3 Data visualisation

3.1 Introduction

> "The simple graph has brought more information to the data analyst's mind than any other device." — John Tukey

This chapter will teach you how to visualise your data using ggplot2. R has several systems for making graphs, but ggplot2 is one of the most elegant and most versatile. ggplot2 implements the **grammar of graphics**, a coherent system for describing and building graphs. With ggplot2, you can do more faster by learning one system and applying it in many places.

3.1.1 Prerequisites

To access the datasets, help pages, and functions that we will use in this chapter, load ggplot2 using the `library()` function. We'll also load tibble, which you'll learn about later. It improves the default printing of datasets.

```
library(ggplot2)
library(tibble)
```

If you run this code and get the error message "there is no package called 'ggplot2'", you'll need to first install it, then run `library()` once again.

```
install.packages("ggplot2")
library(ggplot2)
```

You only need to install a package once, but you need to reload it every time you start a new session.

If we need to be explicit about where a function (or dataset) comes from, we'll use the special form `package::function()`. For example, `ggplot2::ggplot()` tells you explicitly that we're using the `ggplot()` function from the ggplot2 package.

3.2 A graphing template

Let's use our first graph to answer a question: Do cars with big engines use more fuel than cars with small engines? You probably already have an answer, but try to make your answer precise. What does the relationship between engine size and fuel efficiency look like? Is it positive? Negative? Linear? Nonlinear?

You can test your answer with the `mpg` dataset in ggplot2, or `ggplot2::mpg`:

```
mpg
#> # A tibble: 234 x 11
#>   manufacturer model displ  year   cyl      trans   drv   cty   hwy    fl
#>          <chr> <chr> <dbl> <int> <int>      <chr> <chr> <int> <int> <chr>
#> 1         audi    a4   1.8  1999     4   auto(l5)     f    18    29     p
#> 2         audi    a4   1.8  1999     4 manual(m5)     f    21    29     p
#> 3         audi    a4   2.0  2008     4 manual(m6)     f    20    31     p
#> 4         audi    a4   2.0  2008     4   auto(av)     f    21    30     p
#> 5         audi    a4   2.8  1999     6   auto(l5)     f    16    26     p
#> 6         audi    a4   2.8  1999     6 manual(m5)     f    18    26     p
#> # ... with 228 more rows, and 1 more variables: class <chr>
```

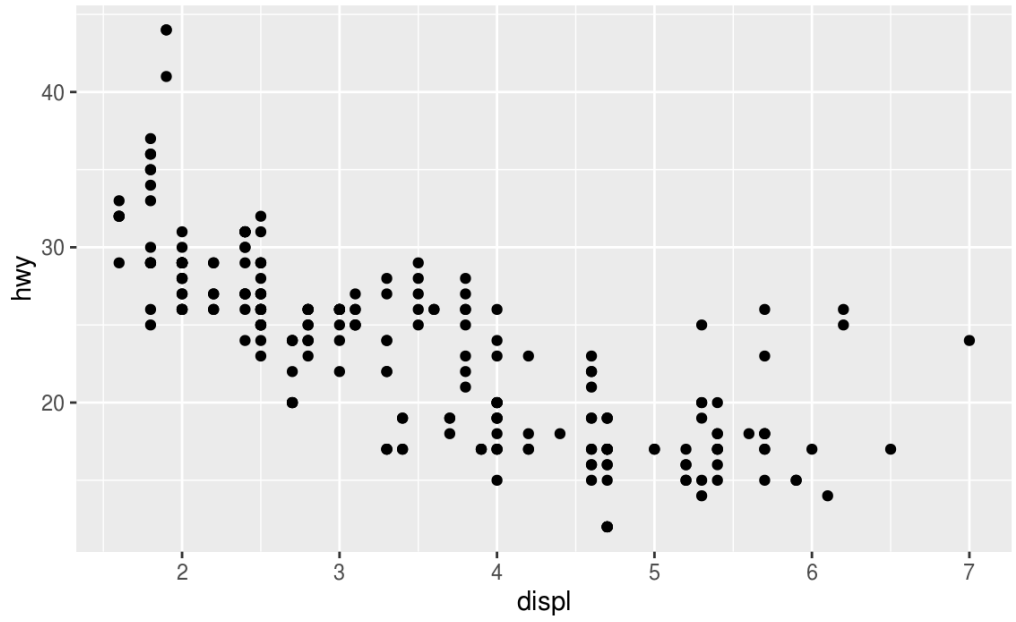
The dataset contains observations collected by the EPA on 38 models of car. Among the variables in `mpg` are:

1. `displ`, a car's engine size, in litres.
2. `hwy`, a car's fuel efficiency on the highway, in miles per gallon (mpg). A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance.

To learn more about `mpg`, open its help page by running `?mpg`.

To plot `mpg`, open an R session and run the code below. The code plots the `mpg` data by putting `displ` on the x-axis and `hwy` on the y-axis:

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



The plot shows a negative relationship between engine size (`displ`) and fuel efficiency (`hwy`). In other words, cars with big engines use more fuel. Does this confirm or refute your hypothesis about fuel efficiency and engine size?

Pay close attention to this code because it is almost a template for making plots with ggplot2.

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

With ggplot2, you begin a plot with the function `ggplot()`. `ggplot()` creates a coordinate system that you can add layers to. The first argument of `ggplot()` is the dataset to use in the graph. So `ggplot(data = mpg)` creates an empty graph, but it's not very interesting so I'm not going to show it here.

You complete your graph by adding one or more layers to `ggplot()`. The function `geom_point()` adds a layer of points to your plot, which creates a scatterplot. ggplot2 comes with many geom functions that each add a different type of layer to a plot. You'll learn a whole bunch of them through out this chapter.

Each geom function in ggplot2 takes a `mapping` argument. This defines how variables in your dataset are mapped to visual properties. You must always use `mapping()` in conjunction with `aes()`. The `x` and `y` arguments of `aes()` describe which variables to map to the x and y axes of your plot, and ggplot2 will look for those variables in your dataset, `mpg`.

Let's turn this code into a reusable template for making graphs with ggplot2. To make a graph, replace the bracketed sections in the code below with a dataset, a geom function, or a set of mappings.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

The rest of this chapter will show you how to complete and extend this template to make different types of graphs. We will begin with the `<MAPPINGS>` component.

