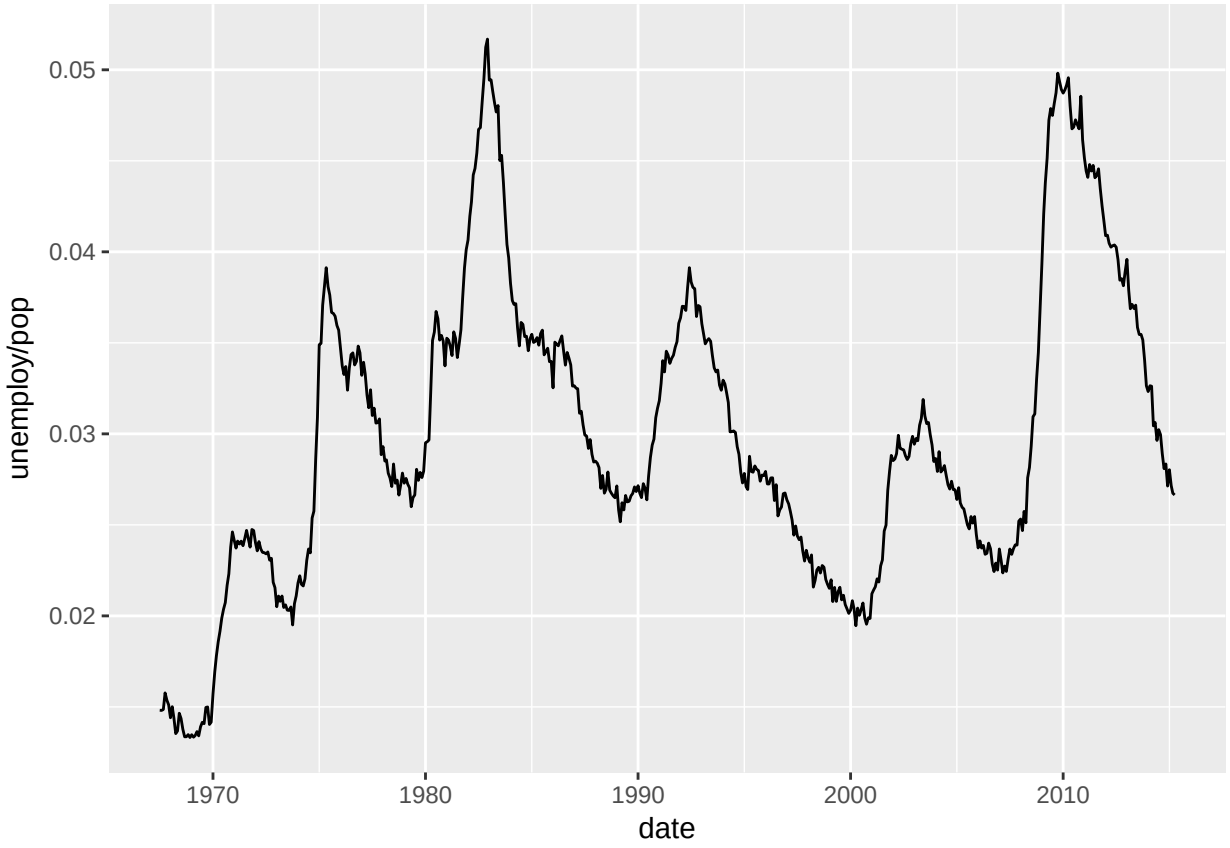
`geom_rect()` uses the recess dataset, so pass this directly as `data = recess` inside `geom_rect()`.

The `geom_rect()` command shouldn't inherit aesthetics from the base `ggplot()` command it belongs to. It would result in an error, since you're using a different dataset and it doesn't contain `unemploy` or `pop`. That's why you should specify `inherit.aes = FALSE` in `geom_rect()`.

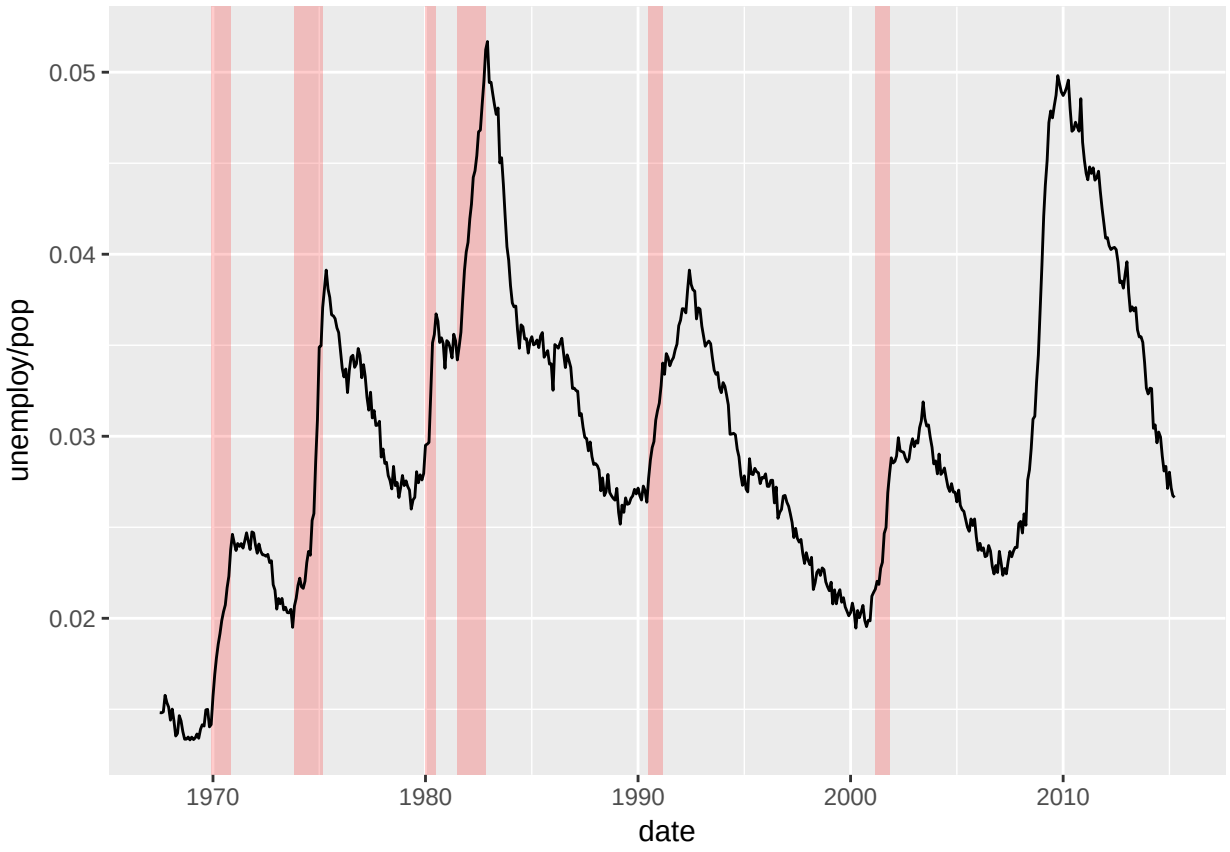
`geom_rect()` needs four aesthetics: `xmin`, `xmax`, `ymin` and `ymax`. These should be set to `begin`, `end` and `-Inf`, `+Inf`, respectively. Define them within `aes()`.

The rectangles you add will be black and opaque by default. Set `fill` to `"red"` and `alpha` to `0.2` to improve this. Define them outside `aes()`.

```r
# Basic line plot
ggplot(economics, aes(x = date, y = unemploy/pop)) +
  geom_line()
```




```r
# Expand the following command with geom_rect() to draw the recess periods
ggplot(economics, aes(x = date, y = unemploy/pop)) +
  geom_rect(data = recess,
        aes(xmin = begin, xmax = end, ymin = -Inf, ymax = +Inf),
        inherit.aes = FALSE, fill = "red", alpha = 0.2) +
  geom_line()
```



Multiple time series, part 1

In the data chapter we discussed how the form of your data affects how you can plot it. Here, you'll explore that topic in the context of multiple time series.

The dataset you'll use contains the global capture rates of seven salmon species from 1950 - 2010.

In your workspace, the following dataset is available:

- `fish.species`: Each variable (column) is a Salmon `Species` and each observation (row) is one `Year`.

To get a multiple time series plot, however, both `Year` and `Species` should be in their own column. You need tidy data: one variable per column. Once you have that you can get the plot shown in the viewer by mapping `Year` to the x aesthetic and `Species` to the color aesthetic.

You'll use the `gather()` function of the tidyr package, which is already loaded for you.

INSTRUCTIONS

Use `gather()` to move from `fish.species` to a tidy data frame, `fish.tidy`. This data frame should have three columns: `Year` (int), `Species` (factor) and `Capture` (int).

`gather()` takes four arguments: the original data frame (`fish.species`), the name of the key column (`Species`), the name of the value column (`Capture`) and the name of the grouping variable, with a minus in front (`-Year`). They can all be specified as object names (i.e. no `""`).

```
# Check the structure as a starting point
str(fish.species)
```

```
## 'data.frame':    61 obs. of  8 variables:
##  $ Year    : int  1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 ...
##  $ Pink    : int  100600 259000 132600 235900 123400 244400 203400 270119 200798 200085 ...
##  $ Chum    : int  139300 155900 113800 99800 148700 143700 158480 125377 132407 113114 ...
##  $ Sockeye : int  64100 51200 58200 66100 83800 72000 84800 69676 100520 62472 ...
##  $ Coho    : int  30500 40900 33600 32400 38300 45100 40000 39900 39200 32865 ...
##  $ Rainbow : int  0 100 100 100 100 100 100 100 100 100 ...
##  $ Chinook : int  23200 25500 24900 25300 24500 27700 25300 21200 20900 20335 ...
##  $ Atlantic: int  10800 9701 9800 8800 9600 7800 8100 9000 8801 8700 ...
```

```
# Use gather to go from fish.species to fish.tidy
fish.tidy <- gather(fish.species, key = Species, value = Capture, group = -Year)
```

