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.

```
library(ggplot2)
library(tidyr)
```

Required data for this session

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

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.

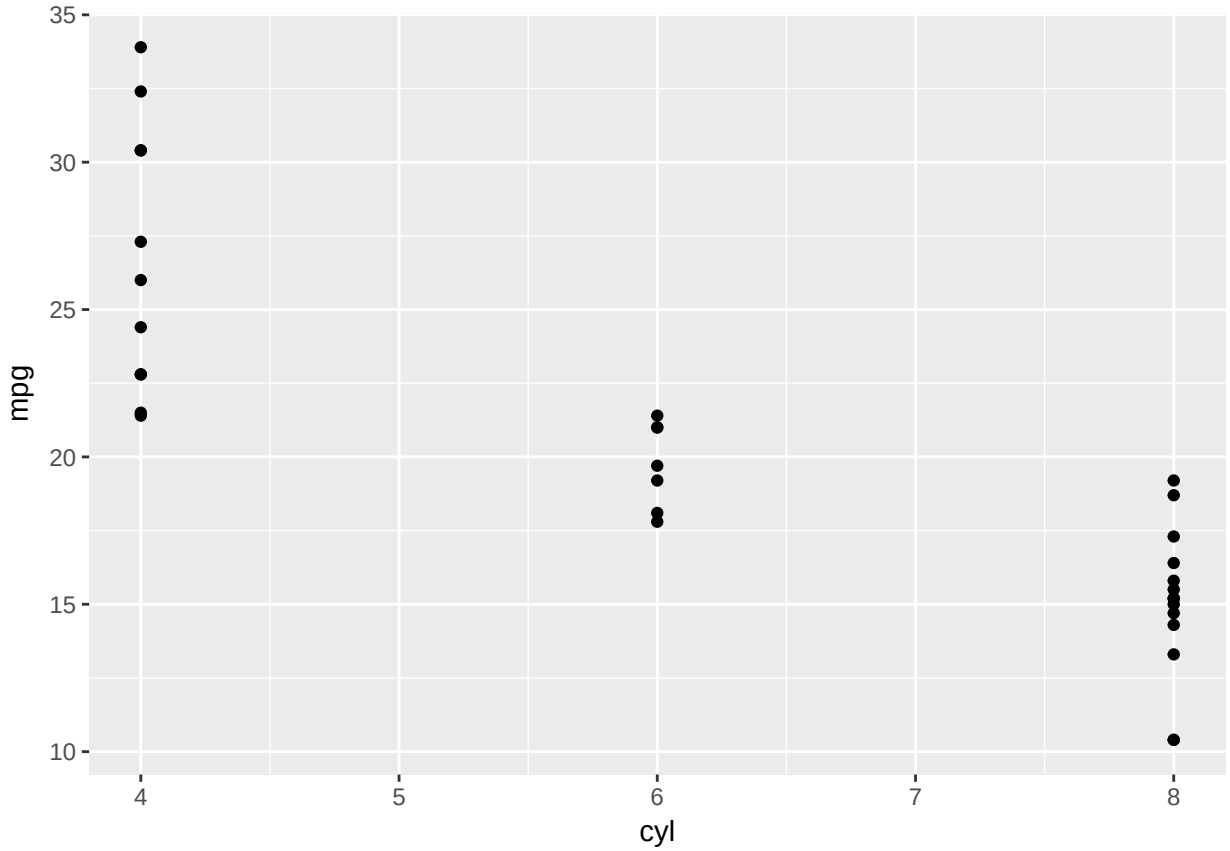
Use `str()` to explore the structure of the `mtcars` dataset.

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

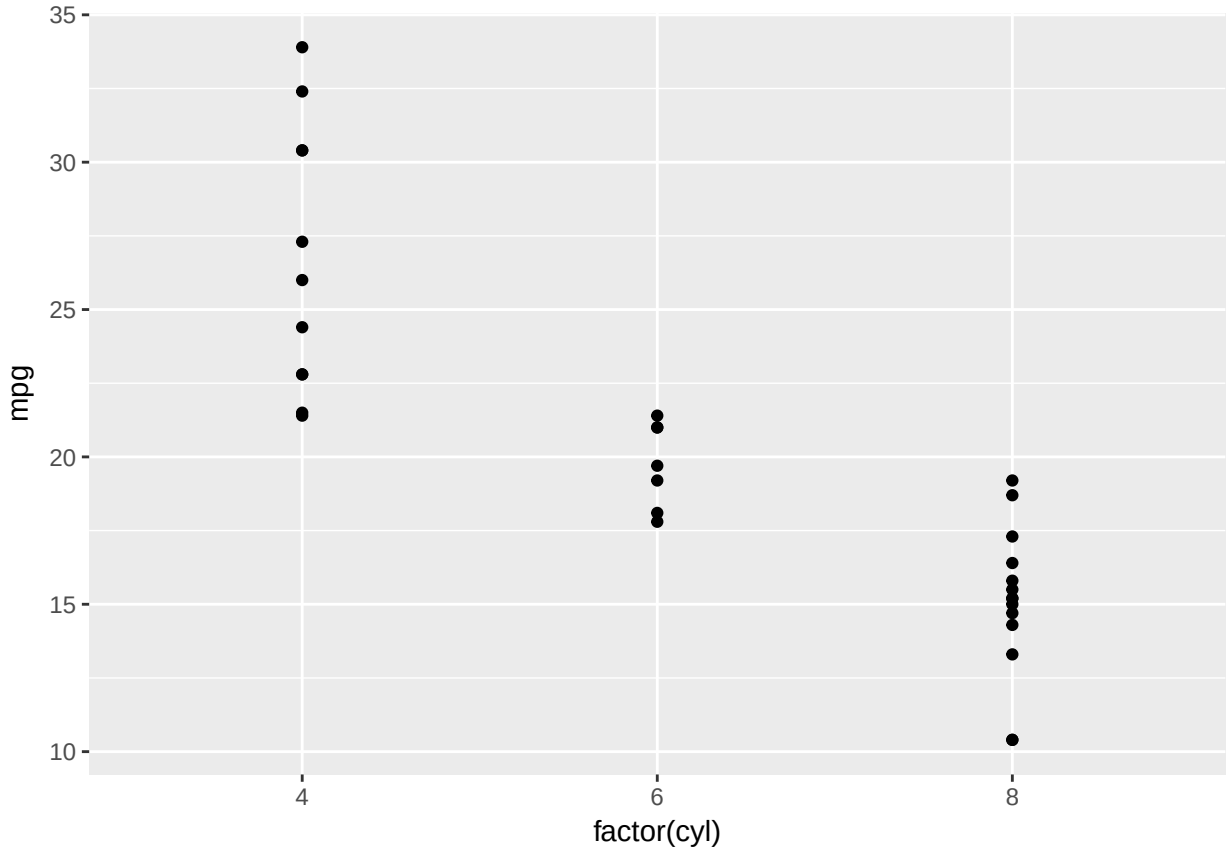
The plot from the previous exercise wasn't really satisfying. Although `cyl` (the number of cylinders) is categorical, it is classified as numeric in `mtcars`. You'll have to explicitly tell `ggplot2` that `cyl`is a categorical variable.

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!

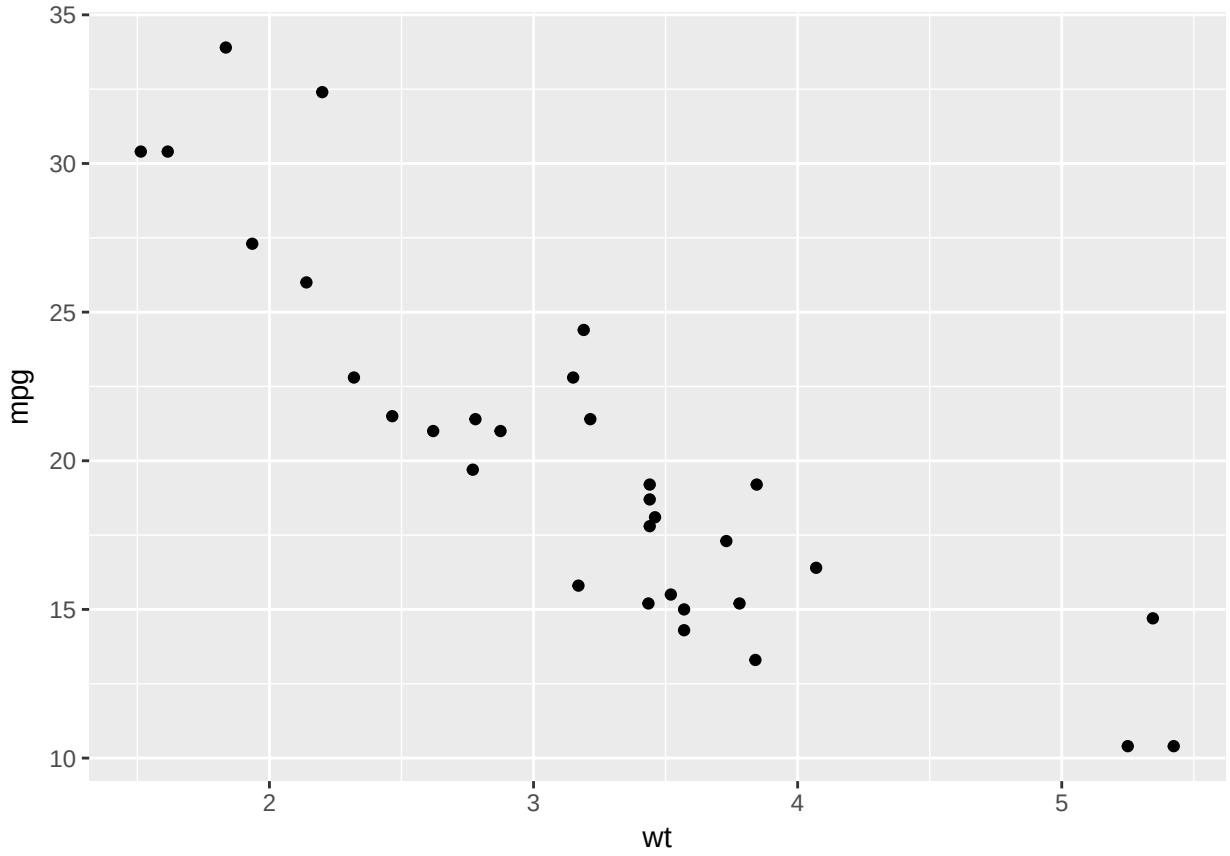
INSTRUCTIONS

`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.