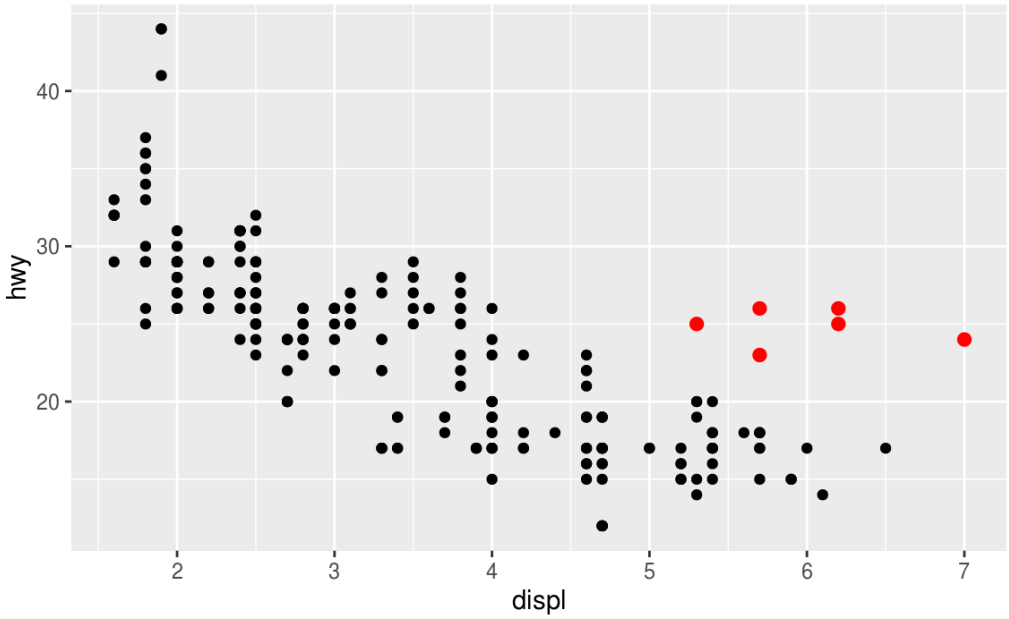
3.2.1 Exercises

1. Run `ggplot(data = mpg)` what do you see?
2. What does the `drv` variable describe? Read the help for `?mpg` to find out.
3. Make a scatterplot of `hwy` vs `cyl`.
4. What happens if you make a scatterplot of `class` vs `drv`. Why is the plot not useful?

3.3 Aesthetic mappings

> "The greatest value of a picture is when it forces us to notice what we never expected to see." — John Tukey

In the plot below, one group of points (highlighted in red) seems to fall outside of the linear trend. These cars have a higher mileage than you might expect. How can you explain these cars?



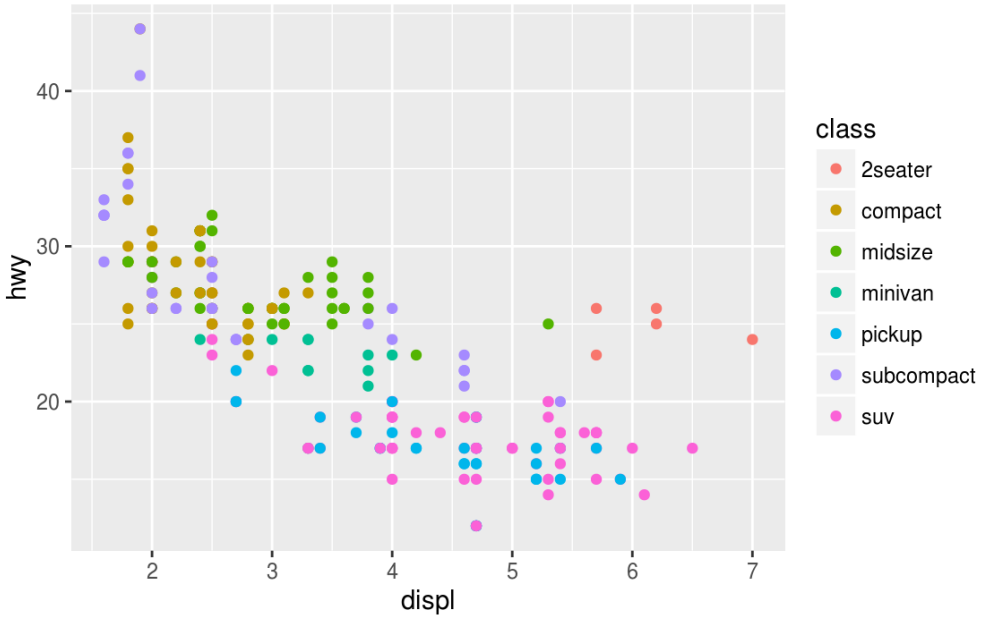
Let's hypothesize that the cars are hybrids. One way to test this hypothesis is to look at the `class` value for each car. The `class` variable of the `mpg` dataset classifies cars into groups such as compact, midsize, and SUV. If the outlying points are hybrids, they should be

classified as compact cars or, perhaps, subcompact cars (keep in mind that this data was collected before hybrid trucks and SUVs became popular).

You can add a third variable, like `class`, to a two dimensional scatterplot by mapping it to an **aesthetic**. An aesthetic is a visual property of the objects in your plot. Aesthetics include things like the size, the shape, or the color of your points. You can display a point (like the one below) in different ways by changing the values of its aesthetic properties. Since we already use the word "value" to describe data, let's use the word "level" to describe aesthetic properties. Here we change the levels of a point's size, shape, and color to make the point small, triangular, or blue:

You can convey information about your data by mapping the aesthetics in your plot to the variables in your dataset. For example, you can map the colors of your points to the `class` variable to reveal the class of each car.

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



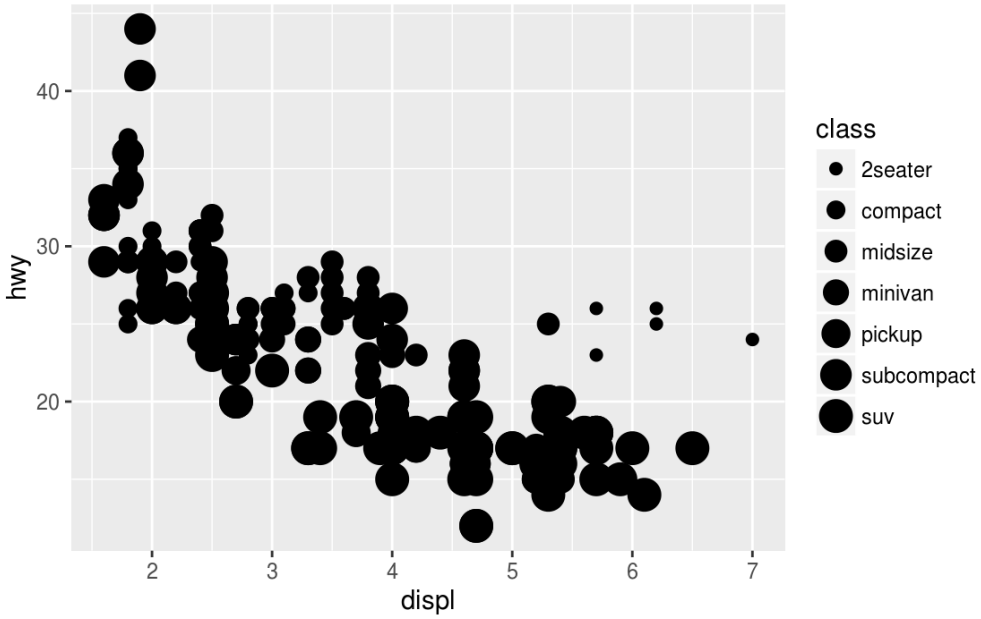
(If you prefer British English, like Hadley, you can use `colour` instead of `color`.)

To map an aesthetic to a variable, set the name of the aesthetic to the name of the variable inside `aes()`. ggplot2 will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable, a process known as **scaling**. ggplot2 will also add a legend that explains which levels correspond to which values.

The colors reveal that many of the unusual points are two seater cars. These cars don't seem like hybrids, and are, in fact, sports cars! Sports cars have large engines like SUVs and pickup trucks, but small bodies like midsize and compact cars, which improves their gas mileage. In hindsight, these cars were unlikely to be hybrids since they have large engines.