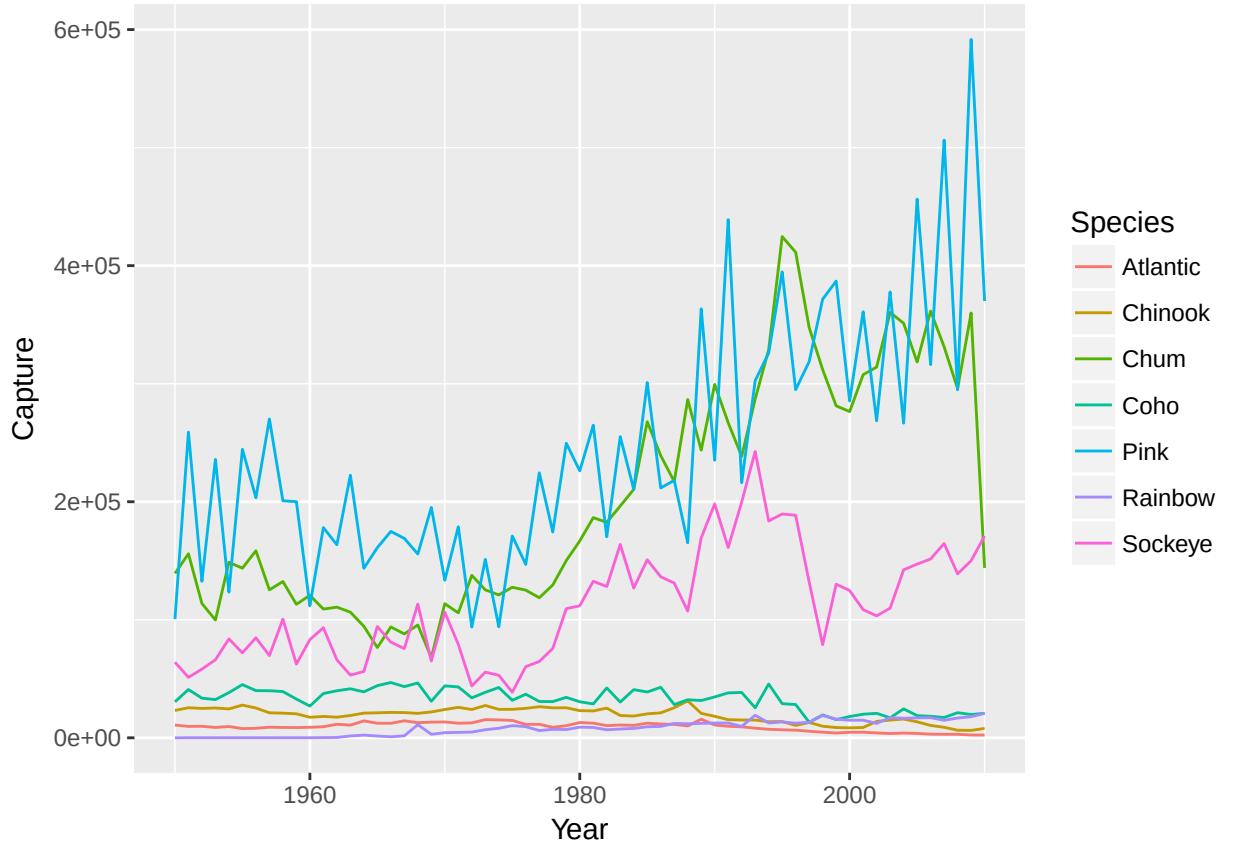
Multiple time series, part 2

Now that you have tidy data, you're ready to make your plot! The data frame `fish.tidy` is already available in the workspace, so you can start right away!

INSTRUCTIONS

Use `ggplot2` and everything you've learned to recreate the plot shown on the right.

```
# Recreate the plot shown on the right
ggplot(fish.tidy, aes(x = Year, y = Capture, color = Species)) +
  geom_line()
```



Chapter 5: qplot and wrap-up

In this chapter you'll learn about qplot; it is a quick and dirty form of ggplot2. It's not as intuitive as the full-fledged ggplot() function but may be useful in specific instances. This chapter also features a wrap-up video and corresponding data visualization exercises.

Using qplot

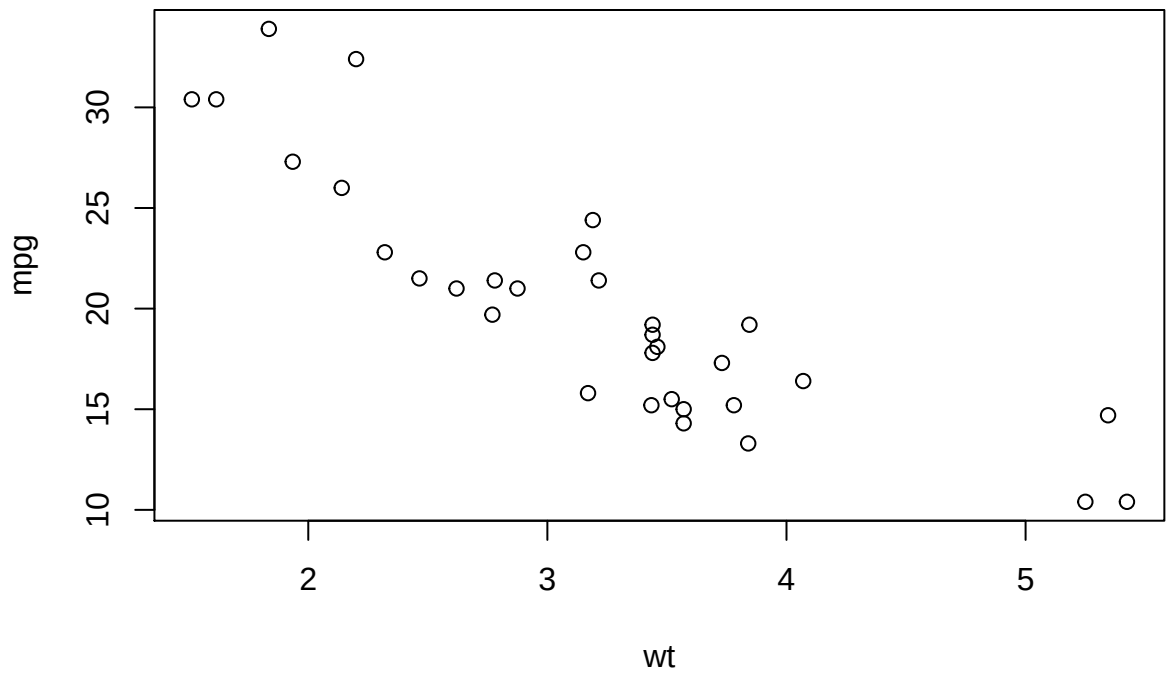
For simple exploratory plots, there are a variety of functions available. `ggplot2` offers a powerful and diverse array of functions, but `qplot()` allows for quick and dirty plots. Plus, you should also be familiar with basic plotting notation.

INSTRUCTIONS

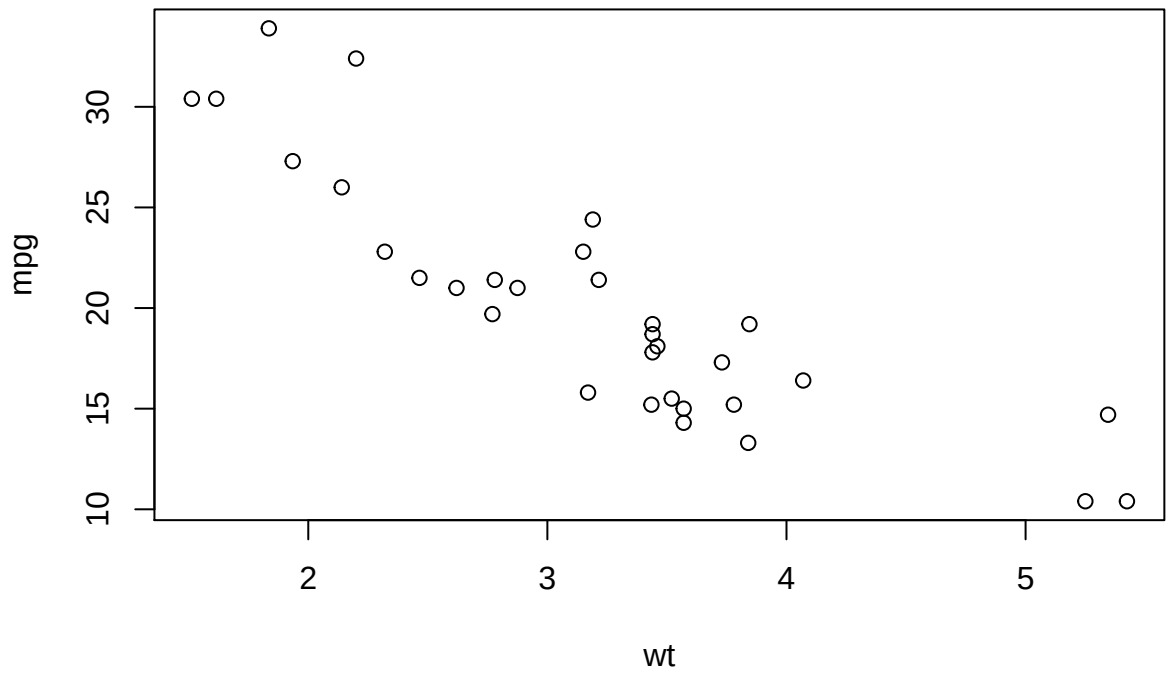
Have a look at the base R plotting function. It plots `mpg` on the y-axis against `wt` on the x-axis. Create the same plot using `ggplot()`.

Create the same plot using `qplot()`.

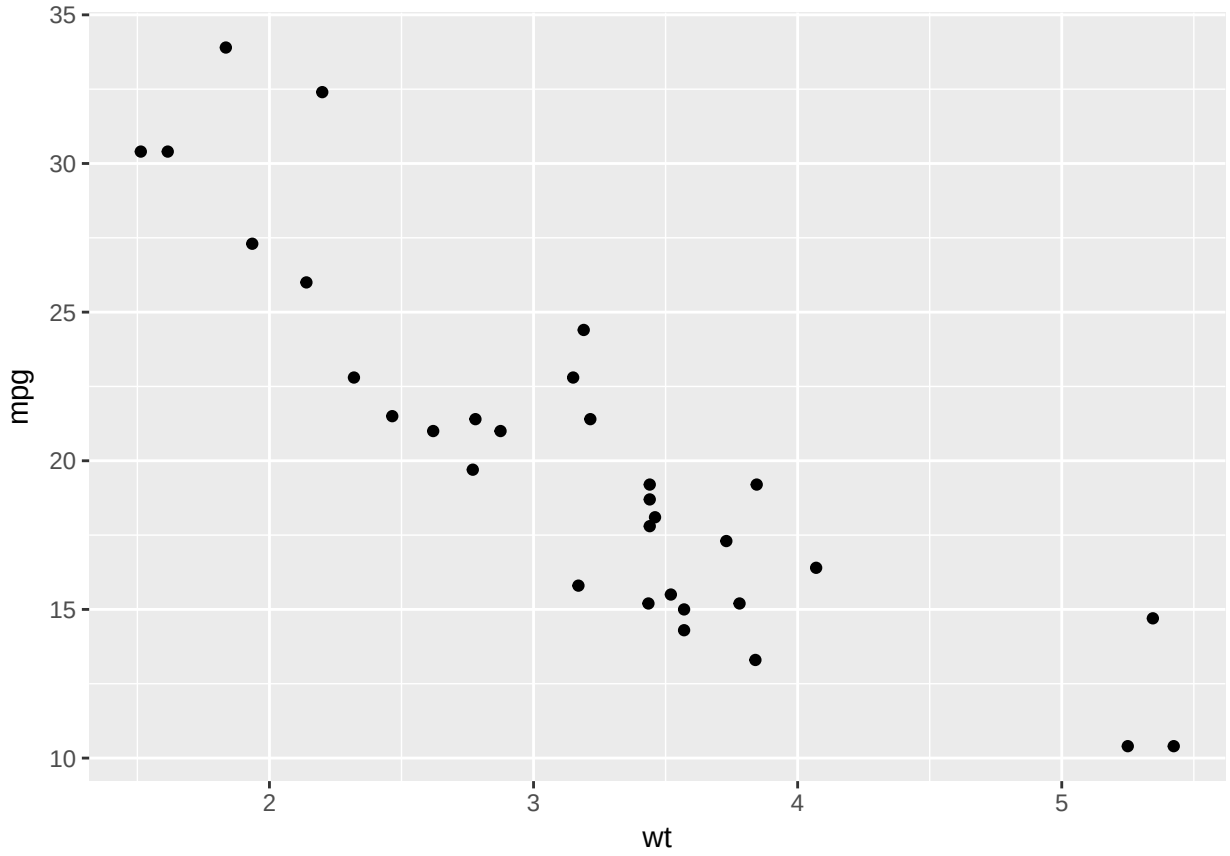
```
# The old way (shown)
plot(mpg ~ wt, data = mtcars) # formula notation
```