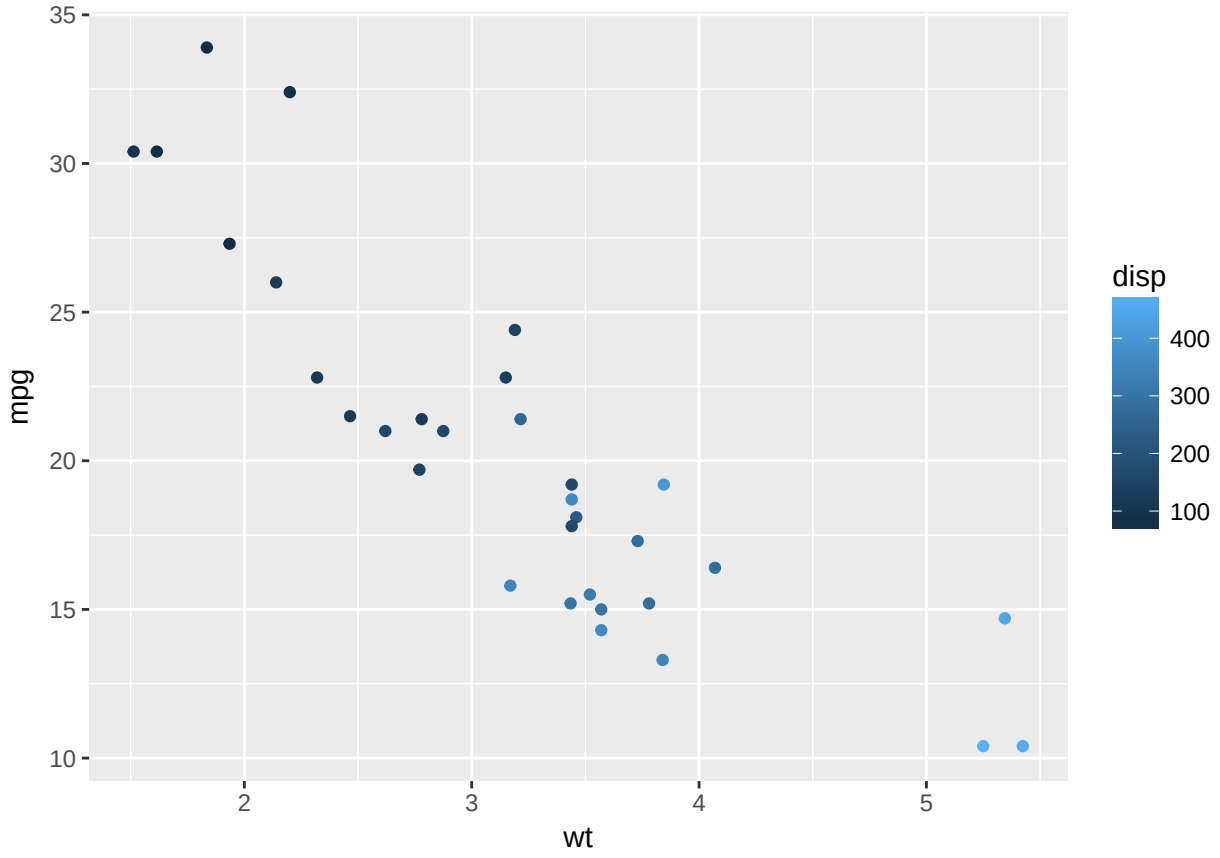
In the second call of `ggplot()` change the `color` argument in `aes()` (which stands for aesthetics). The color should be dependent on the displacement of the car engine, found in `disp`.

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`.

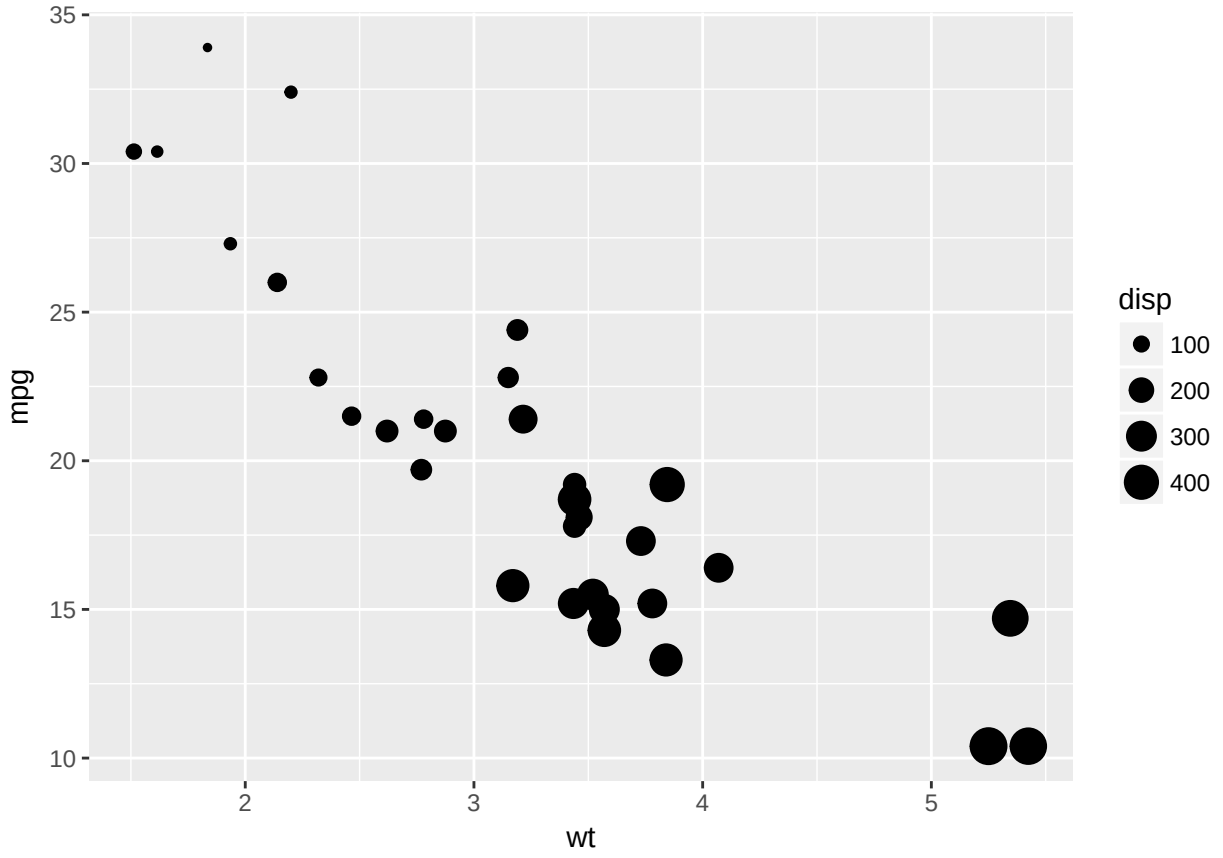
```
# A scatter plot has been made for you
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```




```
# 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.

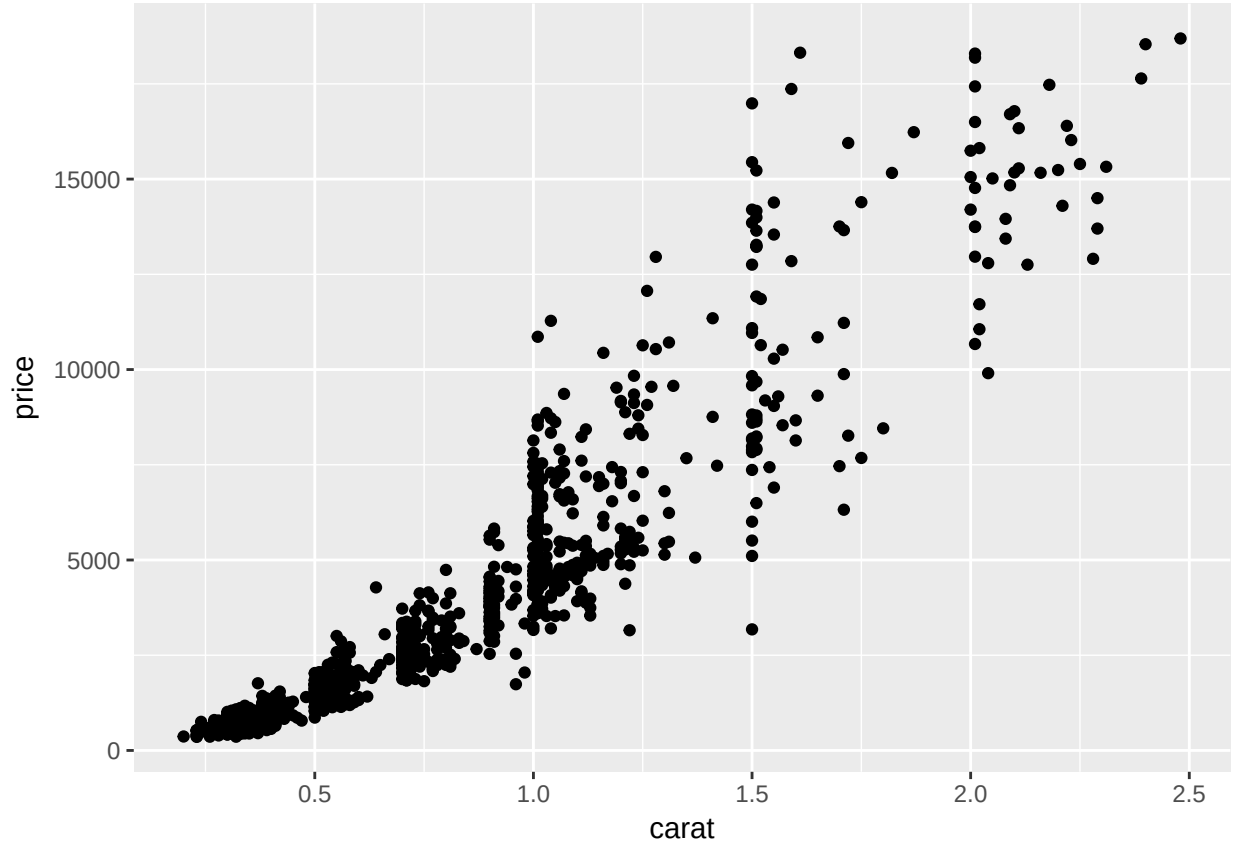
Use the `+` operator to add `geom_point()` to the first `ggplot()` command. This will tell `ggplot2` to draw points on the plot.