In the above example, we mapped `class` to the color aesthetic, but we could have mapped `class` to the size aesthetic in the same way. In this case, the exact size of each point would reveal its class affiliation. We get a *warning* here, because mapping an unordered variable (`class`) to an ordered aesthetic (`size`) is not a good idea.

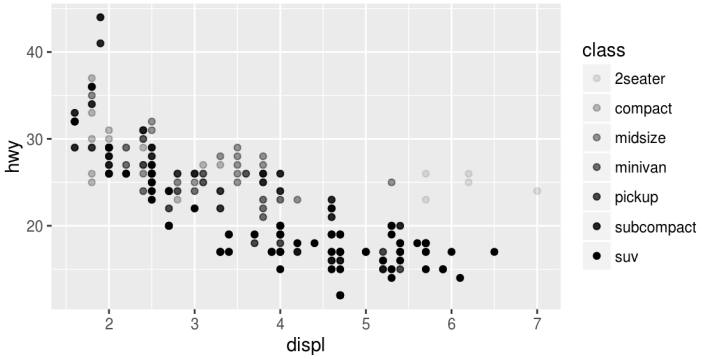
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, size = class))
#> Warning: Using size for a discrete variable is not advised.
```

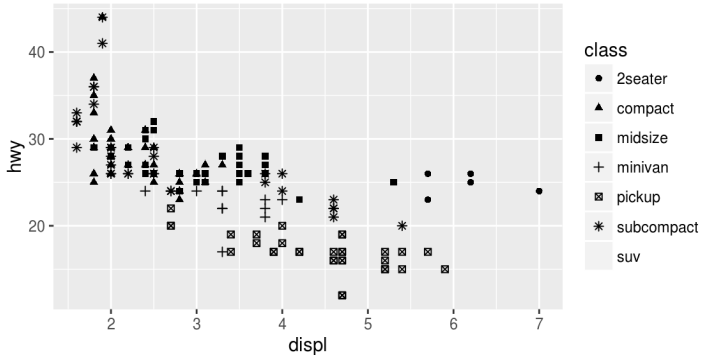


Or we could have mapped `class` to the *alpha* aesthetic, which controls the transparency of the points, or the shape of the points.

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

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





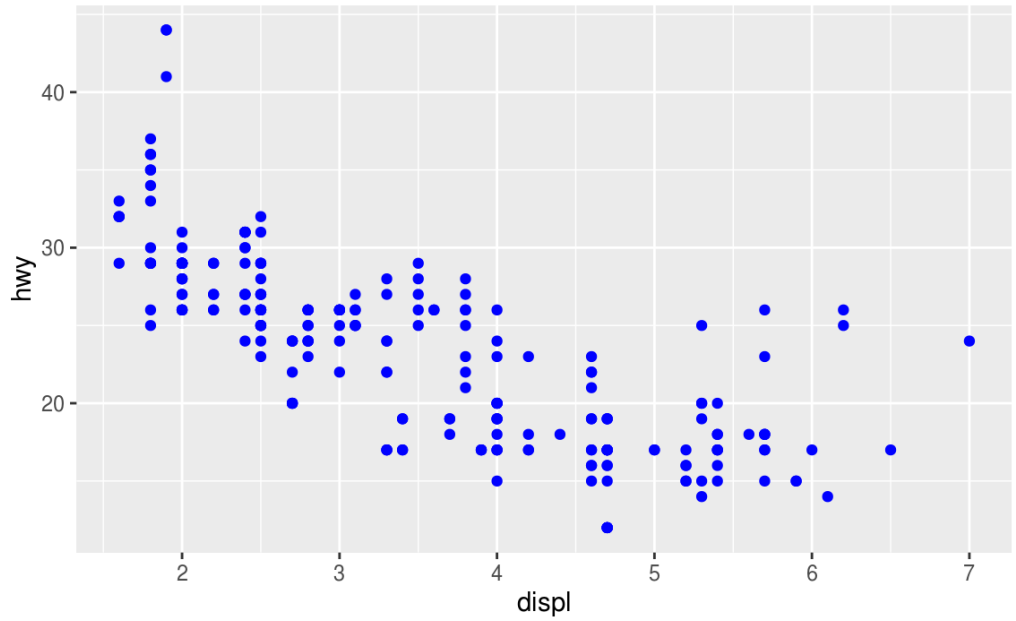
What happened to the SUVs? ggplot2 will only use six shapes at a time. Additional groups will go unplotted when you use this aesthetic.

For each aesthetic, you set the name of the aesthetic to the variable to display within the `aes()` function. The `aes()` function gathers together each of the aesthetic mappings used by a layer and passes them to the layer's mapping argument. The syntax highlights a useful insight about `x` and `y` : the x and y locations of a point are themselves aesthetics, visual properties that you can map to variables to display information about the data.

Once you set an aesthetic, ggplot2 takes care of the rest. It selects a reasonable scale to use with the aesthetic, and it constructs a legend that explains the mapping between levels and values. For x and y aesthetics, ggplot2 does not create a legend, but it creates an axis line with tick marks and a label. The axis line acts as a legend; it explains the mapping between locations and values.

You can also set the aesthetic properties of your geom manually. For example, we can make all of the points in our plot blue:

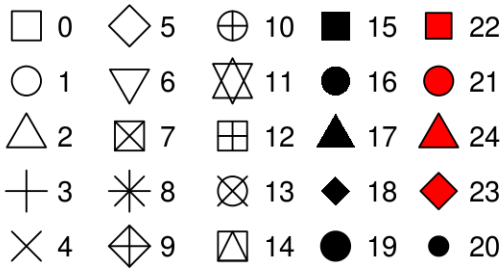
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```



Here, the color doesn't convey information about a variable, but only changes the appearance of the plot. To set an aesthetic manually, set the aesthetic by name as an argument of your geom function. You'll need to pick a value that makes sense for that aesthetic:

- the name of a color as a character string.
- the size of a point in mm.
- the shape as a point as a number, as shown below.

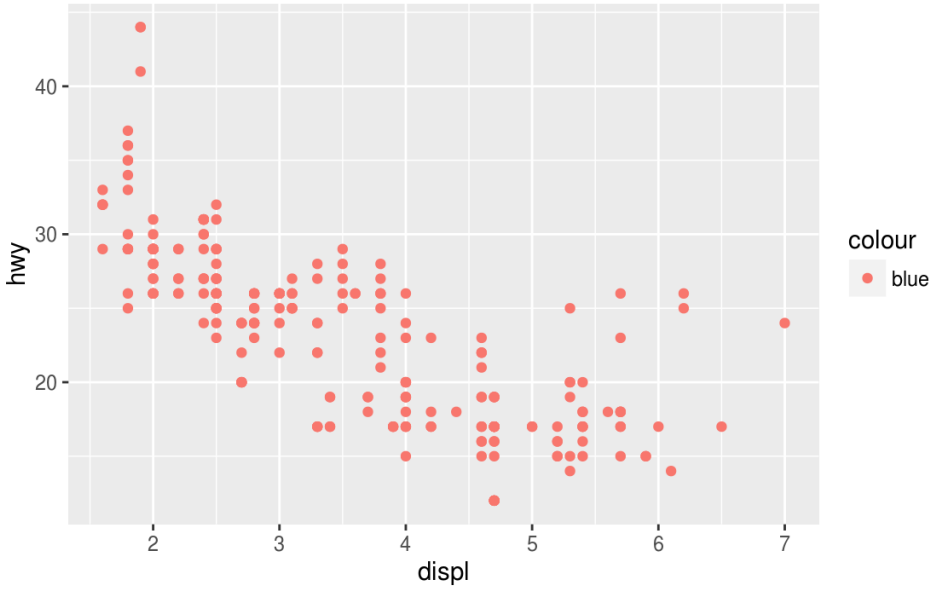
R has a set of 24 built-in shapes, identified by numbers:



3.3.1 Exercises

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

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



2. Which variables in `mpg` are discrete? Which variables are continuous? (Hint: type `?mpg` to read the documentation for the dataset). How can you see this information when you run `mpg` ?

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

4. What happens if you map the same variable across multiple aesthetics? What happens if you map different variables across multiple aesthetics?

5. What does the `stroke` aesthetic do? What shapes does it work with? (Hint: use `?geom_point`)

6. What happens if you set an aesthetic to something other than a variable name, like `displ < 5` ?