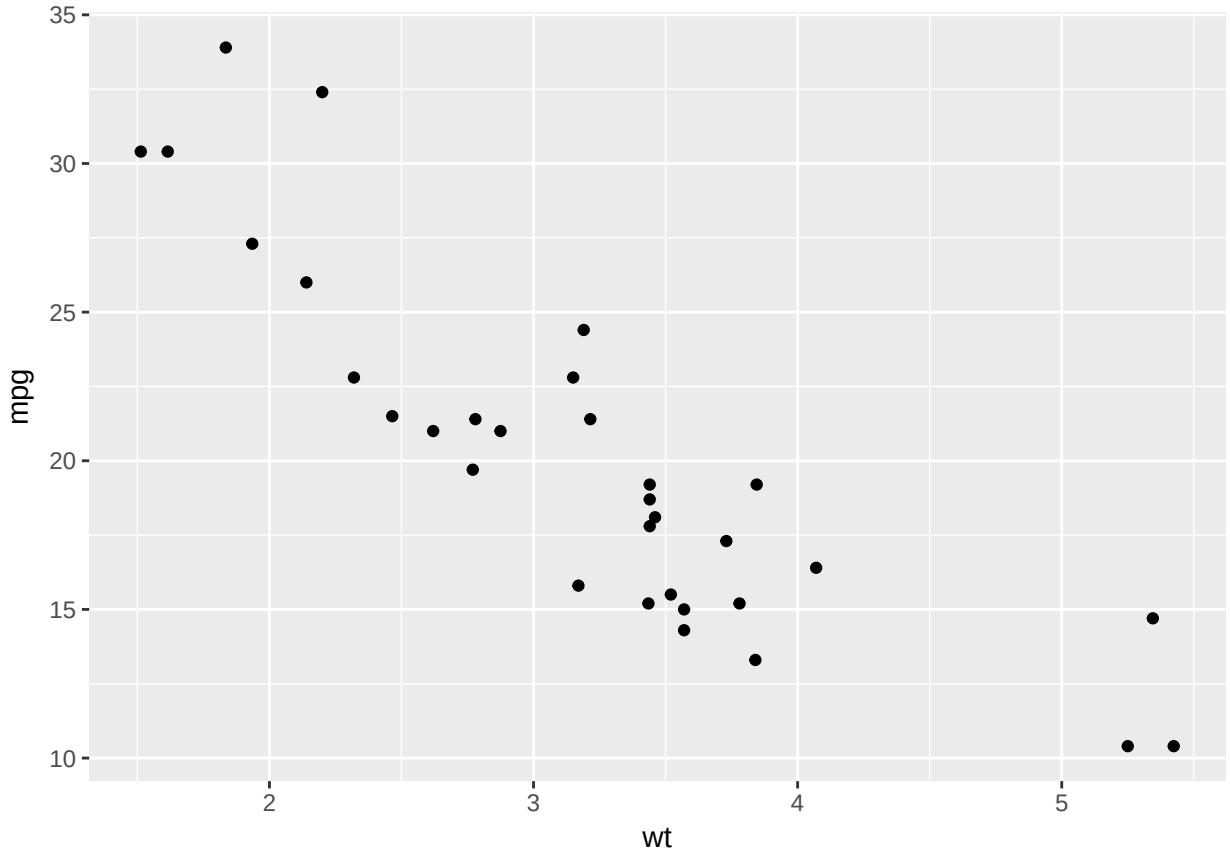

```
with(mtcars, plot(wt, mpg)) # x, y notation
```




```
# Using ggplot:
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```




```
# Using qplot:
qplot(wt, mpg, data = mtcars)
```



Using aesthetics

You already saw how some aesthetics are only applicable to categorical variables, such as shapes and linetypes. But just because others, such as size and color (and hence fill), can be applied to both categorical and continuous variables, doesn't mean that they're suitable for both.

INSTRUCTIONS

A basic scatter plot of `mpg` vs. `wt` from the `mtcars` dataset, made with `qplot()`, is provided.

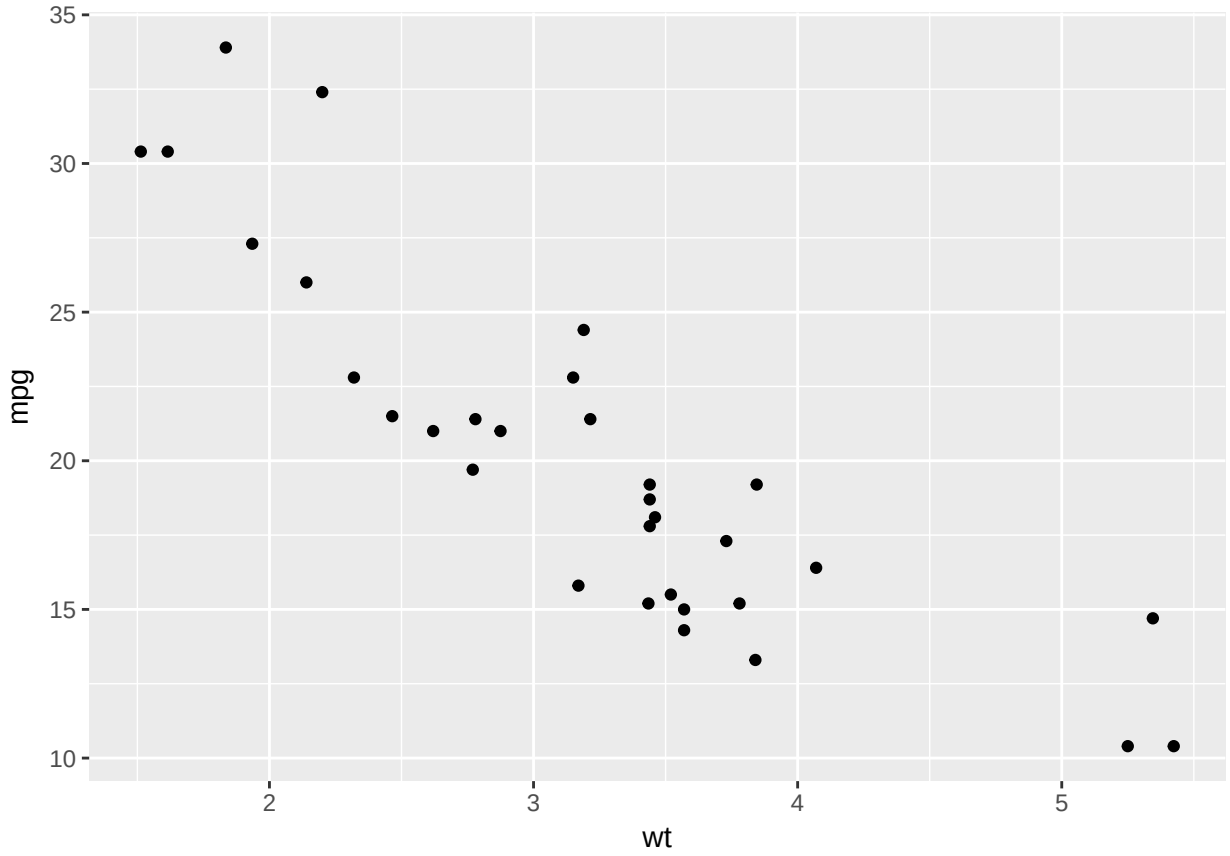
Using `qplot()`, map the categorical variable `cyl` onto `size`. Remember, you'll have to wrap the variable name in a `factor()` function to convert to a categorical variable.

Use `qplot()` again to the same plot, except with `gear` mapped onto `size`.

Using `qplot()`, map the continuous variable `hp` onto `color`.

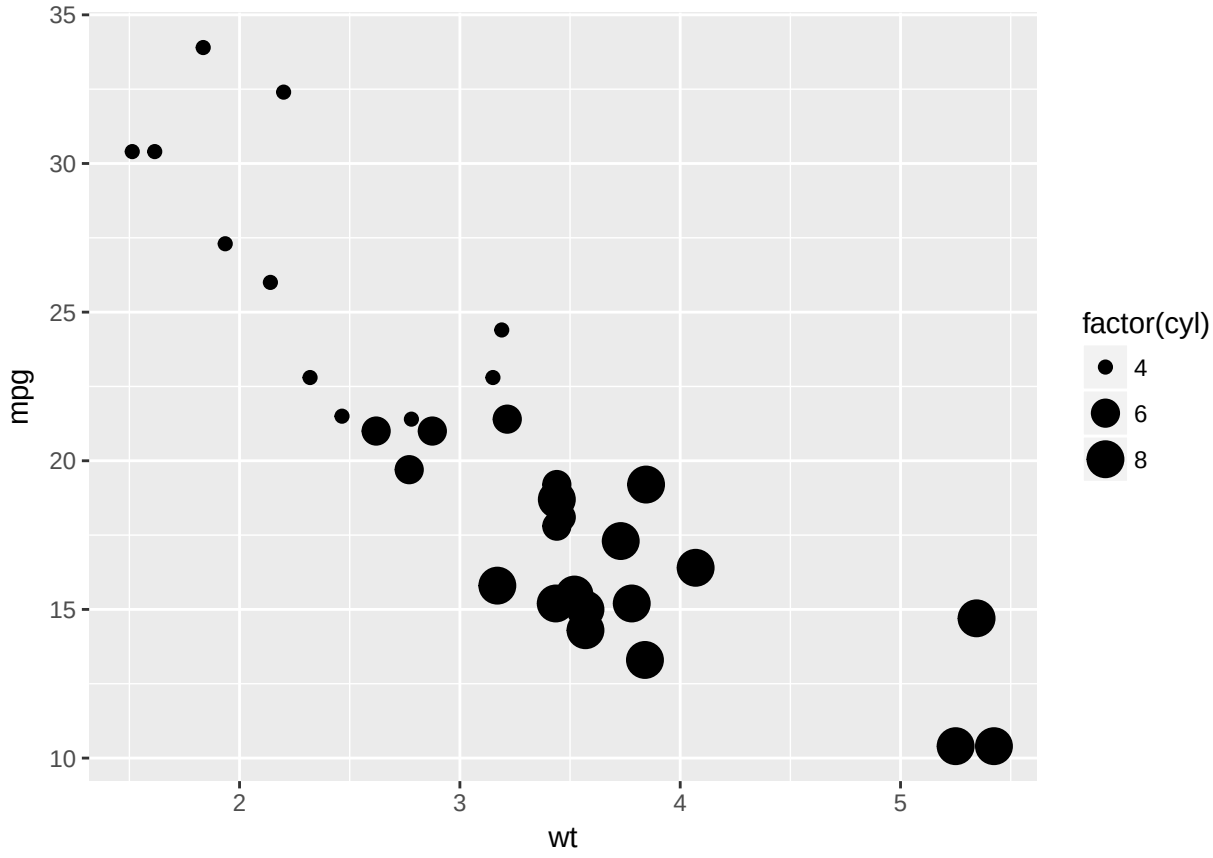
Use `qplot()` again to the same plot, except with `qsec` mapped onto `color`.