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)
```

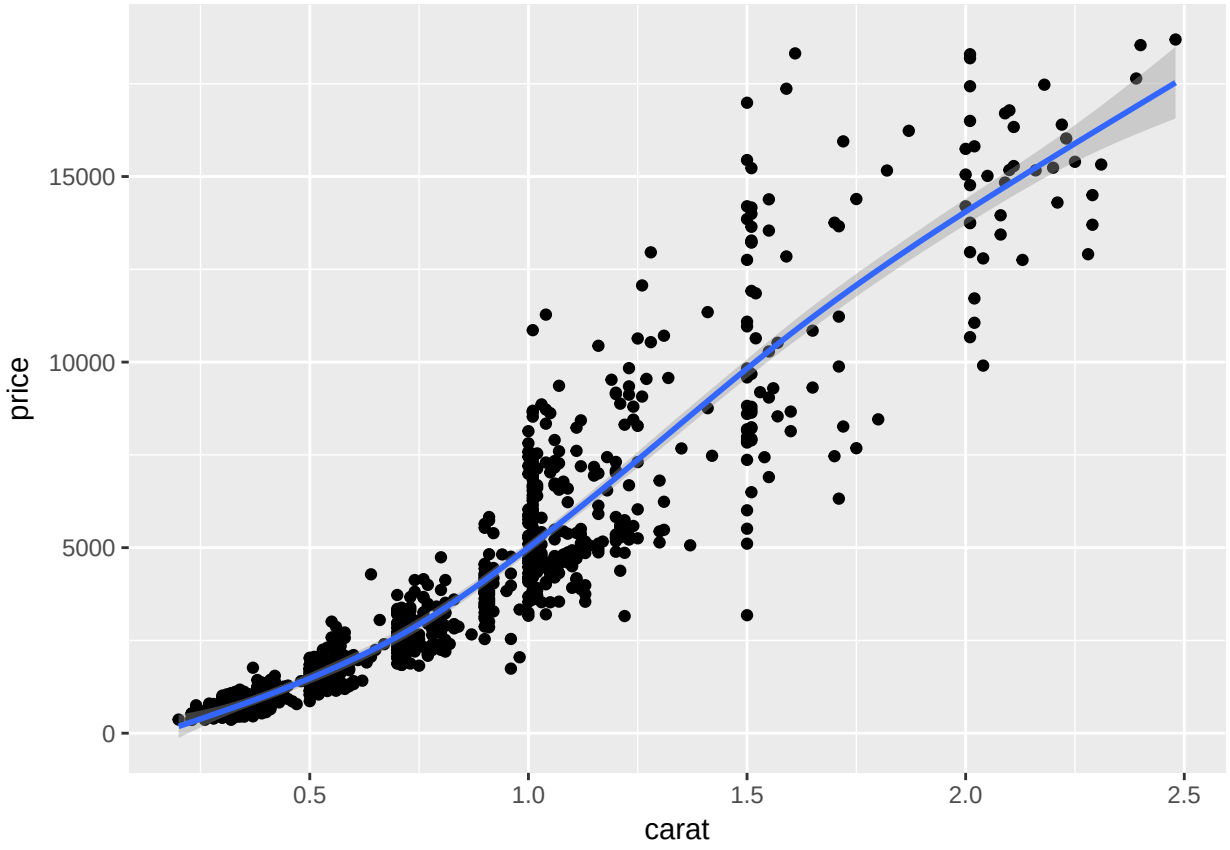
```
## Classes 'tbl_df', 'tbl' and 'data.frame':    1000 obs. of  10 variables:
##  $ carat  : num  0.31 1.5 0.9 1.01 0.33 1.08 1.07 0.33 0.44 1 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 2 4 5 3 5 4 4 5 4 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 4 4 5 3 1 4 4 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.

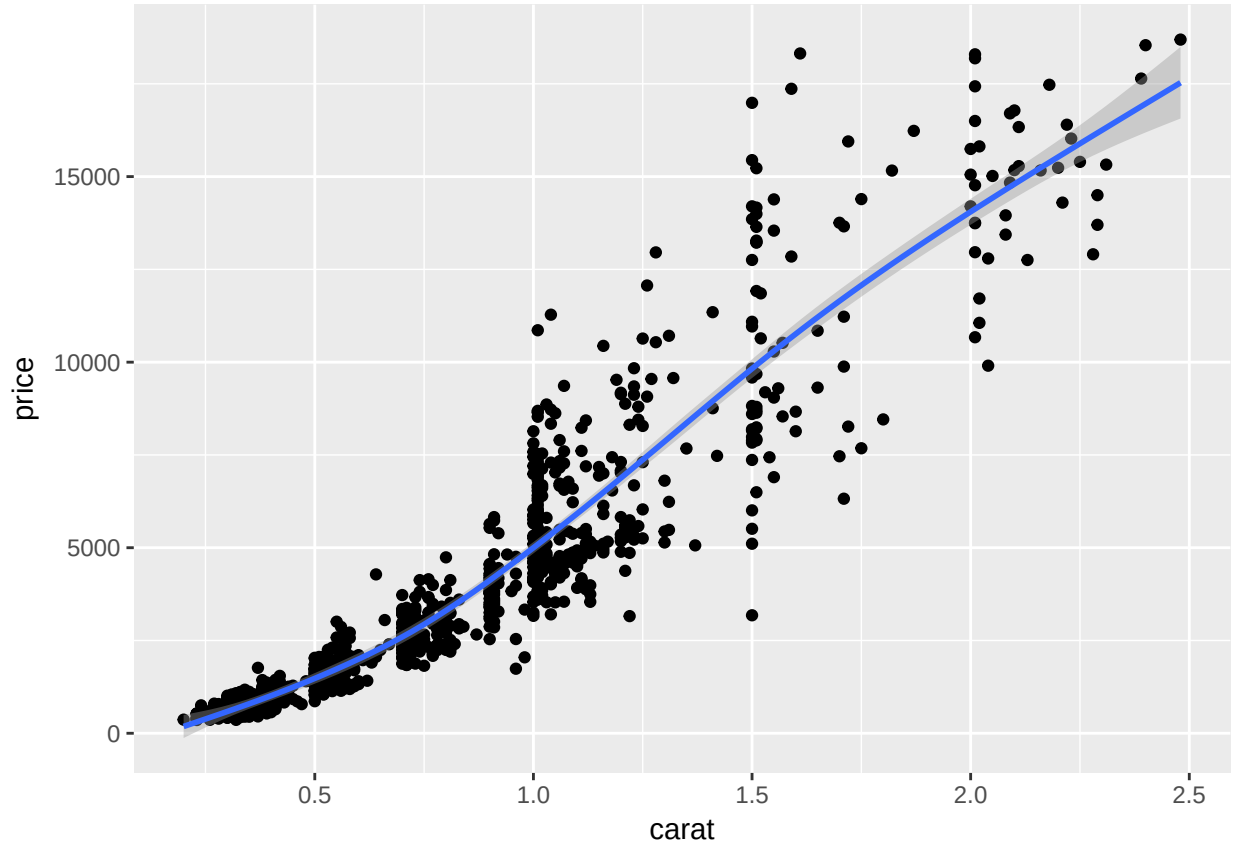
Plot 3 - Show only the smooth line, but color according to clarity by placing the argument `color = clarity` in the `aes()` function of your `ggplot()` call.

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.

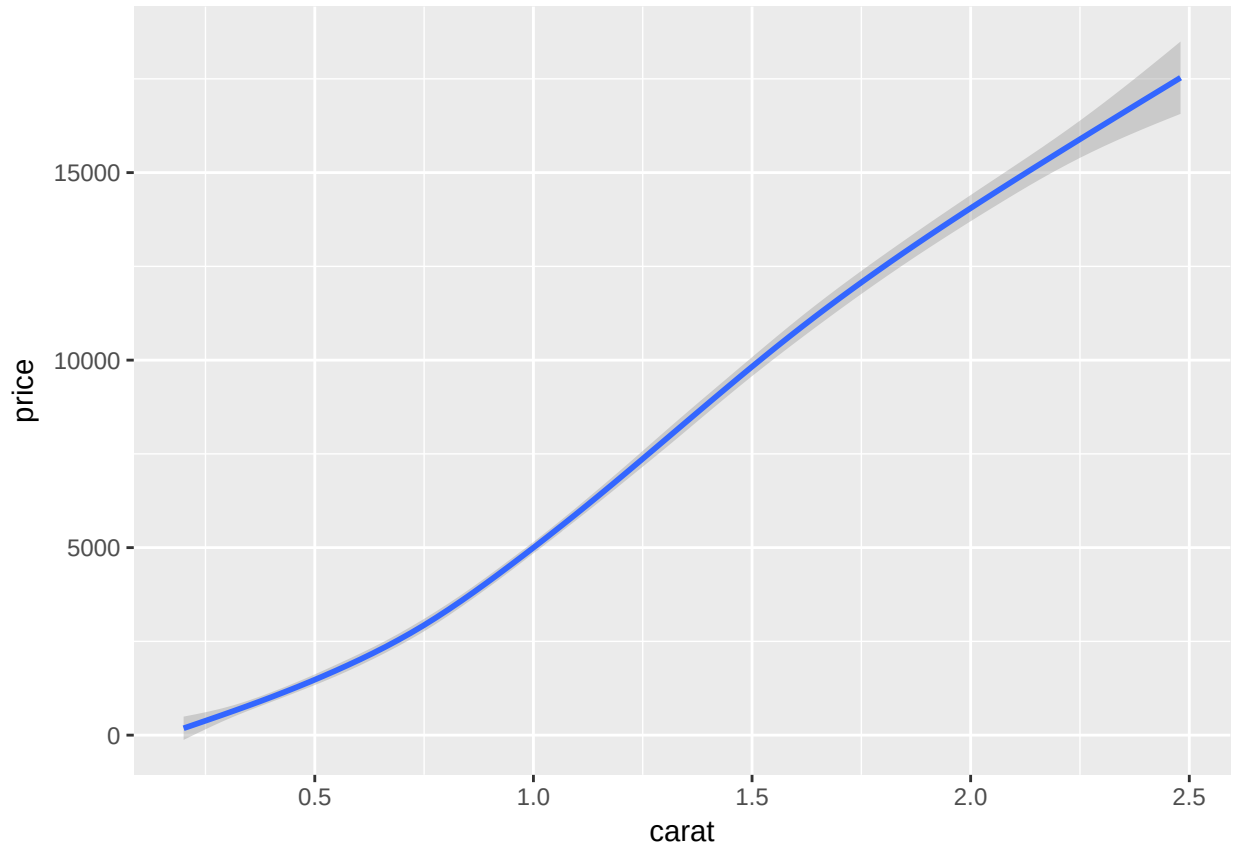
```
# 1 - The plot you created in the previous exercise
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point() +
  geom_smooth()
```

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



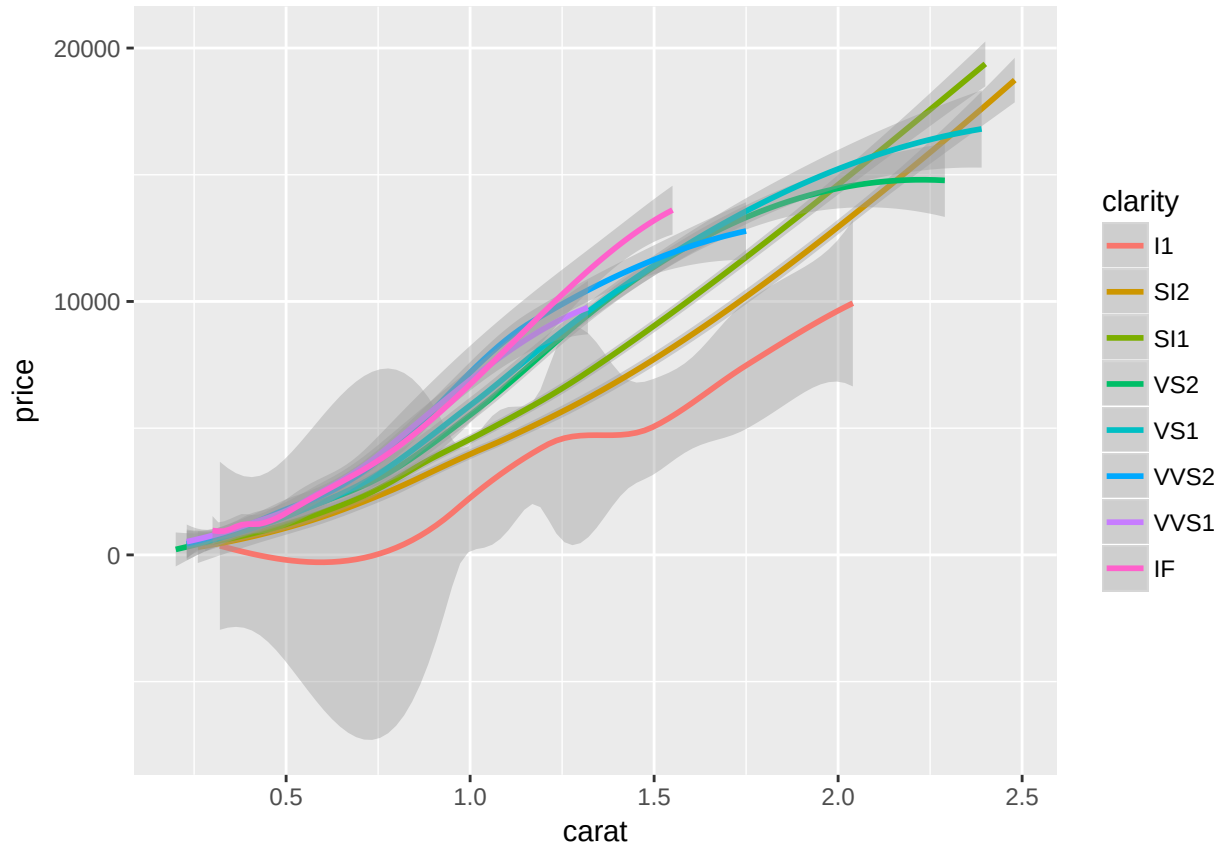

```
# 2 - Copy the above command but show only the smooth line
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_smooth()
```