7. Vignettes are long-form guides the documentation things about a package that affect many functions. ggplot2 has two vignettes. How can you find them and what do they describe? (Hint: google is your friend.)

3.4 Common problems

As you start to run R code, you're likely to run into problems. Don't worry — it happens to everyone. I have been writing R code for years, and every day I still write code that doesn't work!

Start by carefully comparing the code that you're running to the code in the book. R is extremely picky, and a misplaced character can make all the difference. Make sure that every `(` is matched with a `)` and every `"` is paired with another `"`. Sometimes you'll run the code and nothing happens. Check the left-hand of your console: if it's a `+`, it means that R doesn't think you've typed a complete expression and it's waiting for you to finish it. In this case, it's usually easiest to start from scratch again by pressing `Escape` to abort processing the current command.

One common problem when creating ggplot2 graphics is to put the `+` in the wrong place: it has to come at the end of the line, not the start. In other words, make sure you haven't accidentally written code like this:

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

If you're still stuck, try the help. You can get help about any R function by runnning `?function_name` in the console, or selecting the function name and pressing F1 in RStudio. Don't worry if the help doesn't seem that helpful - instead skip down to the examples and look for code that matches what you're trying to do.

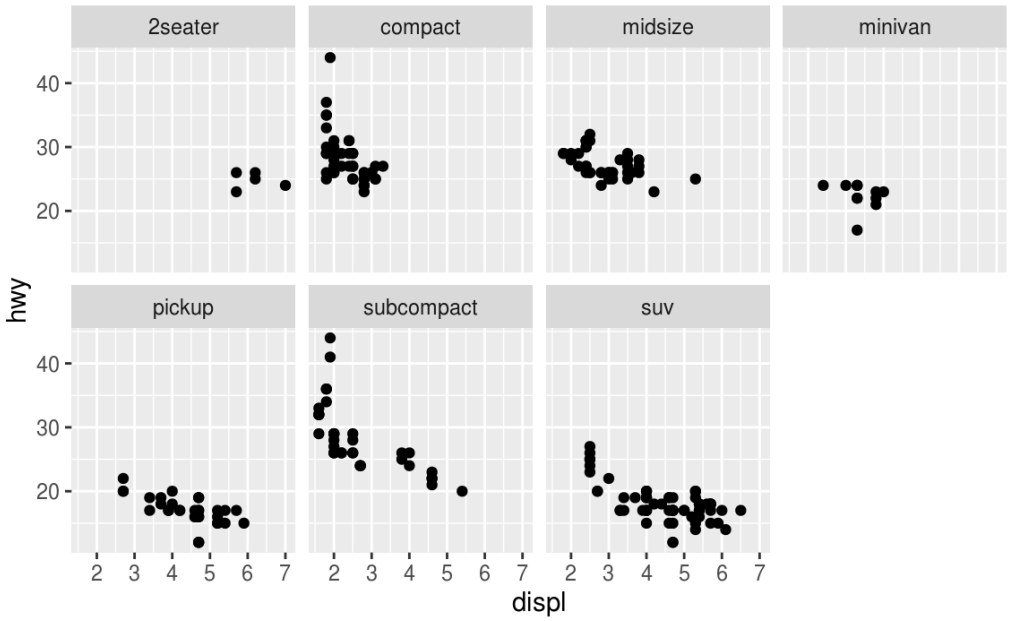
If that doesn't help, carefully read the error message. Sometimes the answer will be buried there! But when you're new to R, the answer might be in the error message but you don't yet know how to understand it. Another great tool is google: trying googling the error message, as it's likely someone else has had the same problem, and have gotten help online.

3.5 Facets

One way to add additional variables is with aesthetics. Another way, particularly useful for categorical variables, is to split your plot into **facets**, subplots that each display one subset of the data.

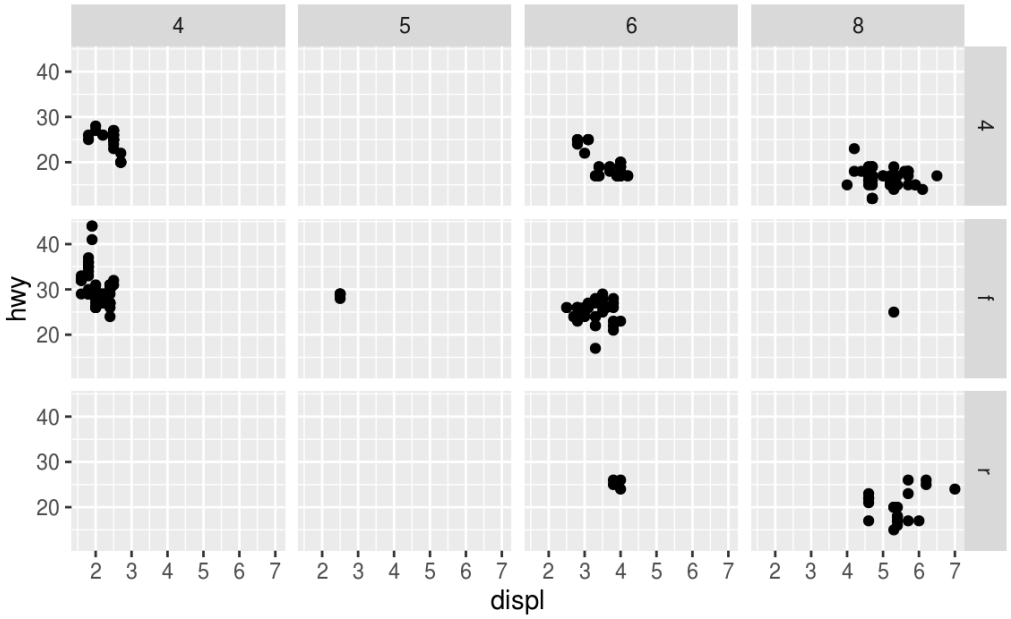
To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` should be a formula, which you create with `~` followed by a variable name (here "formula" is the name of a data structure in R, not a synonym for "equation"). The variable that you pass to `facet_wrap()` should be discrete.

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



To facet your plot on the combination of two variables, add `facet_grid()` to your plot call. The first argument of `facet_grid()` is also a formula. This time the formula should contain two variable names separated by a `~`.

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



If you prefer to not facet in the rows or columns dimension, use a `.` instead of a variable name, e.g. `+ facet_grid(. ~ clarity)` .

3.5.1 Exercises

1. What happens if you facet on a continuous variable?
2. What do the empty cells in plot with `facet_grid(drv ~ cyl)` mean? How do they relate to this plot?

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

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

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