```
# basic qplot scatter plot:
qplot(wt, mpg, data = mtcars)
```



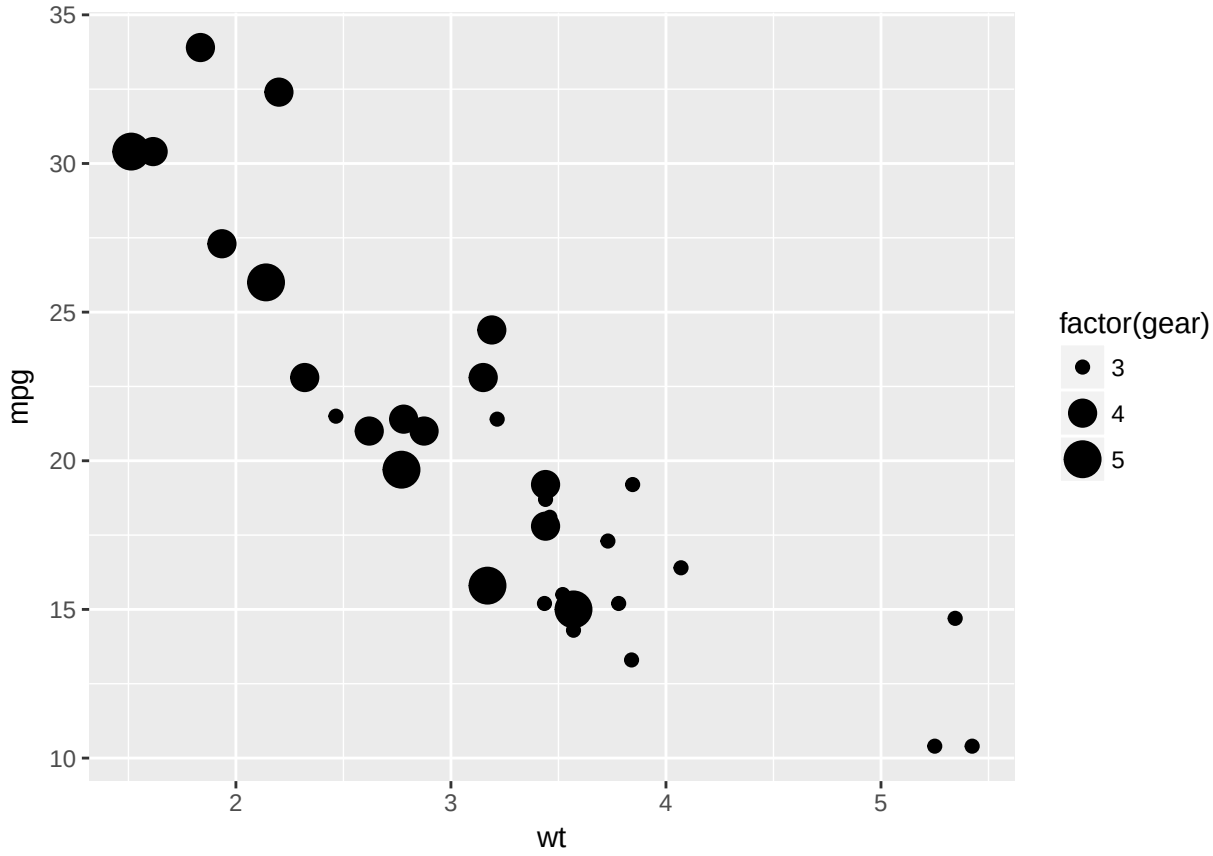

```
# Categorical variable mapped onto size:
# cyl
qplot(wt, mpg, data = mtcars, size = factor(cyl))
```

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

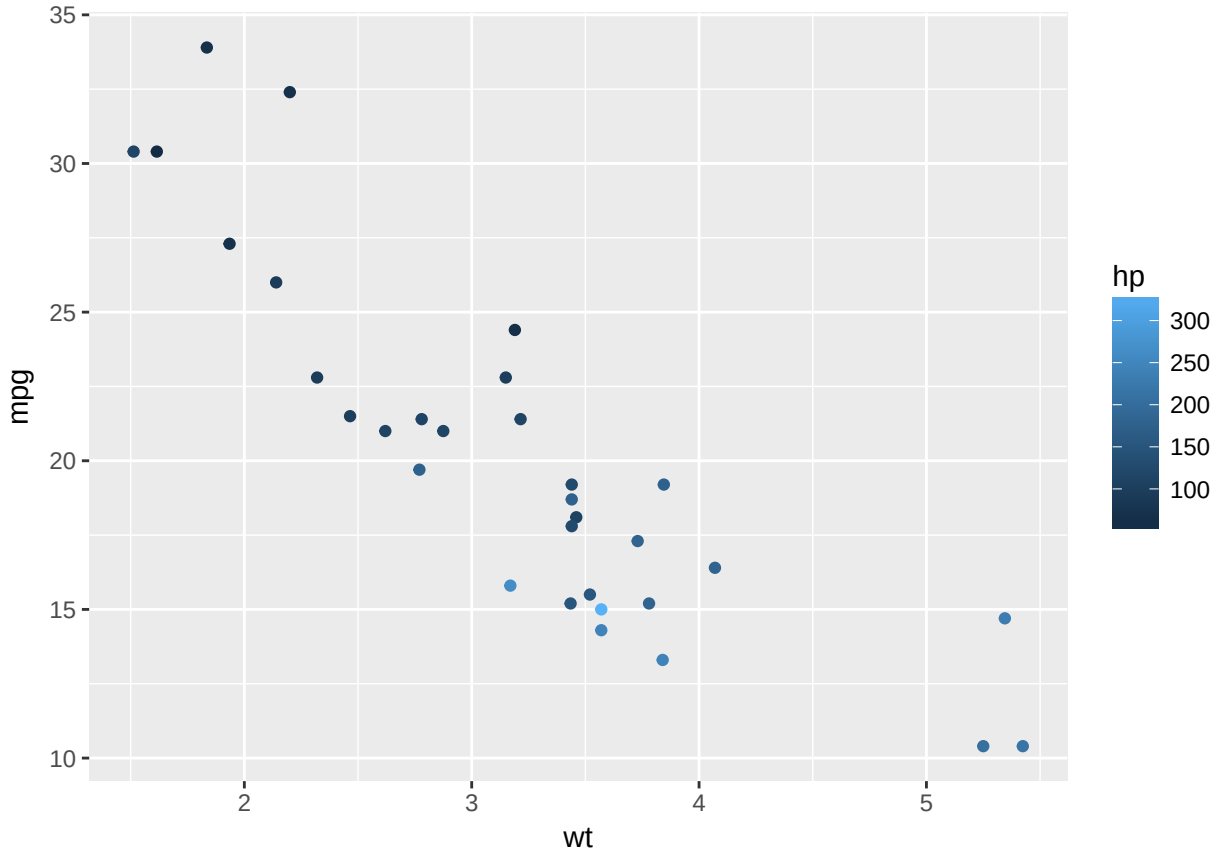



```
# gear
qplot(wt, mpg, data = mtcars, size = factor(gear))
```

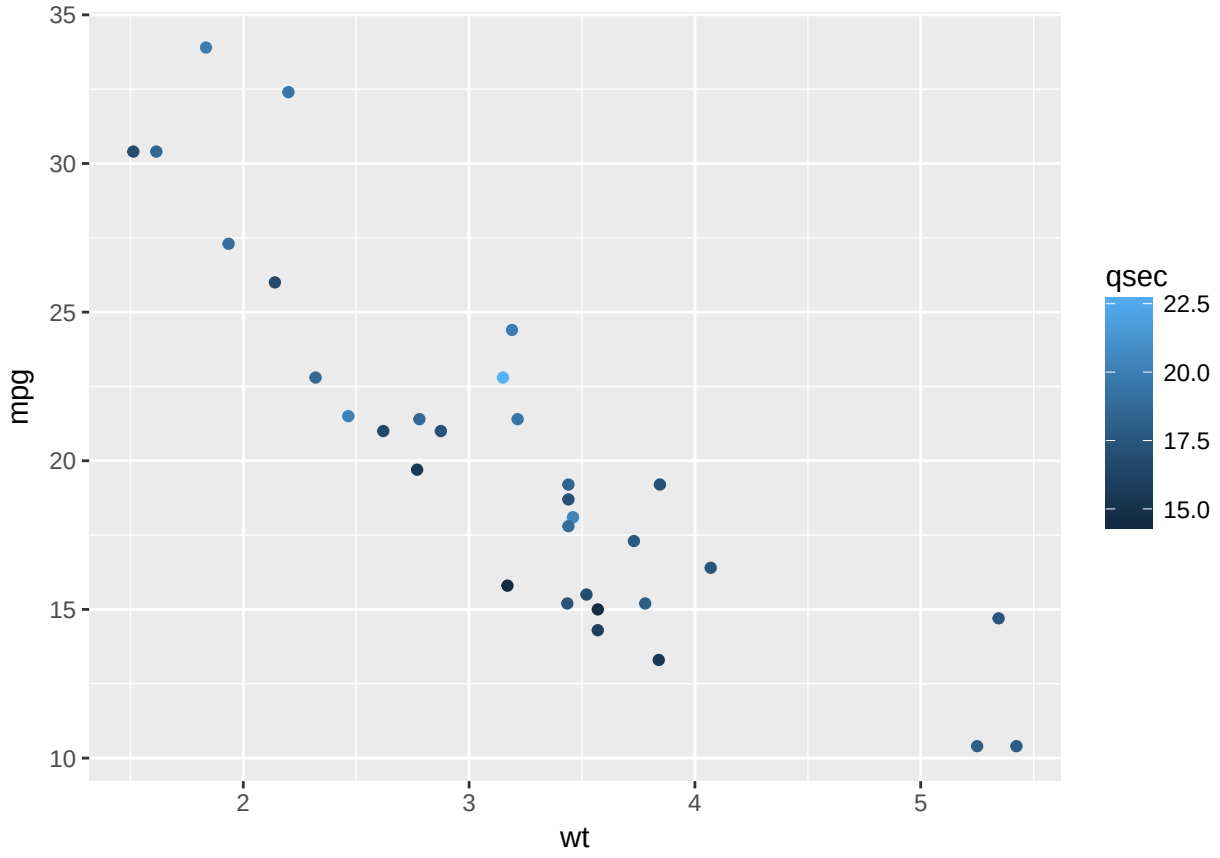
```
## Warning: Using size for a discrete variable is not advised.
```




```
# Continuous variable mapped onto col:
# hp
qplot(wt, mpg, data = mtcars, color = hp)
```