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

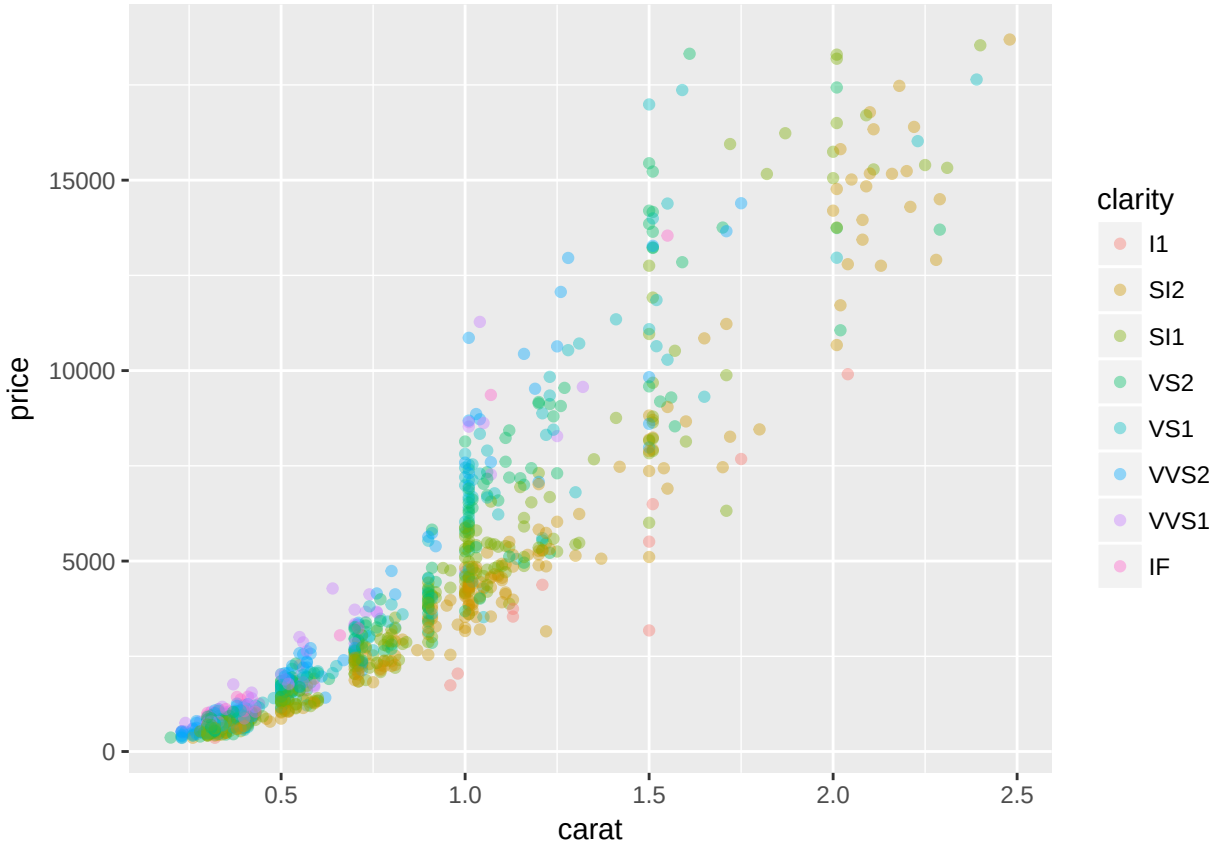



```
# 3 - Copy the above command and assign the correct value to col in aes()
ggplot(diamonds, aes(x = carat, y = price, color = clarity)) +
  geom_smooth()
```

```
## `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.

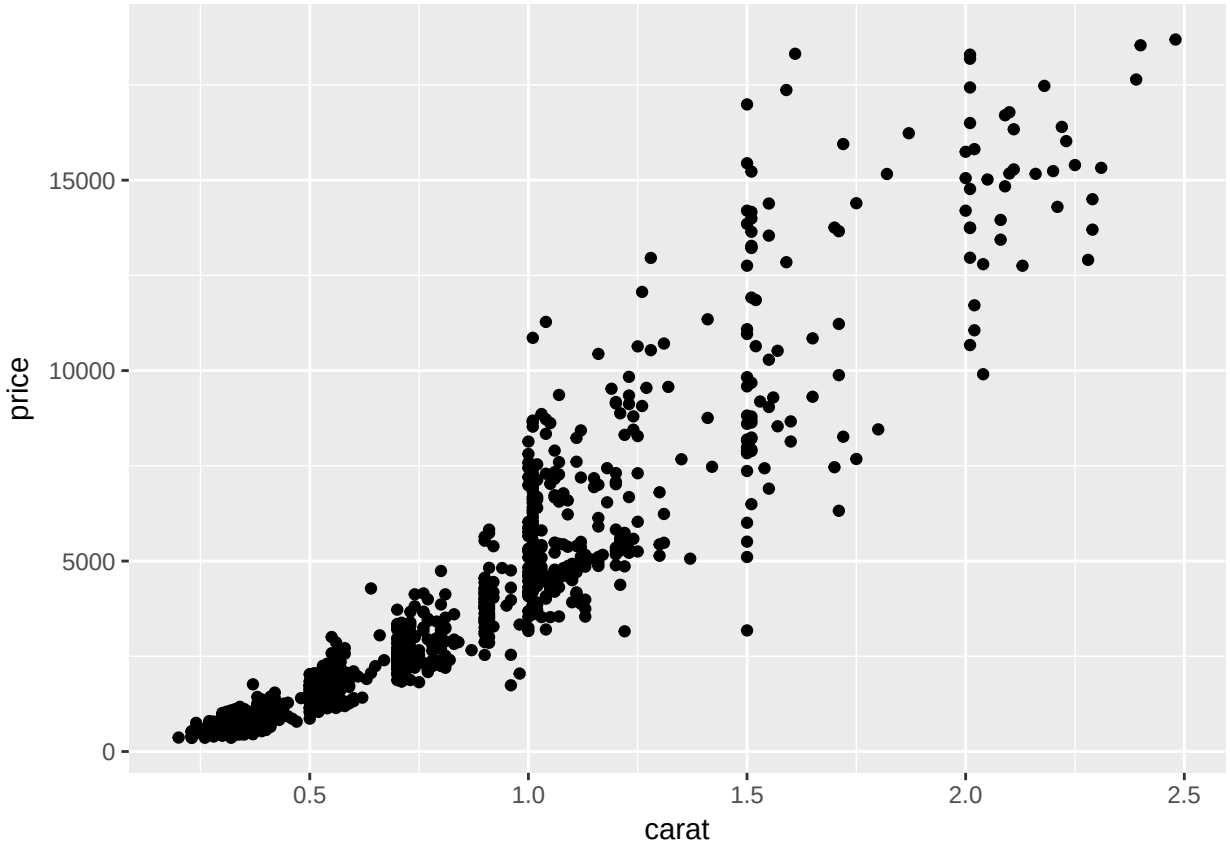
INSTRUCTIONS

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`.

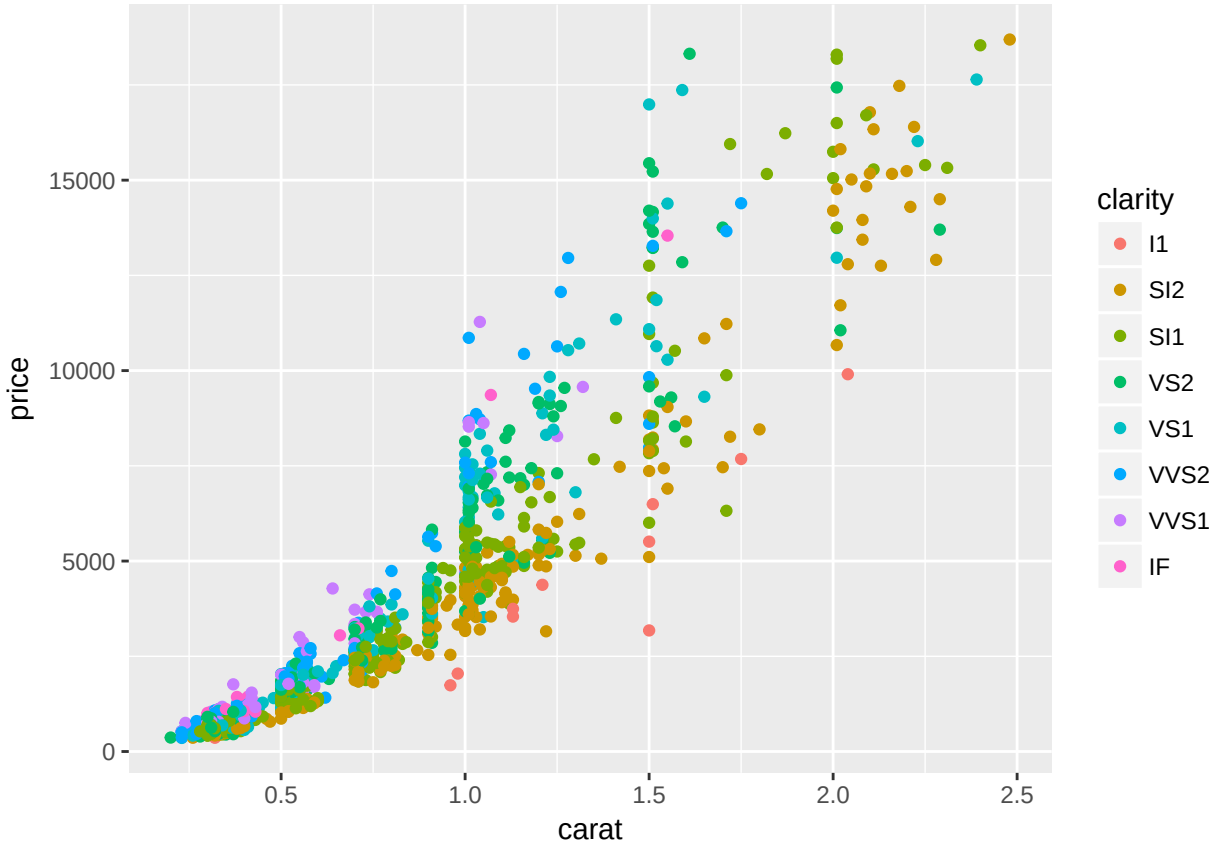
Using `+`, add a `geom_point()` layer (with no arguments), to the `dia_plot` object. This can be in a single or multiple lines.

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.

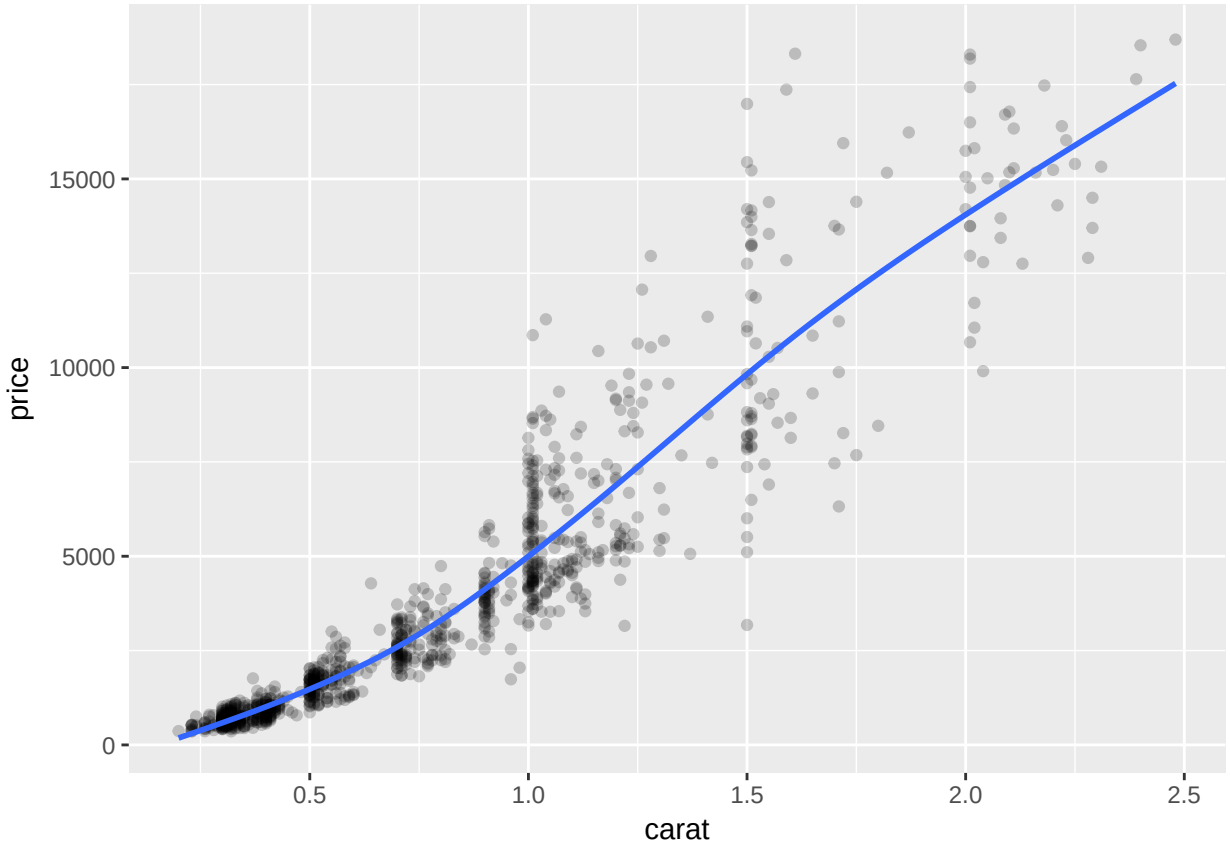
2 - Update `dia_plot` so that it contains all the functions to make a scatter plot by using `geom_point()` for the geom layer. Set `alpha = 0.2`.

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.

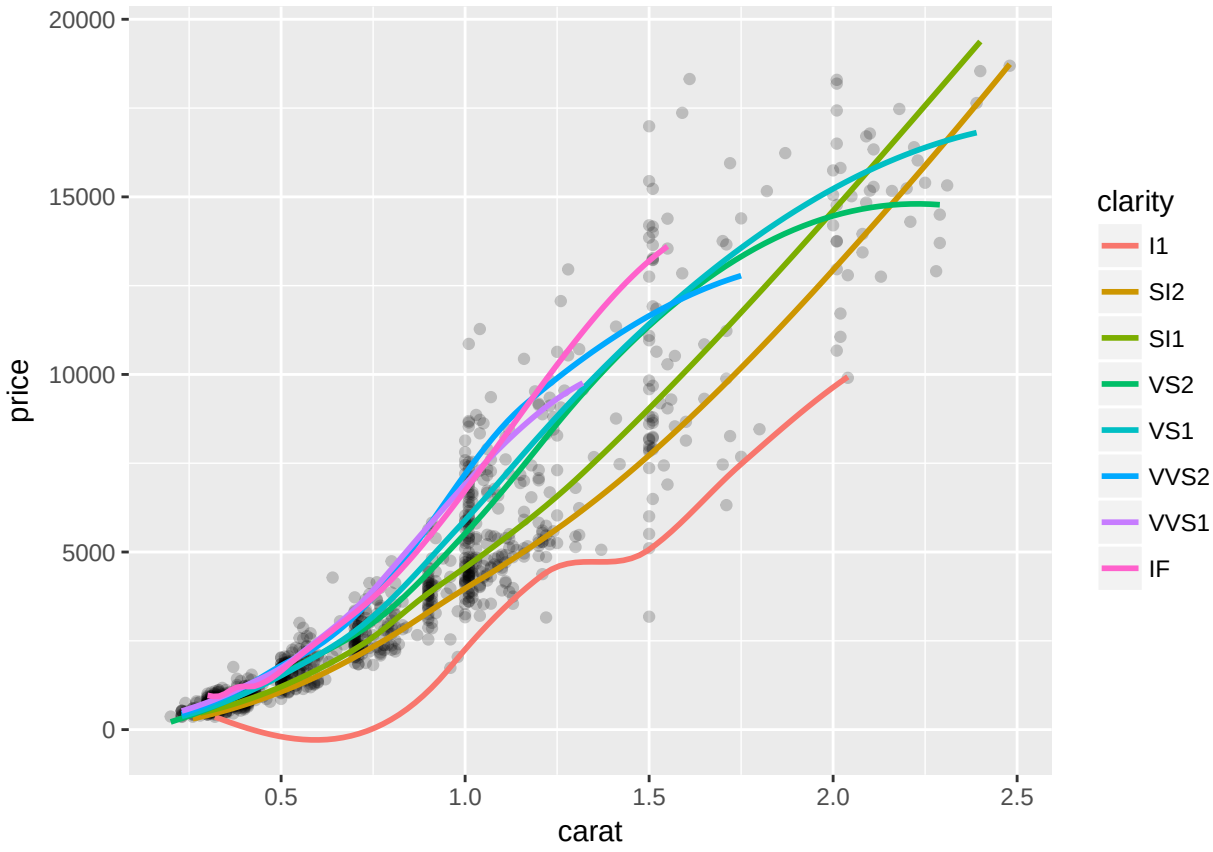
```
# 1 - The dia_plot object has been created for you
dia_plot <- ggplot(diamonds, aes(x = carat, y = price))