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

4. Take the first faceted plot in this section:

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

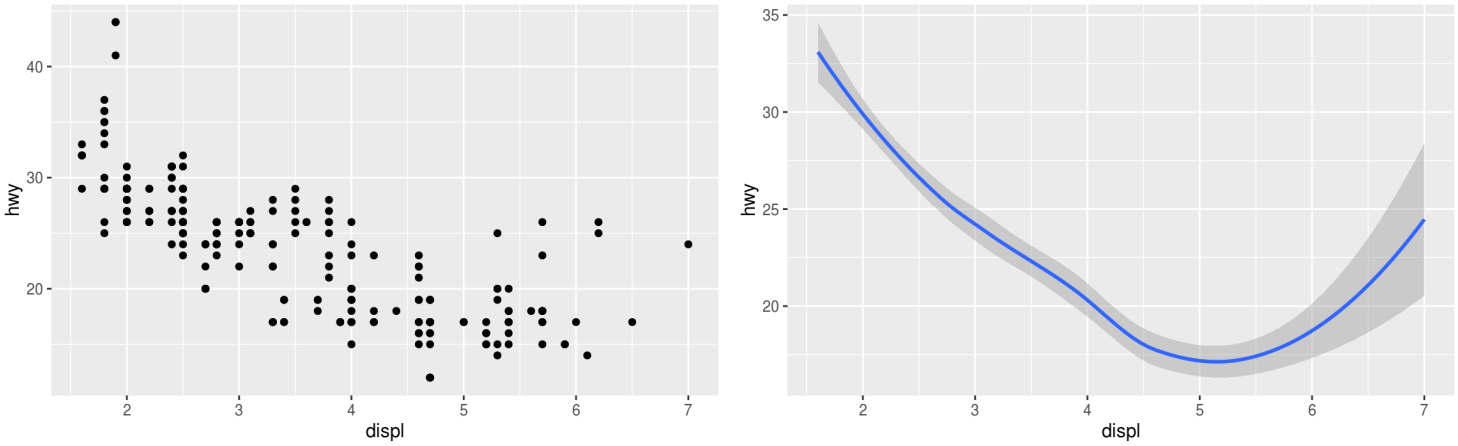
   What are the advantages to using faceting instead of the colour aesthetic? What are the disadvantages? How might the balance change if you had a larger dataset?

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

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

3.6 Geometric objects

How are these two plots similar?



Both plots contain the same x variable, the same y variable, and both describe the same data. But the plots are not identical. Each plot uses a different visual object to represent the data. In ggplot2 syntax, we say that they use different **geoms**.

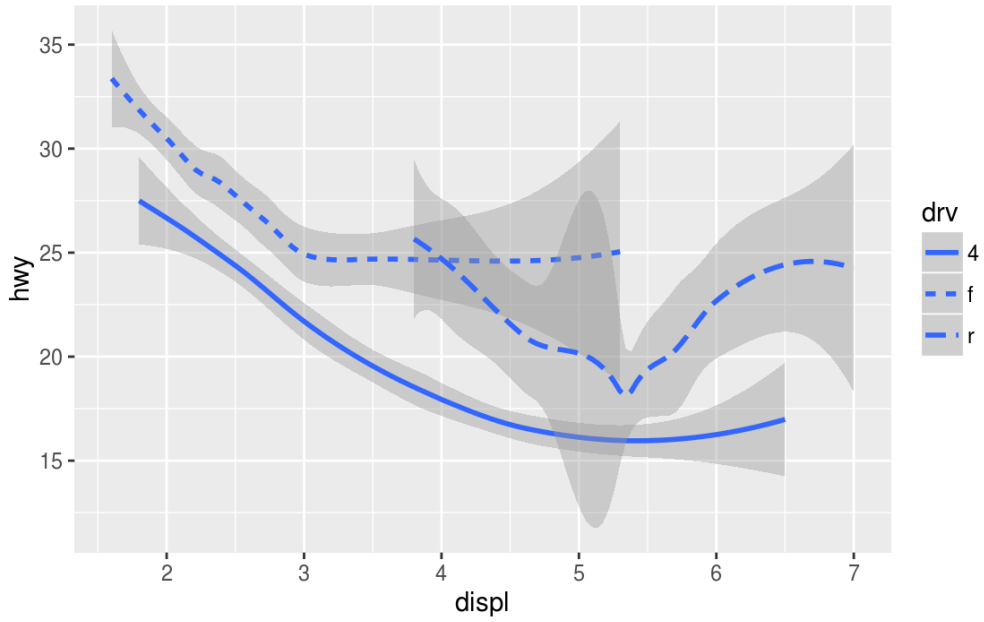
A **geom** is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on. Scatterplots break the trend; they use the point geom. As we see above, you can use different geoms to plot the same data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

To change the geom in your plot, change the geom function that you add to `ggplot()`. For instance, to make the plots above, you can use this code:

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

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

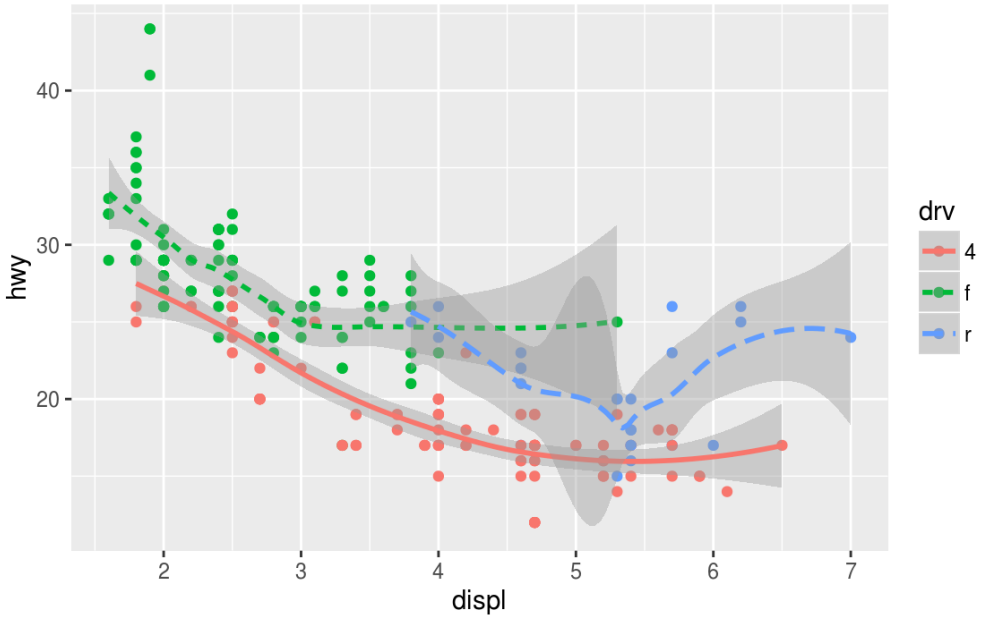
Every geom function in ggplot2 takes a `mapping` argument. However, not every aesthetic works with every geom. You could set the shape of a point, but you couldn't set the "shape" of a line. On the other hand, you *could* set the linetype of a line. `geom_smooth()` will draw a different line, with a different linetype, for each unique value of the variable that you map to linetype.

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



Here `geom_smooth()` separates the cars into three lines based on their `drv` value, which describes a car's drivetrain. One line describes all of the points with a `4` value, one line describes all of the points with an `f` value, and one line describes all of the points with an `r` value. Here, `4` stands for four wheel drive, `f` for front wheel drive, and `r` for rear wheel drive.

If this sounds strange, we can make it more clear by overlaying the lines on top of the raw data and then coloring everything according to `drv` .



Notice that this plot contains two geoms in the same graph! If this makes you excited, buckle up. In the next section, we will learn how to place multiple geoms in the same plot.

ggplot2 provides over 30 geoms, and extension packages provide even more (see https://www.ggplot2-exts.org for a sampling). The table below lists the geoms in ggplot2, loosely organized by the type of relationship that they visualise. Beneath each geom is a list of aesthetics the geom understands, and mandatory aesthetics are bolded. The geom call lists the most important arguments. To learn more about any single geom, open its help page in R by running the command `?` followed by the name of the geom function, e.g. `?geom_smooth`.

Graphical Primitives

These geoms plot basic visual objects. You can use them to construct more sophisticated geoms.

geom_blank()
(Useful for expanding limits)

geom_curve()
x, xend, y, yend, alpha, angle, color, curvature, linetype, size