```
# qsec
qplot(wt, mpg, data = mtcars, color = qsec)
```



Choosing geoms, part 1

`qplot` automatically takes care of assigning a geom to our plot given the type of data, but you can specify the geom yourselves.

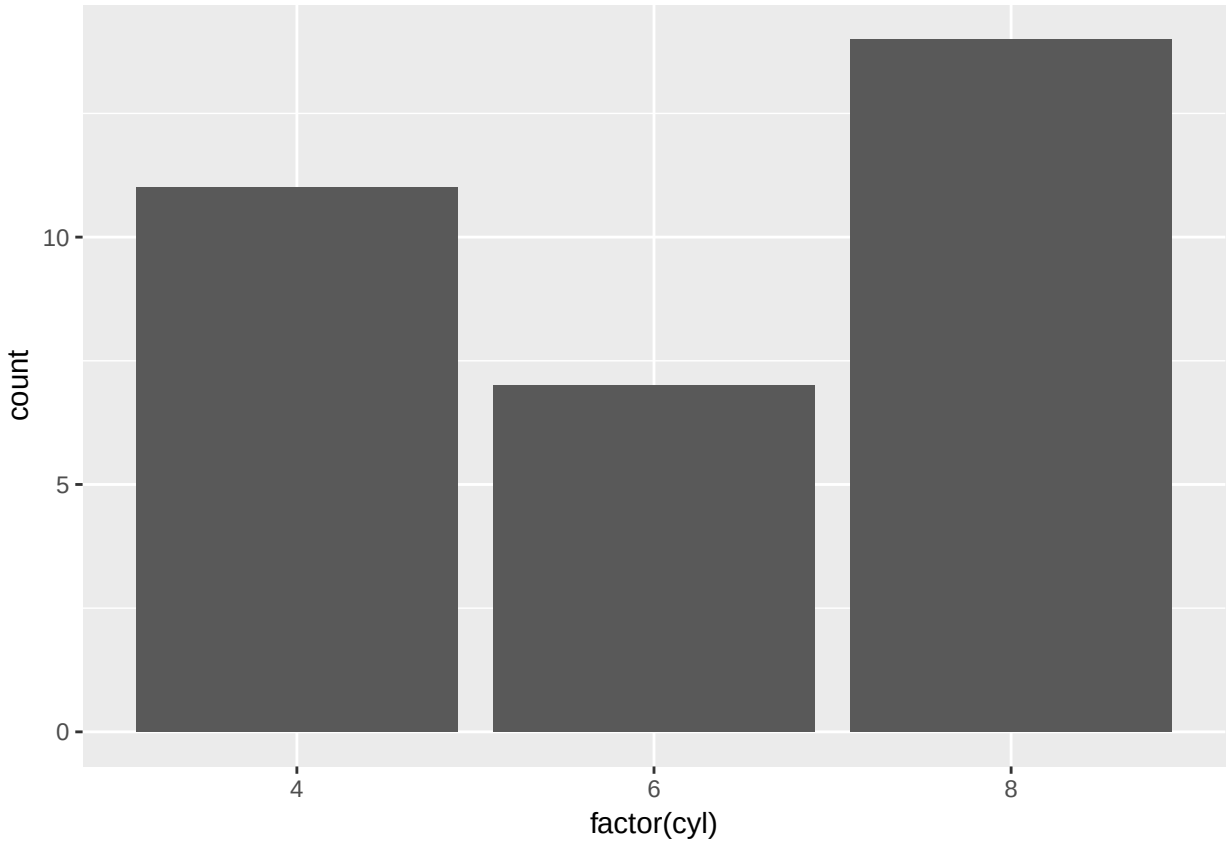
INSTRUCTIONS

Make a quick plot using `qplot()`. Use the `mtcars` dataset and plot only `factor(cyl)` onto `x`. Which geom does `qplot()` choose?

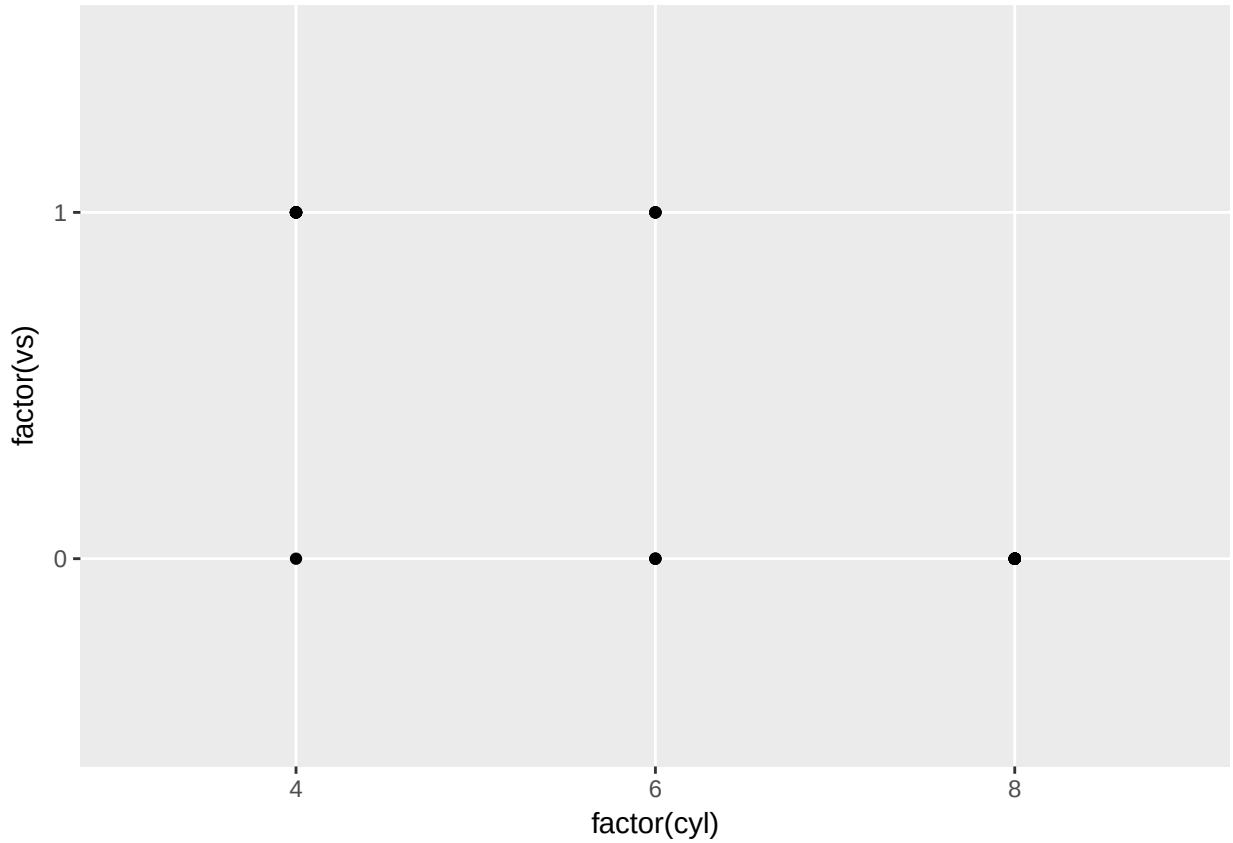
Extend the previous `qplot()` command so that it maps `factor(vs)` onto `y`. Which geom does `qplot()` use now?

The previous plot had overlapping points. For the last instruction, copy the previous `qplot()`, but manually set geom to `"jitter"` in `qplot()`.

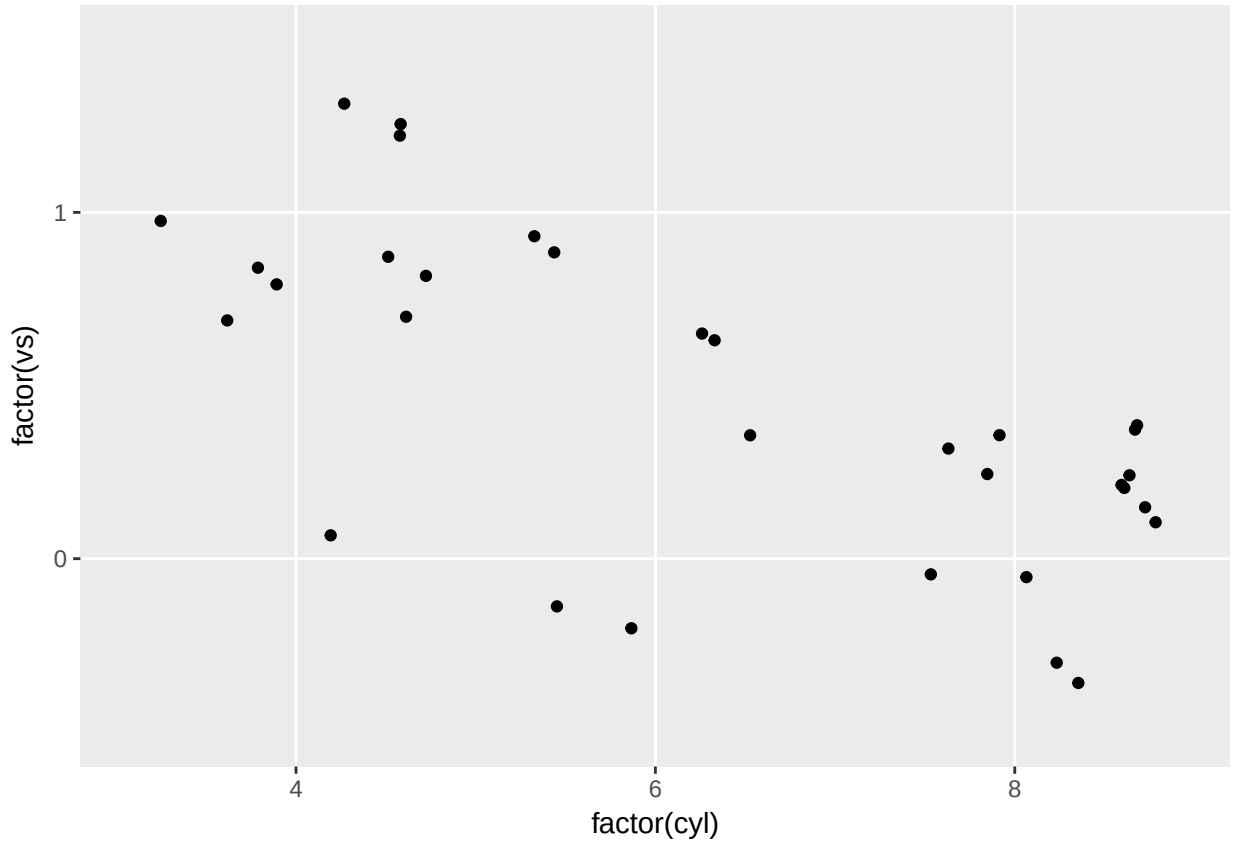
```
# qplot() with x only
qplot(factor(cyl), data = mtcars)
```




```
# qplot() with x and y
qplot(factor(cyl), factor(vs), data = mtcars)
```




```
# qplot() with geom set to jitter manually
qplot(factor(cyl), factor(vs), data = mtcars, geom = "jitter")
```



Choosing geoms, part 2 - dotplot

Some naming conventions:

- Scatter plots:
- Continuous `x`, continuous `y`.
- Dot plots: Categorical `x`, continuous `y`.

You use `geom_point()` for both plot types. Jittering position is set in the `geom_point()` layer.

However, to make a "true" dot plot, you can use `geom_dotplot()`. The difference is that unlike `geom_point()`,`geom_dotplot()` uses a binning statistic. Binning means to cut up a continuous variable (the `y` in this case) into discrete "bins". You already saw binning with `geom_histogram()` (see this exercise for a refresher).

One thing to notice is that `geom_dotplot()` uses a different plotting symbol to `geom_point()`. For these symbols, the `color` aesthetic changes the color of its border, and the `fill` aesthetic changes the color of its interior.

Let's take a look at how the two geoms compare.

INSTRUCTIONS

A "basic" dot plot is shown in the viewer (see the code in the editor). Here, `cyl` (categorical) is mapped onto the `x` and `wt` (continuous) is mapped onto the `y` aesthetic. For this exercise we've already converted `am` to a factor variable for you.

1 - Re-draw that plot in the viewer as a "true" dot plot.

Add a dotplot geom by calling `geom_dotplot()`.

Set the arguments `stackdir = "center"` and `binaxis = "y"`. These are our standard settings, but take a look at the help pages and try different settings to get familiar with these arguments.

2 - Convert the previous `ggplot()` command to a `qplot()` command.