# 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.

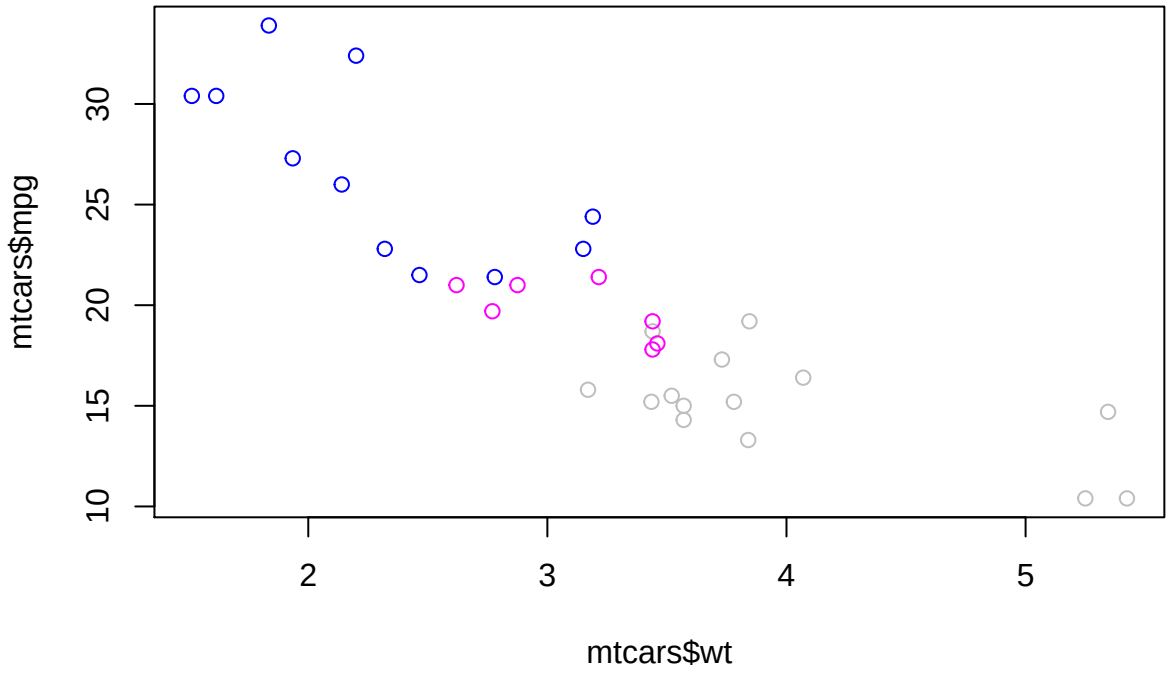
INSTRUCTIONS

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.

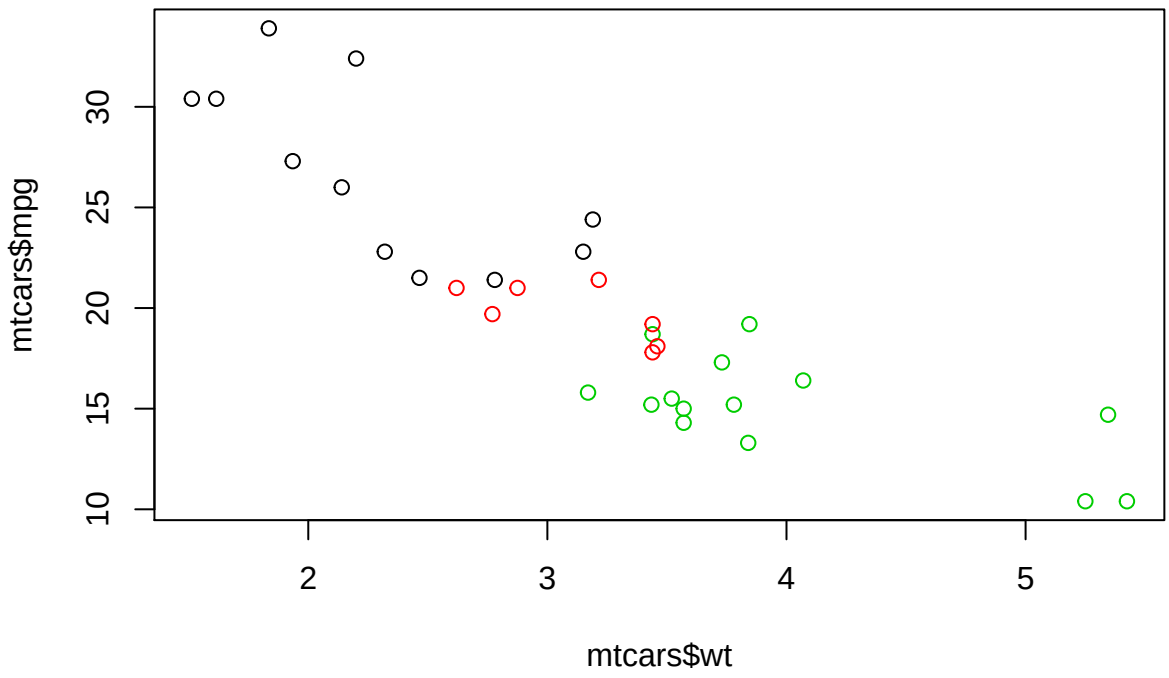
Add a new column, `fcyl`, to the mtcars data frame. This should be `cyl` converted to a factor.