geom_path()
x, y, alpha, color, group, linetype, size

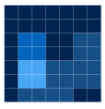
geom_polygon()
x, y, alpha, color, fill, group, linetype, size

geom_rect()
xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size

geom_ribbon()
x, ymax, ymin, alpha, color, fill, group, linetype, size

geom_segment()
x, xend, y, yend, alpha, color, linetype, size

geom_spoke()
x, y, angle, radius, alpha, color, linetype, size

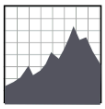


geom_tile()
x, y, alpha, color, fill, linetype, size, width

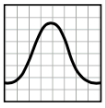
One Variable

These geoms are useful for visualizing the distribution of values in a single variable.

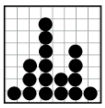
Continuous



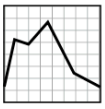
geom_area(stat = "bin"**)**
x, y, alpha, color, fill, linetype, size



geom_density(kernel = "gaussian"**)**
x, y, alpha, color, fill, group, linetype, size, weight



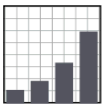
geom_dotplot()
x, y, alpha, color, fill



geom_freqpoly(binwidth = 5**)**
x, y, alpha, color, group, linetype, size

geom_histogram(binwidth = 5**)**
x, y, alpha, color, fill, linetype, size, weight

Discrete

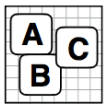


geom_bar()
x, alpha, color, fill, linetype, size, weight

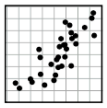
Two Variables

These geoms are useful for visualizing the relationship between two variables.

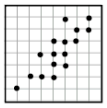
Continuous X, Continuous Y



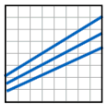
geom_label(check_overlap = TRUE, nudge_x = 1, nudge_y = 1**)**
x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust



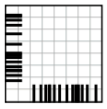
geom_jitter(height = 2, width = 2**)**
x, y, alpha, color, fill, shape, size



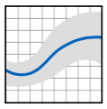
geom_point()
x, y, alpha, color, fill, shape, size, stroke



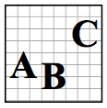
geom_quantile()
x, y, alpha, color, group, linetype, size, weight



geom_rug(sides = "bl"**)**
x, y, alpha, color, linetype, size



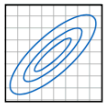
geom_smooth(method = lm, se = FALSE**)**
x, y, alpha, color, fill, group, linetype, size, weight



geom_text(check_overlap = TRUE**)**
x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

Continuous Bivariate Distribution

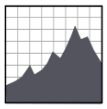
geom_bin2d(binwidth = c(5, 50)**)**
x, y, alpha, color, fill, linetype, size, weight



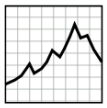
geom_density2d()
x, y, alpha, colour, group, linetype, size

geom_hex()
x, y, alpha, colour, fill, size

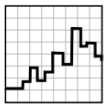
Continuous Function



geom_area()
x, y, alpha, color, fill, linetype, size

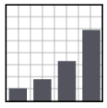


geom_line()
x, y, alpha, color, group, linetype, size

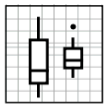


geom_step(direction = "hv"**)**
x, y, alpha, color, group, linetype, size

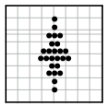
Discrete X, Continuous Y



geom_bar(stat = "identity"**)**
x, y, alpha, color, fill, linetype, size, weight



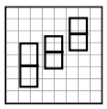
geom_boxplot()
x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight



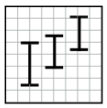
geom_dotplot(binaxis = "y", stackdir = "up"**)**