```
# cyl and am are factors, wt is numeric
class(mtcars$cyl)
```

```
## [1] "factor"
```

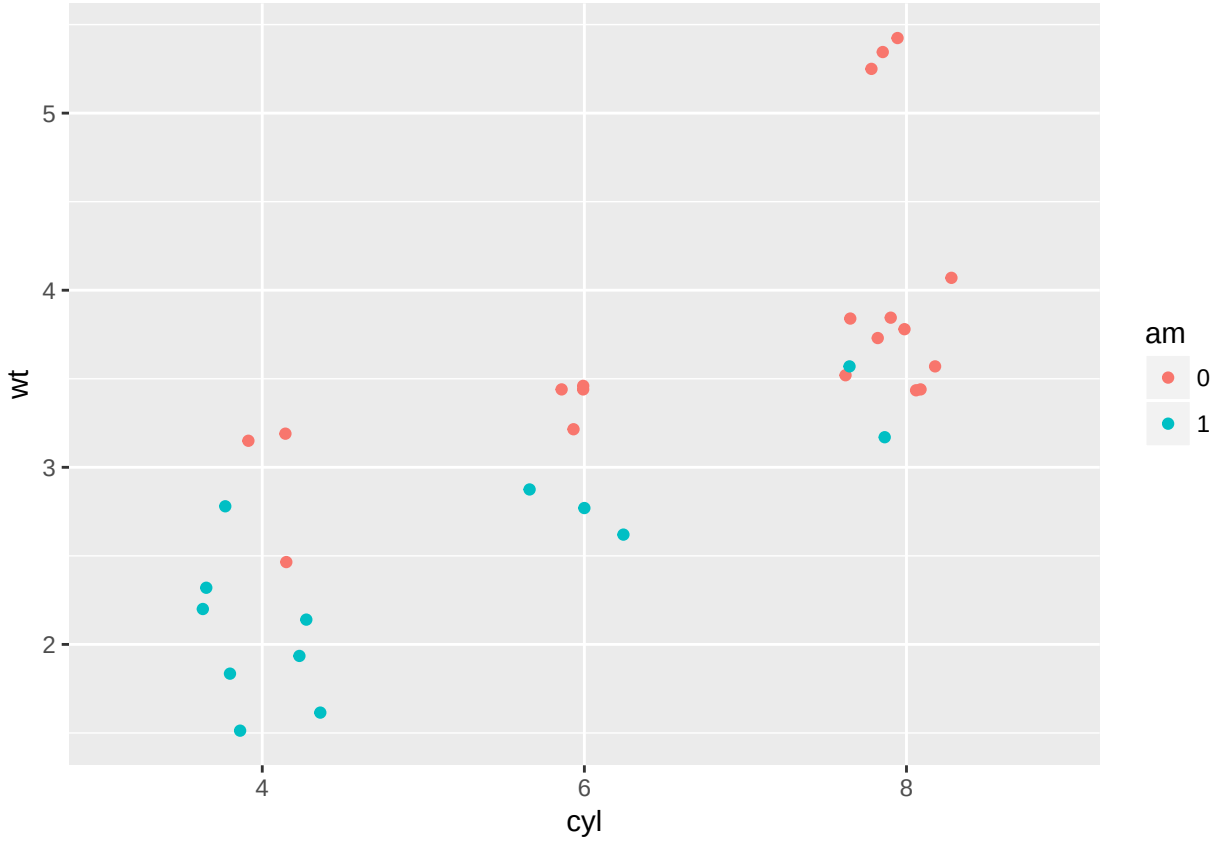
```
class(mtcars$am)
```

```
## [1] "factor"
```

```
class(mtcars$wt)
```

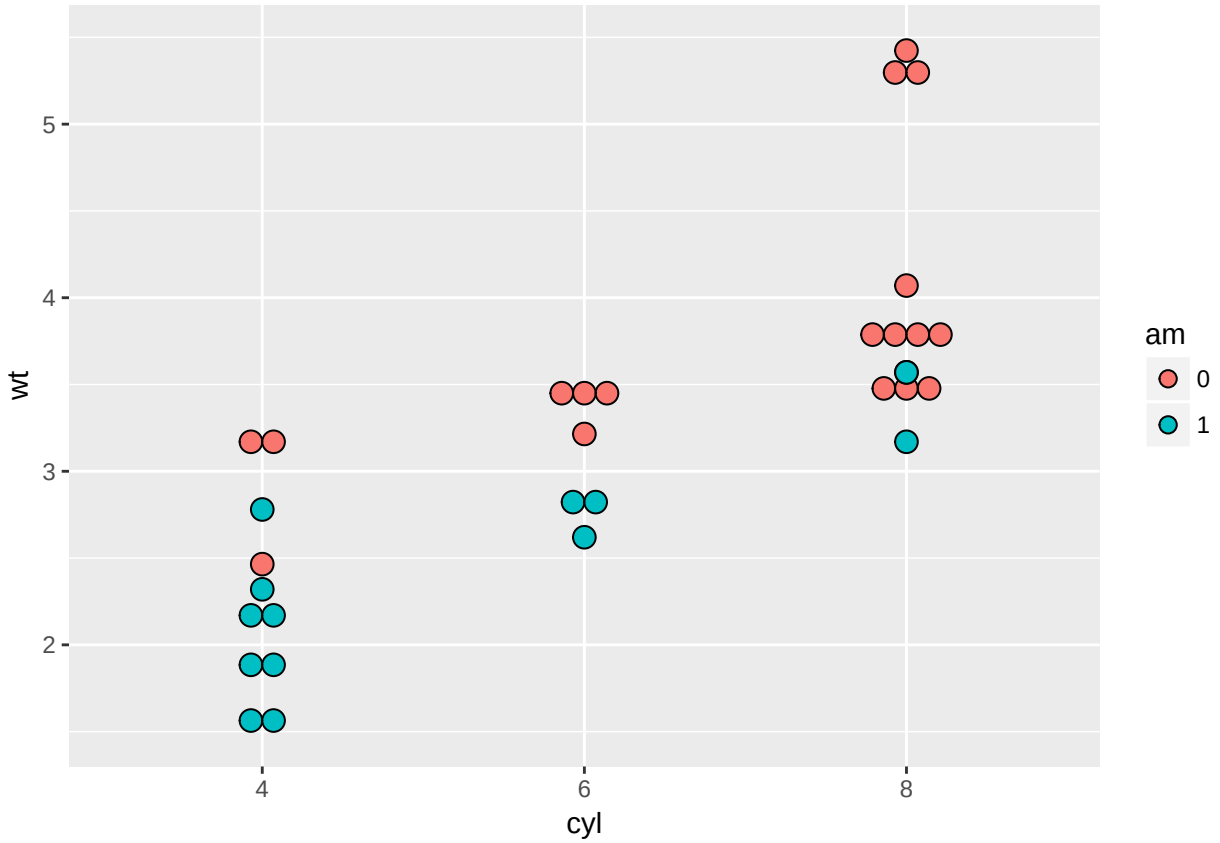
```
## [1] "numeric"
```

```
# "Basic" dot plot, with geom_point():
ggplot(mtcars, aes(cyl, wt, col = am)) +
  geom_point(position = position_jitter(0.2, 0))
```



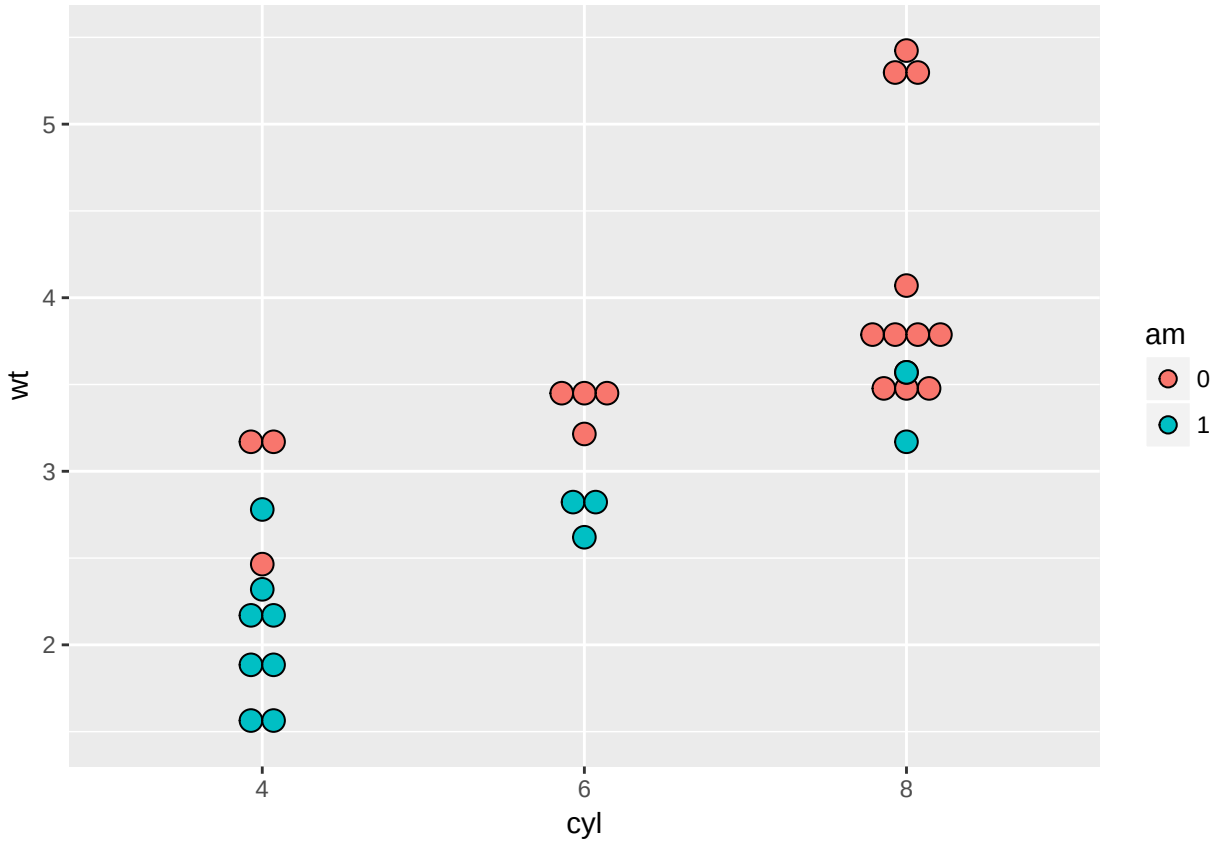

```
# 1 - "True" dot plot, with geom_dotplot():
ggplot(mtcars, aes(cyl, wt, fill = am)) +
  geom_dotplot(binaxis = "y", stackdir = "center")
```