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`.

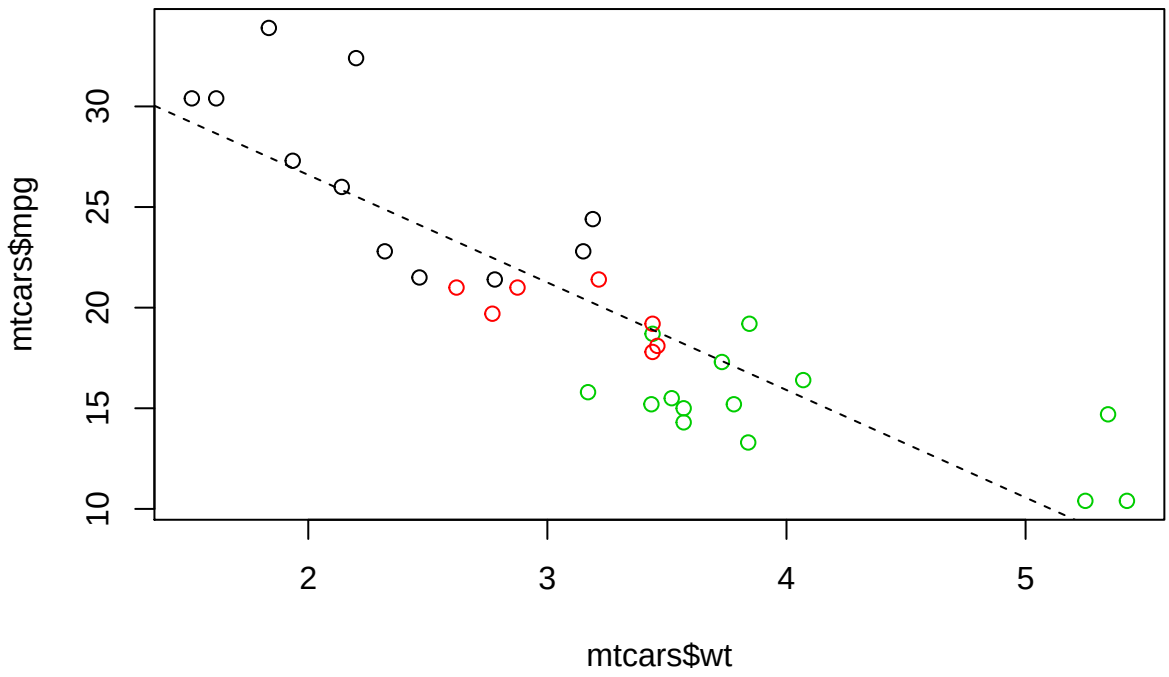
Draw the linear model on the scatterplot.

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)
```



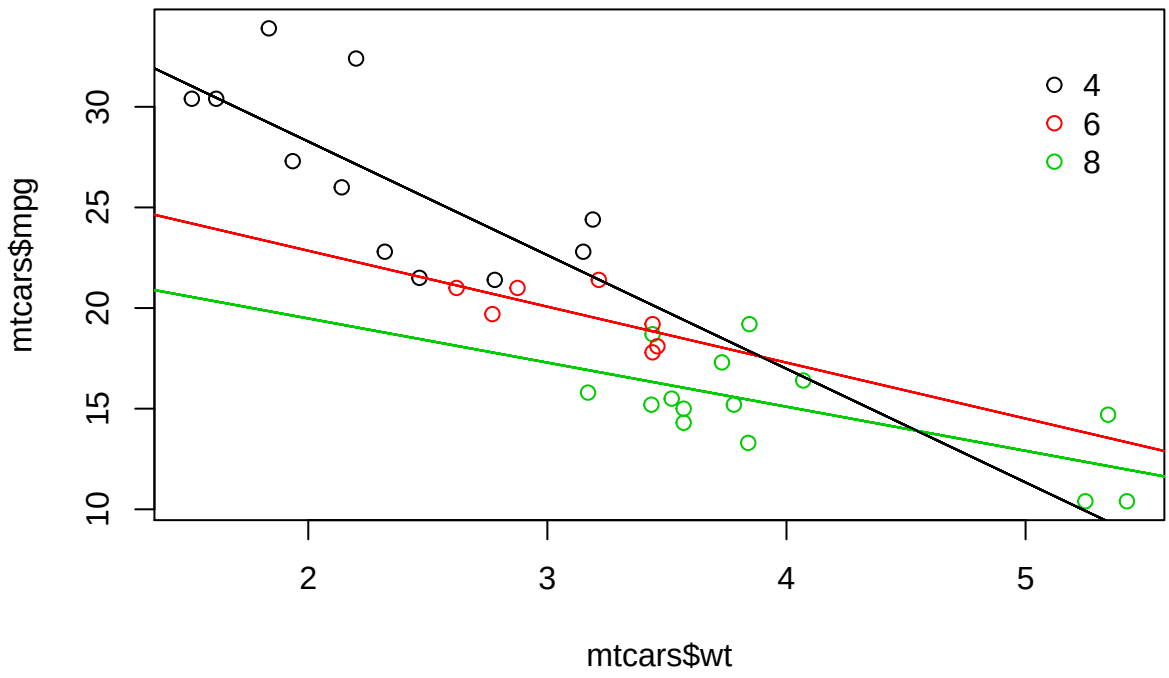

```
# Plot each subset efficiently with lapply
# You don't have to edit this code
plot(mtcars$wt, mtcars$mpg, col = mtcars$cyl)
lapply(mtcars$cyl, function(x) {
  abline(lm(mpg ~ wt, mtcars, subset = (cyl == x)), col = x)
  })
```

```
## [[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()`.

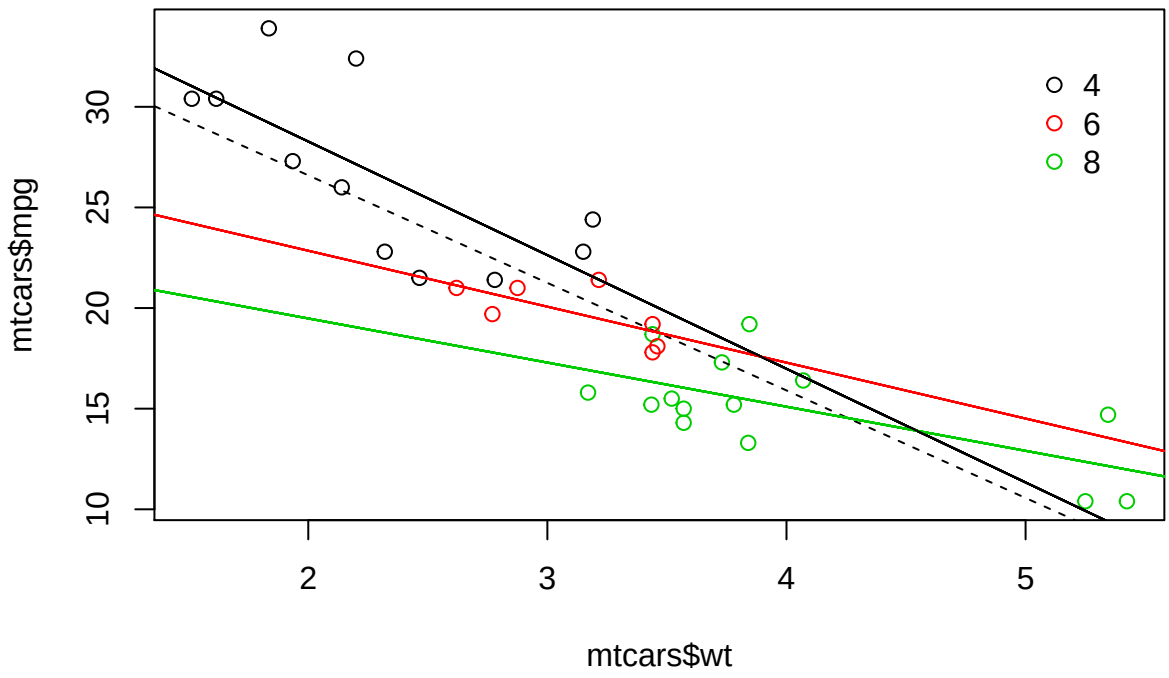
Note: the group aesthetic will tell `ggplot()` to draw a single linear model through all the points.

```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)
  })
```