x, y, alpha, color, fill, group

geom_violin(scale = "area"**)**
x, y, alpha, color, fill, group, linetype, size, weight

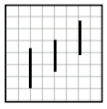
Visualizing error



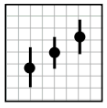
geom_crossbar(fatten = 2**)**
x, y, ymax, ymin, alpha, color, fill, group, linetype, size



geom_errorbar()
x, ymax, ymin, alpha, color, group, linetype, size, width (also **geom_errorbarh()**)

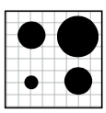


geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size



geom_pointrange()
x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

Discrete X, Discrete Y



geom_count()
x, y, alpha, color, fill, shape, size, stroke

Maps

geom_map(map = map_data("state")**)**
map_id, alpha, color, fill, linetype, size

Three Variables

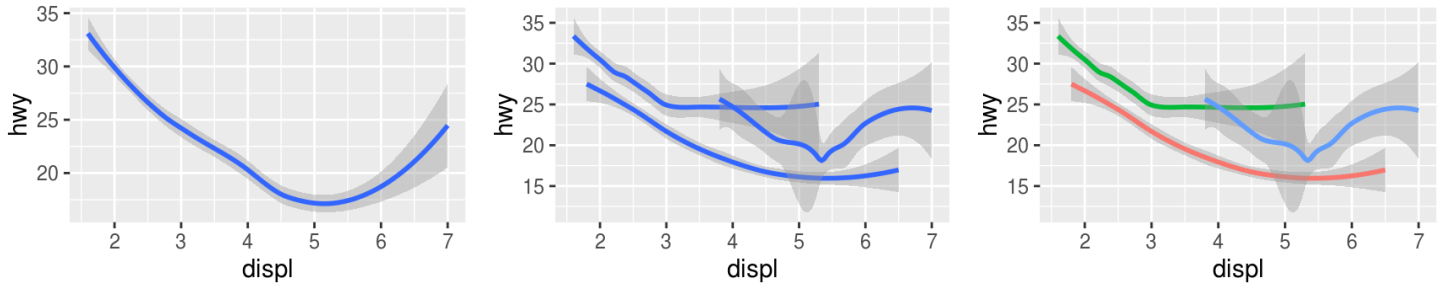
These geoms are useful for visualizing the relationship between three variables.

geom_contour()
x, y, z, alpha, colour, group, linetype, size, weight

geom_raster(hjust=0.5, vjust=0.5, interpolate=FALSE**)**
x, y, alpha, fill

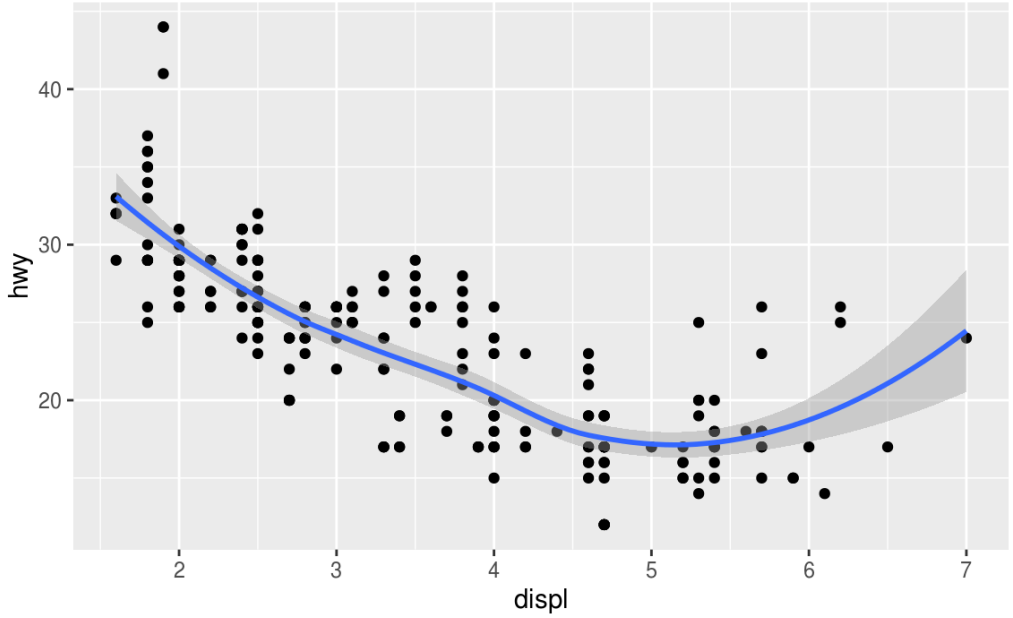
Many geoms, like `geom_smooth()`, use a single geometric object to display multiple rows of data. For these geoms, you can set the `group` aesthetic to a discrete variable to draw multiple objects. ggplot2 will draw a separate object for each unique value of the grouping variable. In practice, ggplot2 will automatically group the data for these geoms whenever you map an aesthetic to a discrete variable (as in the `linetype` example). It is convenient to rely on this feature because the group aesthetic by itself does not add a legend or distinguishing features to the geoms.

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

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

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



To display multiple geoms in the same plot, add multiple geom functions to `ggplot()`:

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

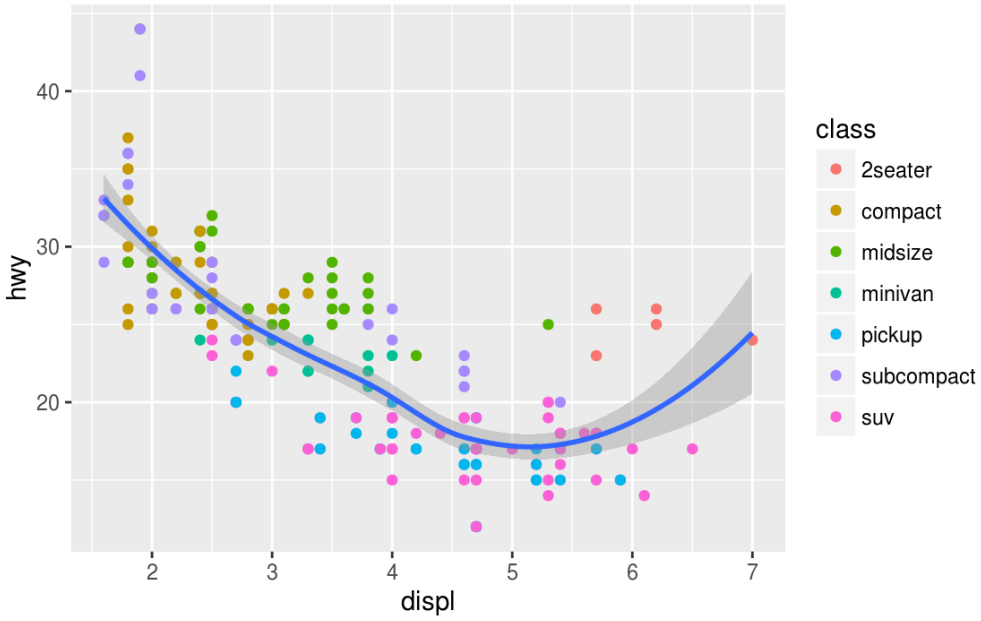


This, however, introduces some duplication in our code. Imagine if you wanted to change the y-axis to display `cty` instead of `hwy`. You'd need to change the variable in two places, and you might forget to update one. You can avoid this type of repetition by passing a set of mappings to `ggplot()`. ggplot2 will treat these mappings as global mappings that apply to each geom in the graph. In other words, this code will produce the same plot as the previous code:

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

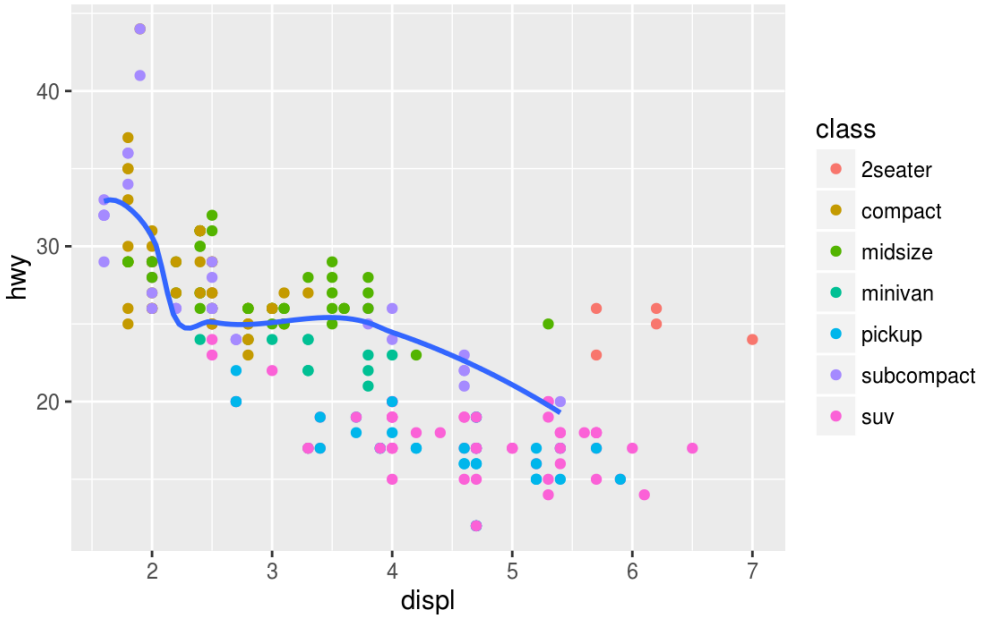
If you place mappings in a geom function, ggplot2 will treat them as local mappings for the layer. It will use these mappings to extend or overwrite the global mappings *for that layer only*. This makes it possible to display different aesthetics in different layers.

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



You can use the same idea to specify different `data` for each layer. Here, our smooth line displays just a subset of the `mpg` dataset, the subcompact cars. The local data argument in `geom_smooth()` overrides the global data argument in `ggplot()` for that layer only.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth(data = dplyr::filter(mpg, class == "subcompact"), se = FALSE)
```