```
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```




```
# 2 - qplot with geom "dotplot", binaxis = "y" and stackdir = "center"
qplot(
  cyl, wt,
  data = mtcars,
  fill = am,
  geom = "dotplot",
  binaxis = "y",
  stackdir = "center"
)
```

```
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```



Chicken weight

The `ChickWeight` dataset is a data frame which represents the progression of weight of several chicks. The little chicklings are each given a specific diet. There are four types of diet and the farmer wants to know which one fattens the chicks the fastest.

It's time to do some exploratory statistics on the data frame using the techniques you learned in this course! Let's do some `ggplot`-ing!

INSTRUCTIONS

1 - Execute `head(ChickWeight)` to check the first few rows of this dataset. Looks like the data is pretty tidy!

2 - Plot a line for each chick.

Use `ggplot()` and map `Time` to `x` and `weight` to `y` within the `aes()` function. Add `geom_line()` at the end to draw the lines.

To draw one line per chick, add `group = Chick` to the `aes()` of `geom_line()`.

Oops! That looks pretty chaotic and you can't really conclude anything from it. Let's try again.

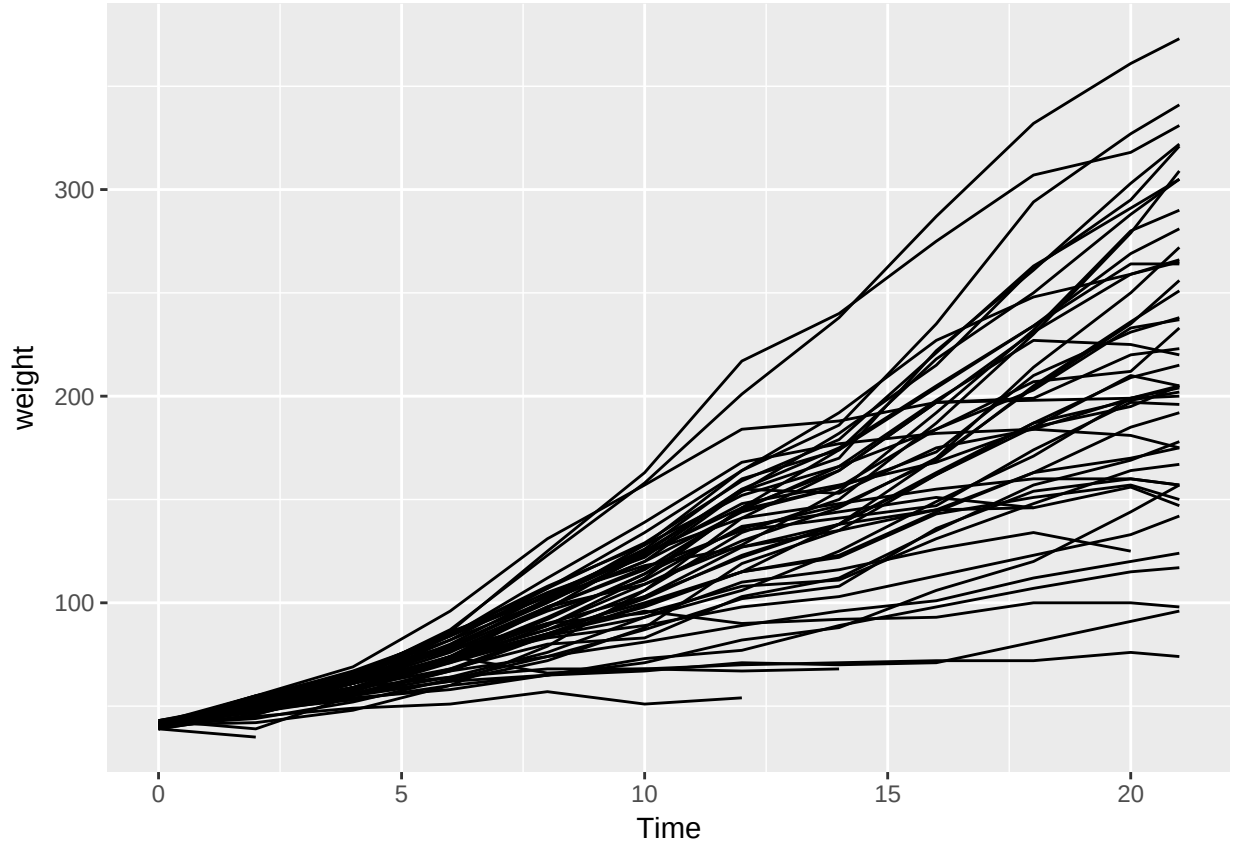
3 - Take plot 2 and add `color = Diet` within the `aes()` of `ggplot()`. There's some more information here, although it would be better to have some summary statistics as well. What do you think would be helpful?

4 - Take plot 3 and add `geom_smooth()` with attributes `lwd` set to `2` and `se` set to `FALSE`. Inside `geom_line()`, set `alpha` of to `0.3`.

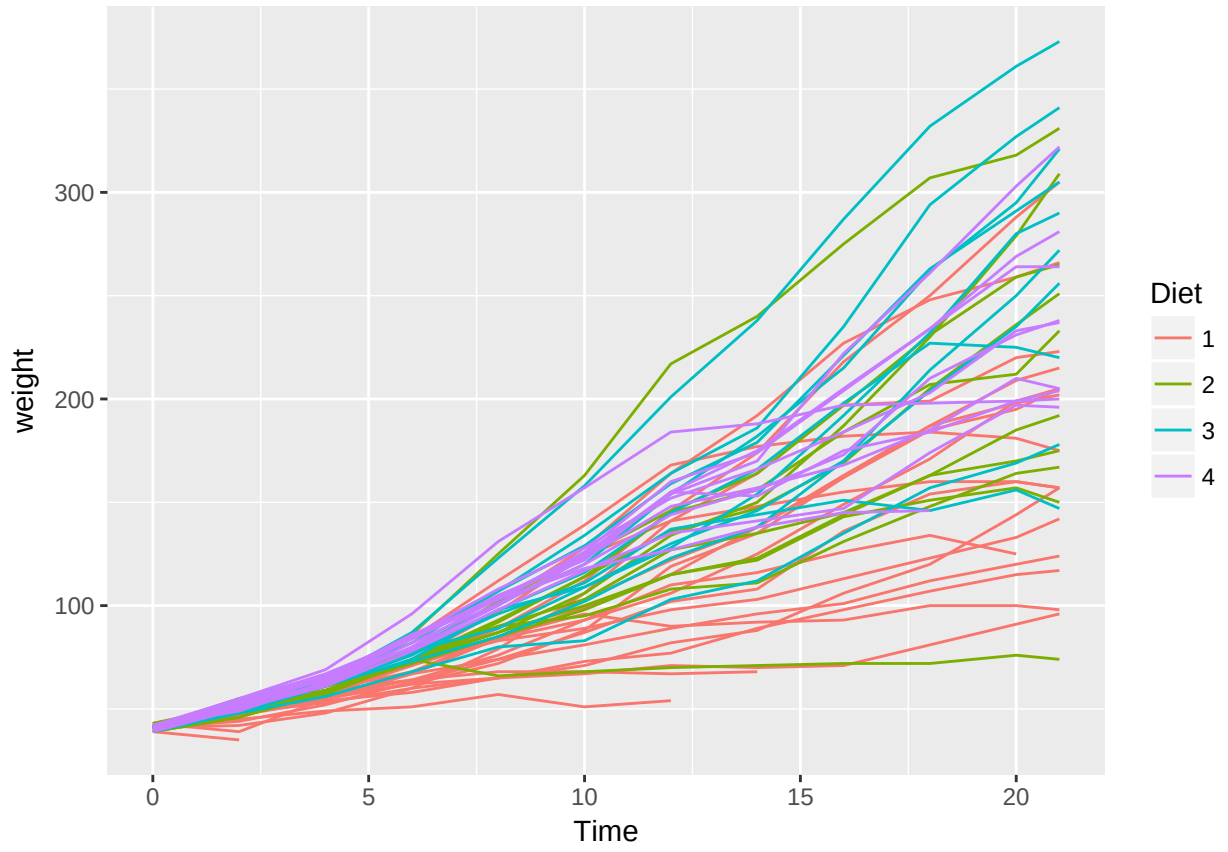
```
# ChickWeight is available in your workspace
# 1 - Check out the head of ChickWeight
head(ChickWeight)
```

```
## Grouped Data: weight ~ Time | Chick
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
## 6     93   10     1    1
```

```
# 2 - Basic line plot
ggplot(ChickWeight, aes(x = Time, y = weight)) +
  geom_line(aes(group = Chick))
```

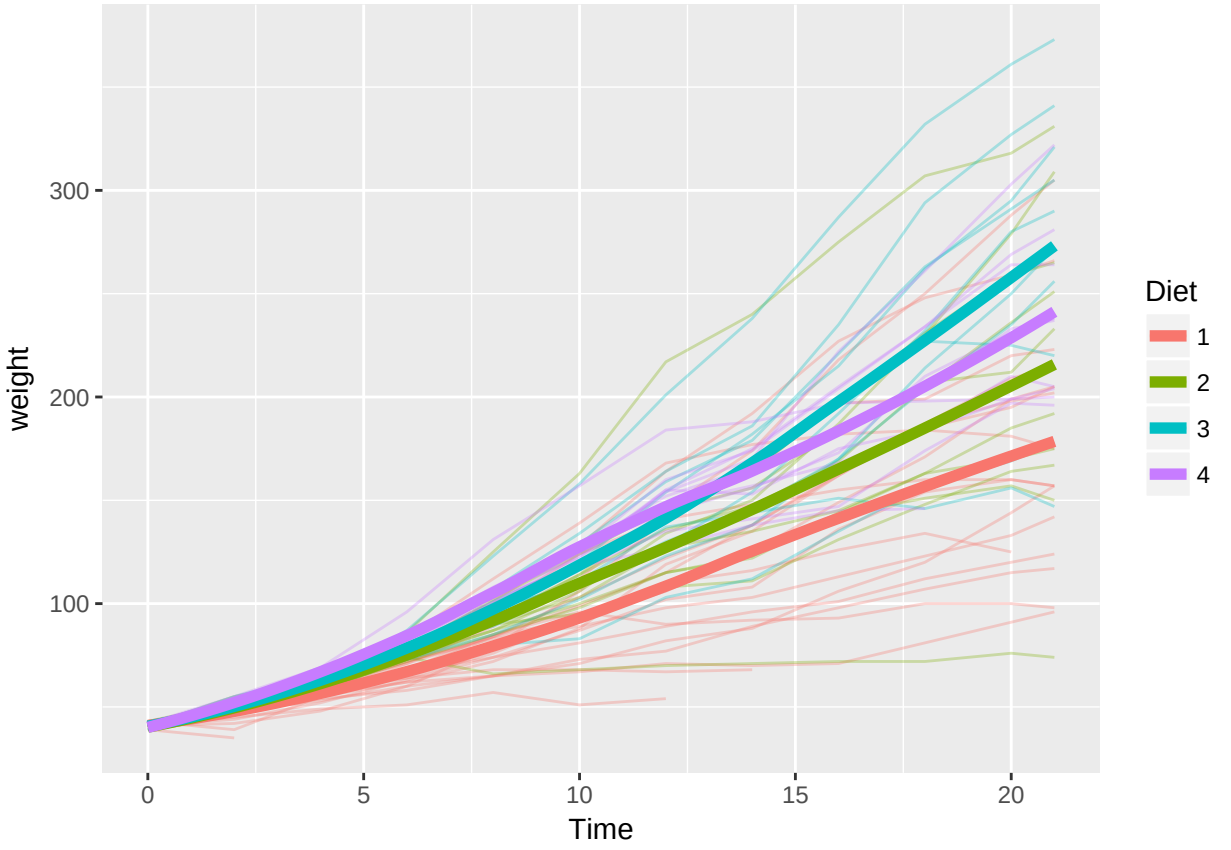



```
# 3 - Take plot 2, map Diet onto col.
ggplot(ChickWeight, aes(x = Time, y = weight, color = Diet)) +
  geom_line(aes(group = Chick))
```