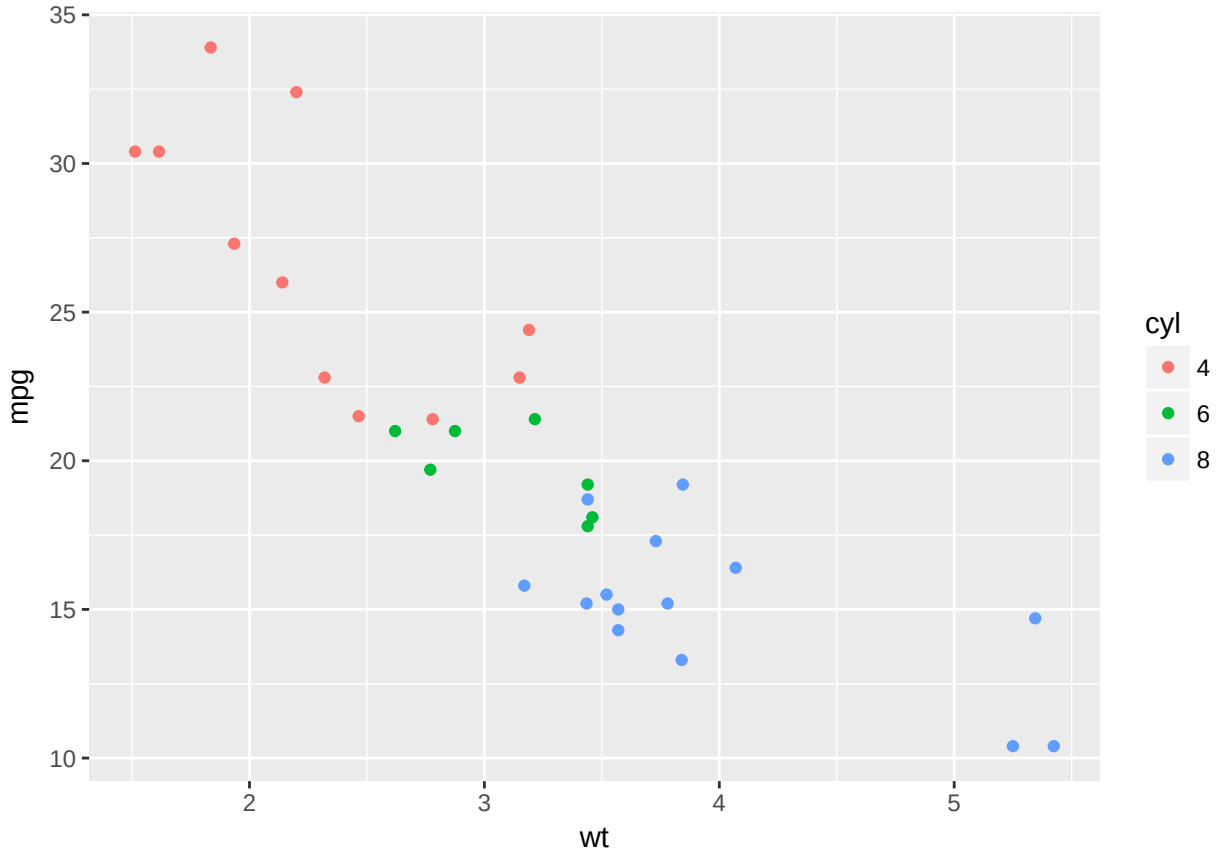
```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
##
## [[7]]
## NULL
##
## [[8]]
## NULL
##
## [[9]]
## NULL
##
## [[10]]
## NULL
##
## [[11]]
## NULL
##
## [[12]]
## NULL
##
## [[13]]
## NULL
##
## [[14]]
## NULL
##
## [[15]]
##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

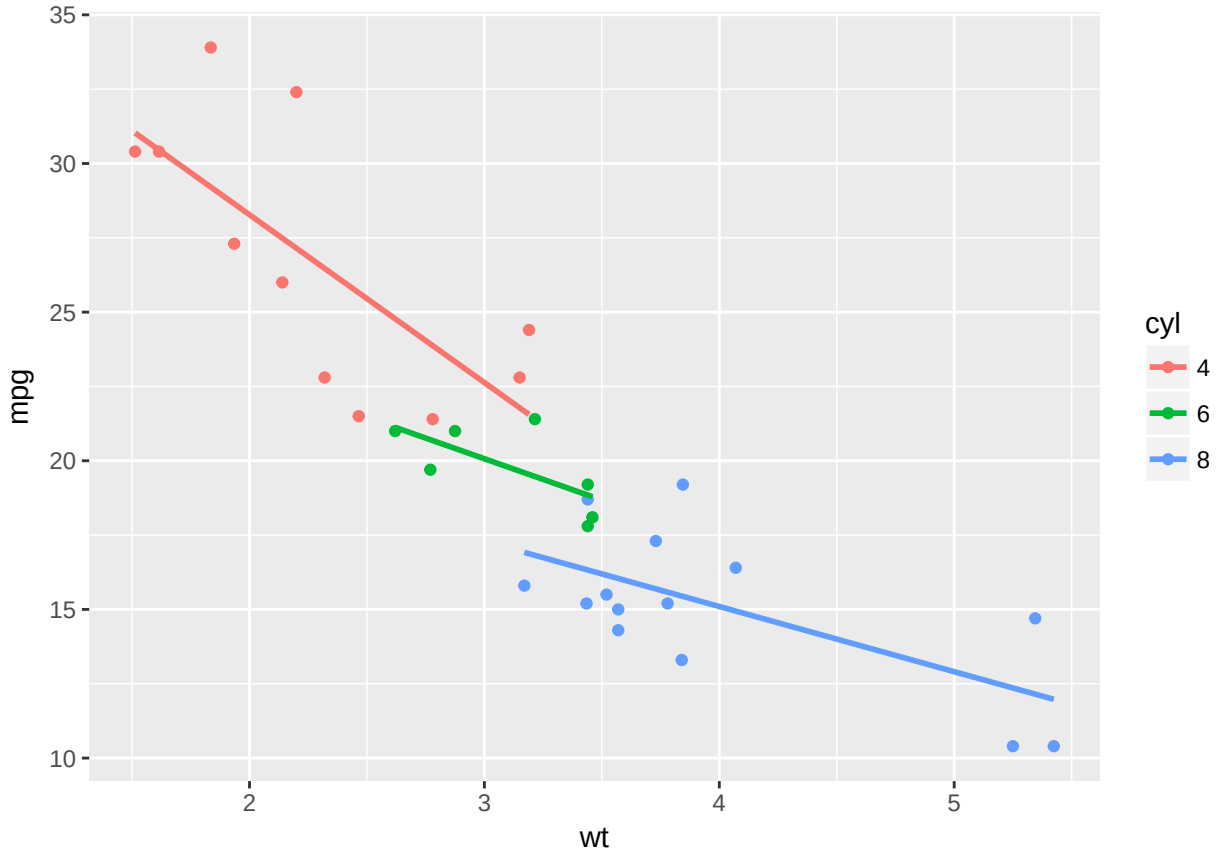
```
legend(x = 5, y = 33, legend = levels(mtcars$cyl),
       col = 1:3, pch = 1, bty = "n")
```



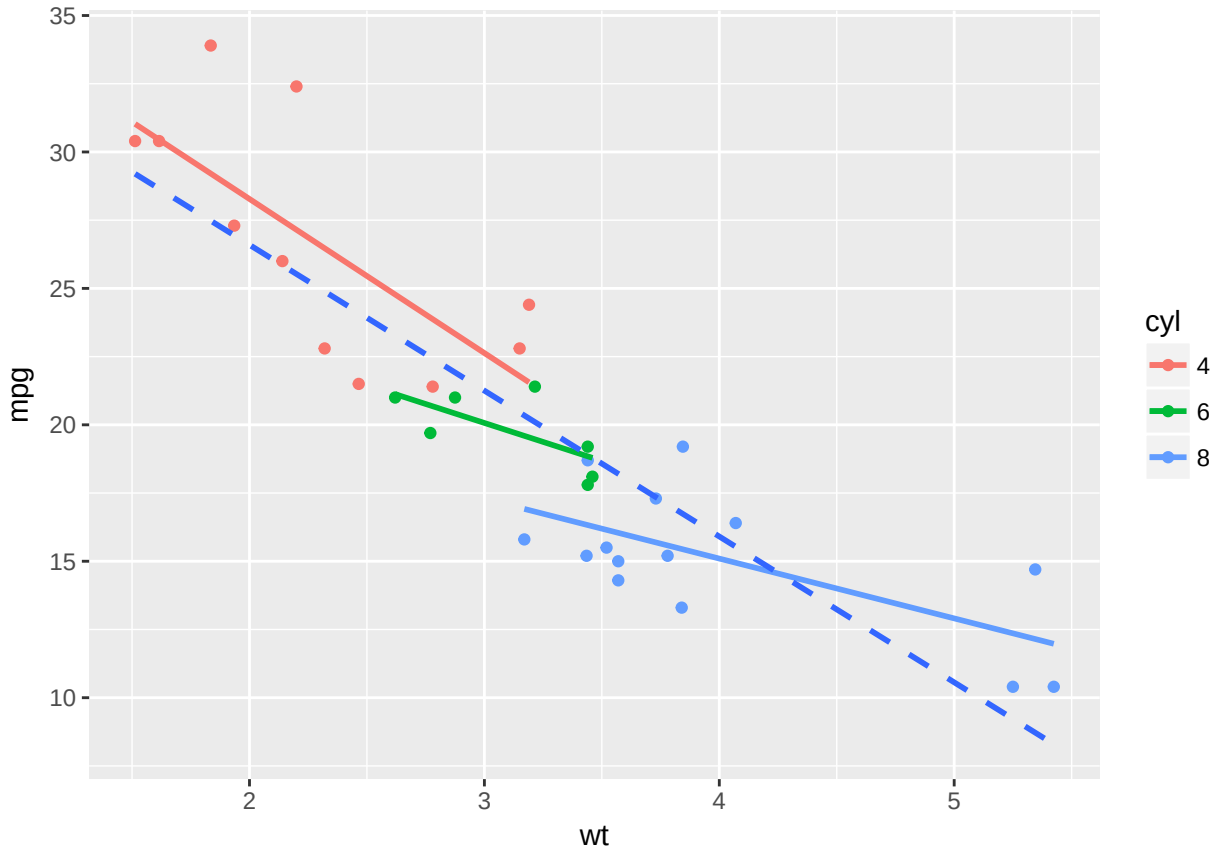

```
# Plot 1: add geom_point() to this command to create a scatter plot
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl)) +
  geom_point()  # Fill in using instructions Plot 1
```




```
# 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.

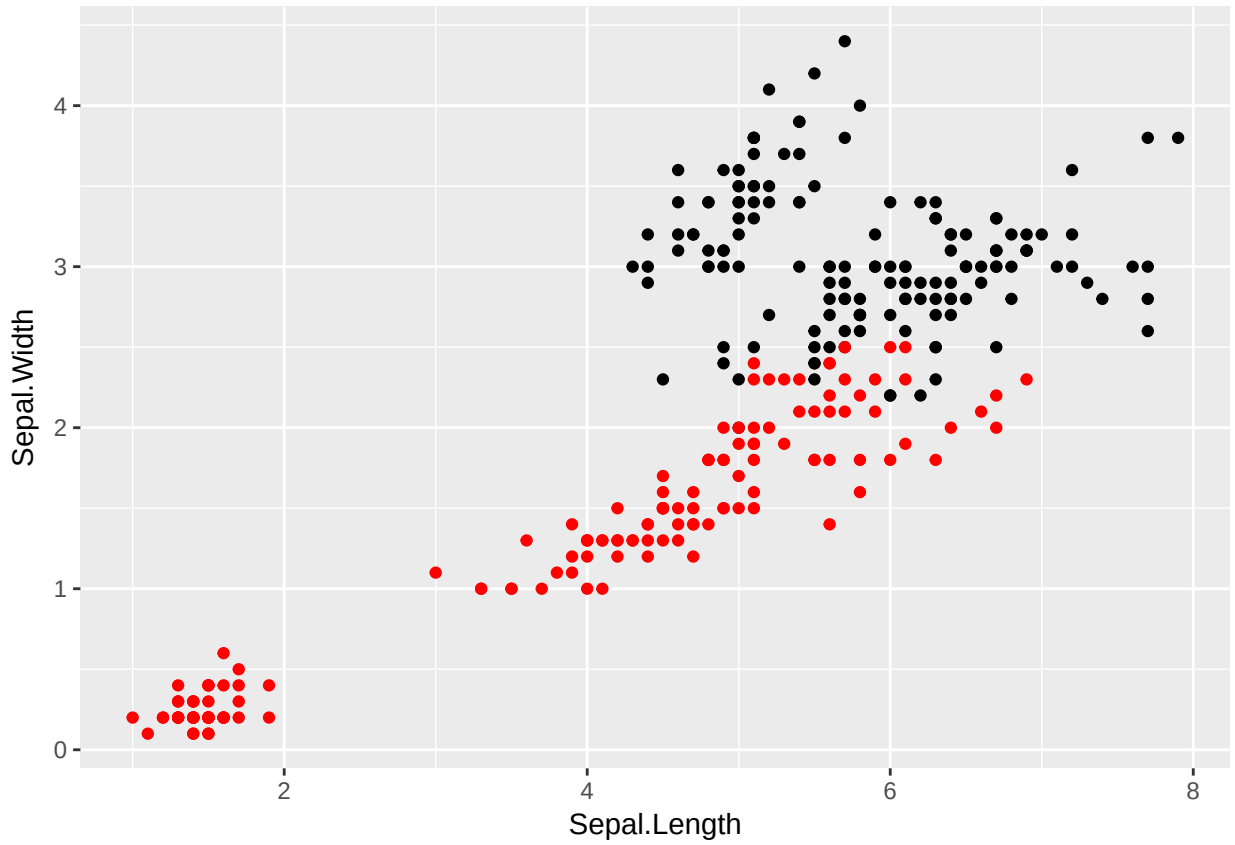
Options 1, 2, and 3.

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

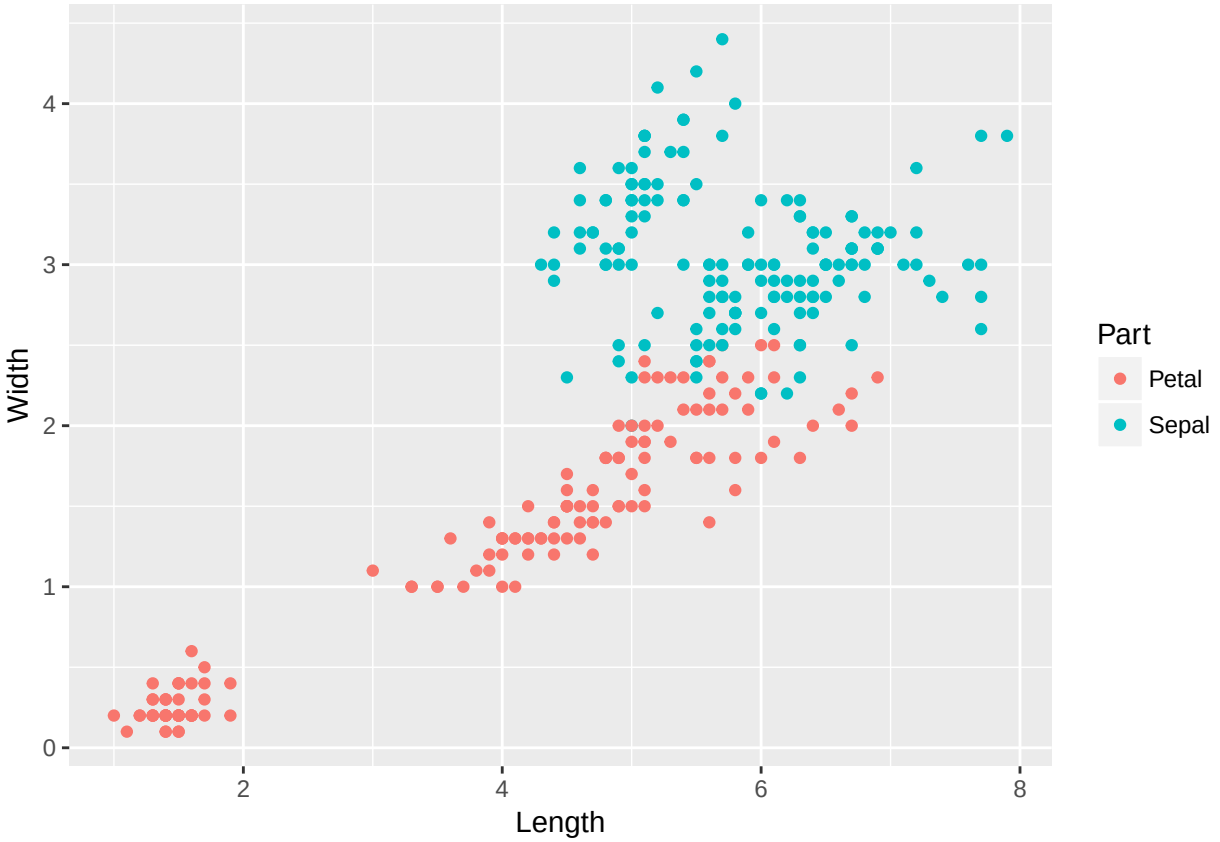
Plotting the ggplot2 way

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 ...
```

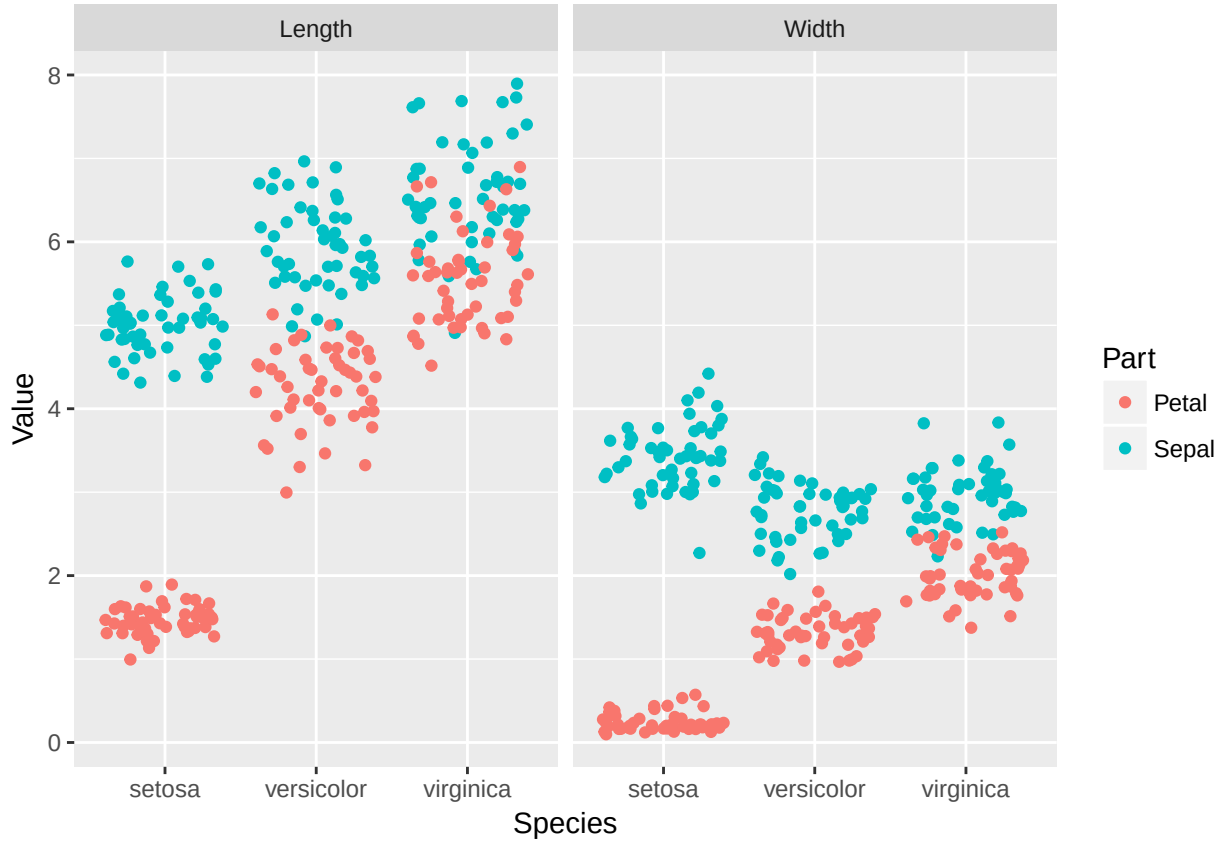
```
str(iris.wide)
```

```
## '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
```

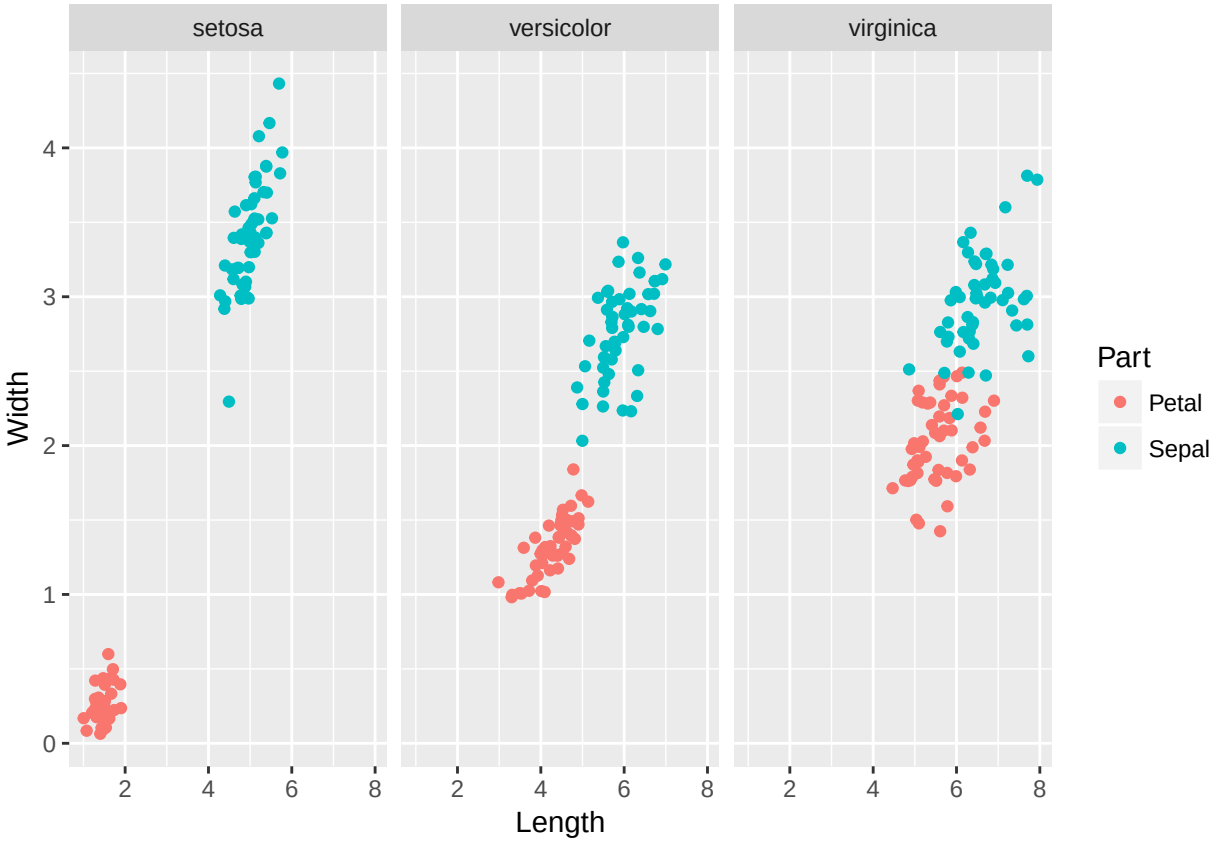
```
head(iris.wide)
```

```
##   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

All about aesthetics, part 2

All about aesthetics, part 3

All about attributes, part 1

All about attributes, part 2

Going all out

Aesthetics for categorical and continuous variables

Position

Setting a dummy aesthetic

Overplotting 1 - Point shape and transparency

Overplotting 2 - alpha with large datasets

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)

Scatter plots and jittering (2)

Histograms

Position

Overlapping bar plots

Icon exercise interactive

Overlapping histograms

Bar plots with color ramp, part 1

Bar plots with color ramp, part 2

Overlapping histograms (2)

Line plots

Periods of recession

Multiple time series, part 1

Multiple time series, part 2

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

Using aesthetics

Choosing geoms, part 1

Choosing geoms, part 2 - dotplot

Chicken weight

Titanic