```
# 4 - Take plot 3, add geom_smooth()
ggplot(ChickWeight, aes(x = Time, y = weight, color = Diet)) +
  geom_line(aes(group = Chick), alpha = 0.3) +
  geom_smooth(lwd = 2, se = FALSE)
```

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



Titanic

You've watched the movie Titanic by James Cameron (1997) again and after a good portion of sobbing you decide to investigate whether you'd have a chance of surviving this disaster.

To start your investigation, you decide to do some exploratory visualization with `ggplot()`. You have information on who survived the sinking given their age, sex and passenger class.

INSTRUCTIONS

1 - Have a look at the `str()` of the `titanic` dataset, which has been loaded into your workspace. Looks like the data is pretty tidy!

2 - Plot the distribution of sexes within the classes of the ship.

Use `ggplot()` with the data layer set to `titanic`.

Map `Pclass` onto the `x` axis, `Sex` onto `fill` and draw a dodged bar plot using `geom_bar()`, i.e. set the geom `position` to `"dodge"`.

3 - These bar plots won't help you estimate your chances of survival. Copy the previous bar plot, but this time add a `facet_grid()` layer: `. ~ Survived`.

4 - We've defined a `position` object for you.

5 - Include `Age`, the final variable.

Take plot 3 and add a mapping of `Age` onto the `y` aesthetic.

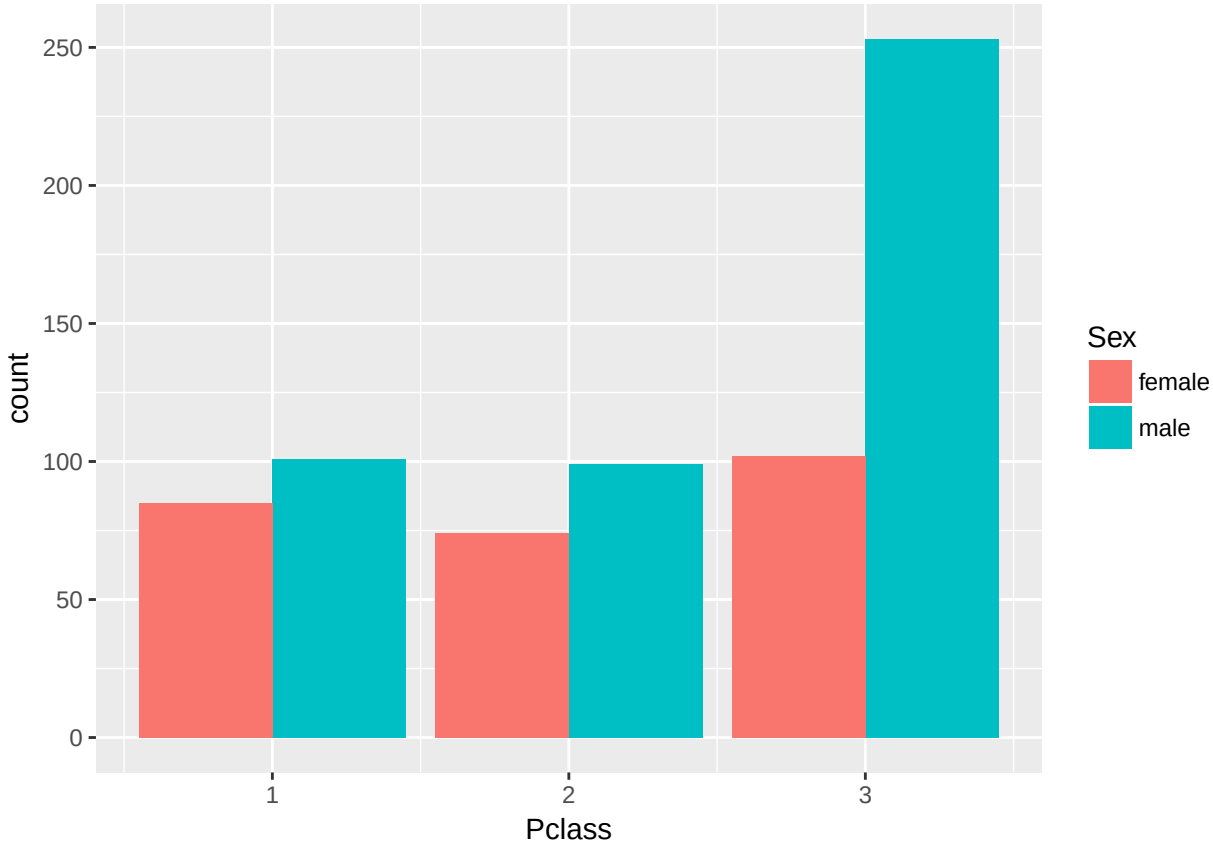
Change `geom_bar()` to `geom_point()` and set its attributes `size = 3`, `alpha = 0.5` and `position = posn.jd`.

Make sure that `Sex` is mapped onto `color` instead of `fill` to correctly color the scatter plots. (This was discussed in detail here and here).

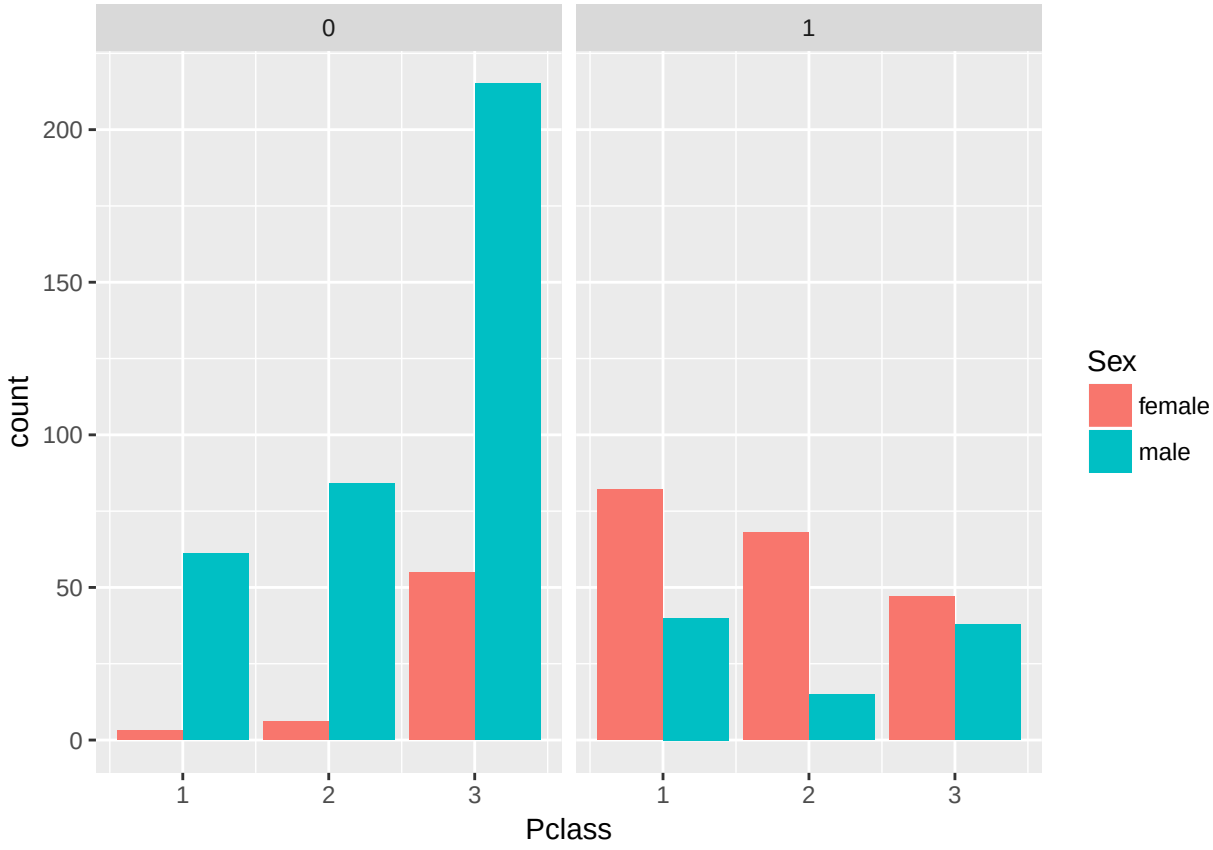
```
# titanic is avaliable in your workspace
# 1 - Check the structure of titanic
str(titanic)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    714 obs. of  4 variables:
##  $ Survived: int  0 1 1 1 0 0 0 1 1 1 ...
##  $ Pclass  : int  3 1 3 1 3 1 3 3 2 3 ...
##  $ Sex     : chr  "male" "female" "female" "female" ...
##  $ Age     : num  22 38 26 35 35 54 2 27 14 4 ...
##  - attr(*, "na.action")=Class 'omit'  Named int [1:177] 6 18 20 27 29 30 32 33 37 43 ...
##   .. ..- attr(*, "names")= chr [1:177] "6" "18" "20" "27" ...
```

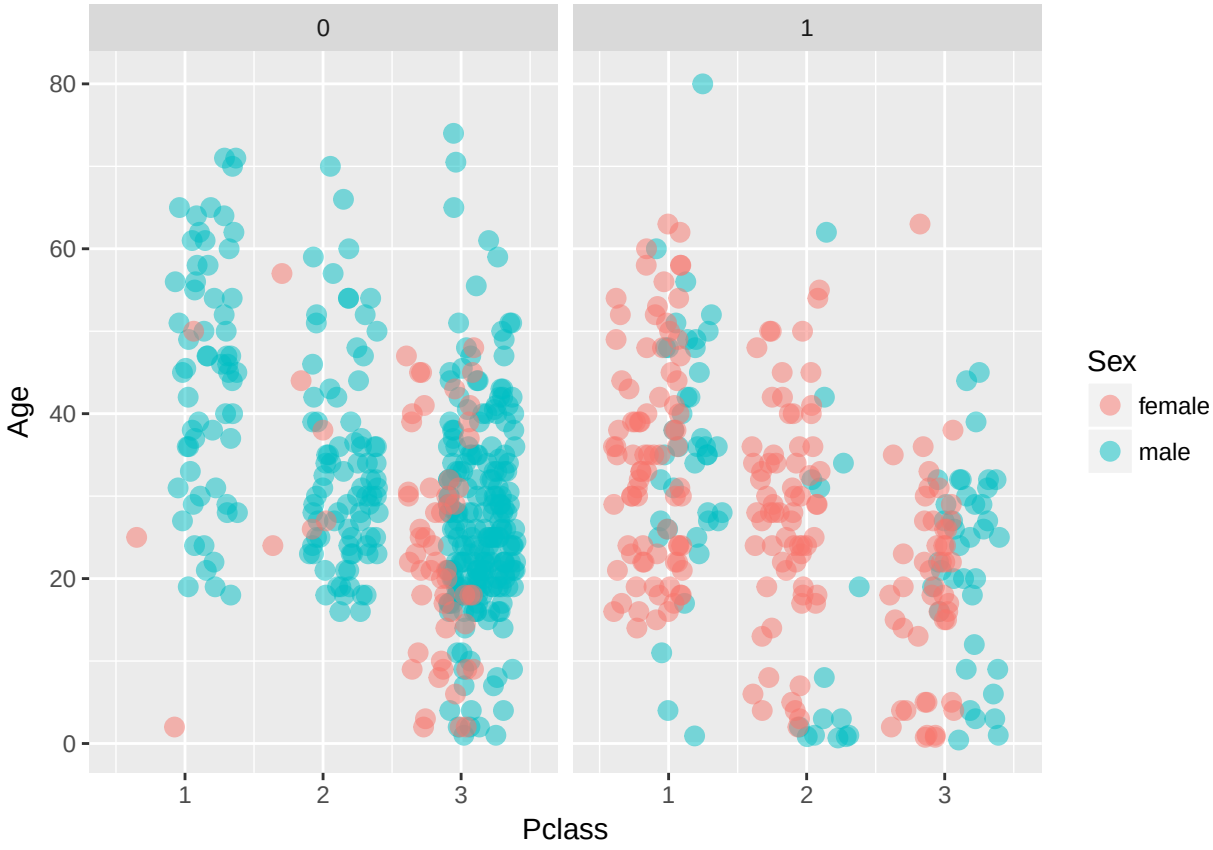
```
# 2 - Use ggplot() for the first instruction
ggplot(titanic, aes(x = Pclass, fill = Sex)) +
  geom_bar(position = "dodge")
```




```
# 3 - Plot 2, add facet_grid() layer
ggplot(titanic, aes(x = Pclass, fill = Sex)) +
  geom_bar(position = "dodge") +
  facet_grid(. ~ Survived)
```




```
# 4 - Define an object for position jitterdodge, to use below
posn.jd <- position_jitterdodge(0.5, 0, 0.6)

# 5 - Plot 3, but use the position object from instruction 4
ggplot(titanic, aes(x = Pclass, y = Age, color = Sex)) +
  geom_point(position = posn.jd, size = 3, alpha = 0.5) +
  facet_grid(. ~ Survived)
```


0
1
80
60
40
20
0
Age
1
2
3
Pclass
Sex
female
male