(Remember, `dplyr::filter()` calls the `filter()` function from the dplyr package. You'll learn how `filter()` works in the next chapter.)

3.6.1 Exercises

1. What geom would you use to draw a line chart? A boxplot? A histogram? An area chart?
2. Run this code in your head and predict what the output will look like. Run the code in R and check your predictions.

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

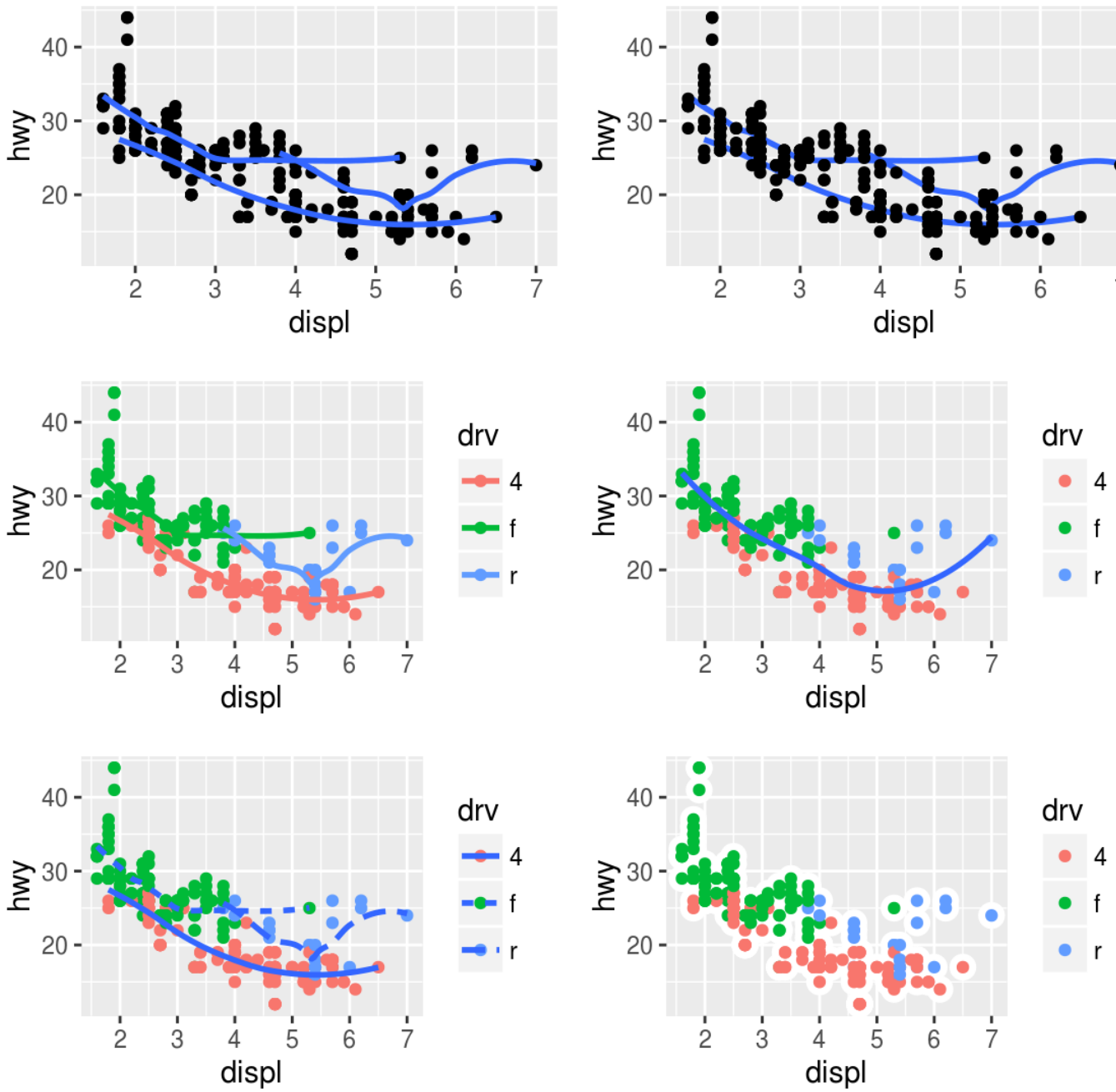
3. What does `show.legend = FALSE` do? What happens if you remove it? Why do you think I used it in the example above.

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

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

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

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

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

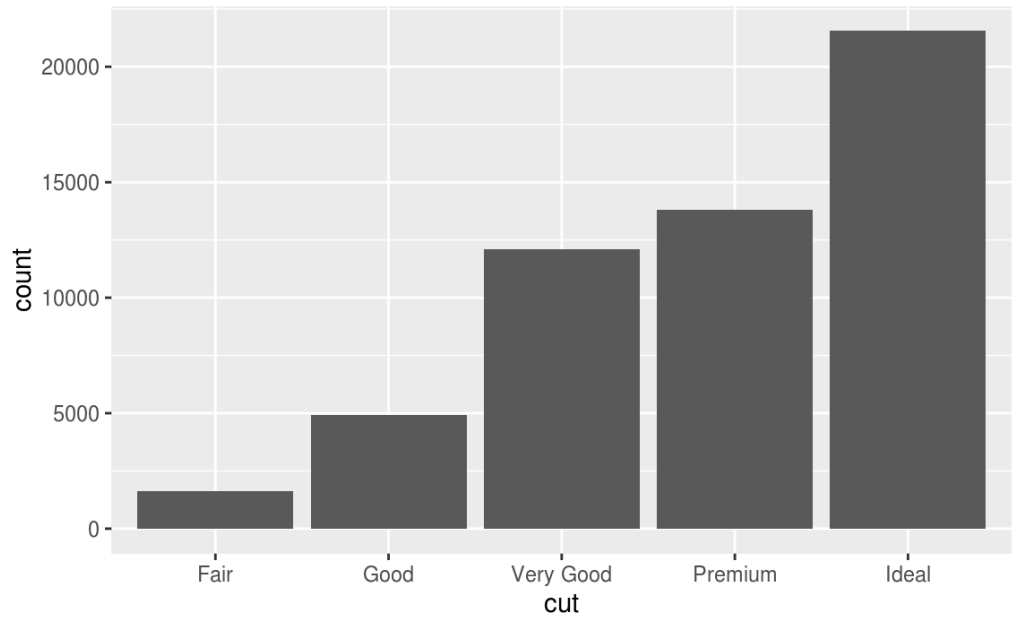


3.7 Statistical transformations

Next, let's take a look at a bar chart. Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with `geom_bar()`. The following chart displays the total number of diamonds in the `diamonds` dataset, grouped by `cut`. The `diamonds` dataset comes in ggplot2 and contains information

about ~54,000 diamonds, including the `price`, `carat`, `color`, `clarity`, and `cut` of each diamond. The chart shows that more diamonds are available with high quality cuts than with low quality cuts.

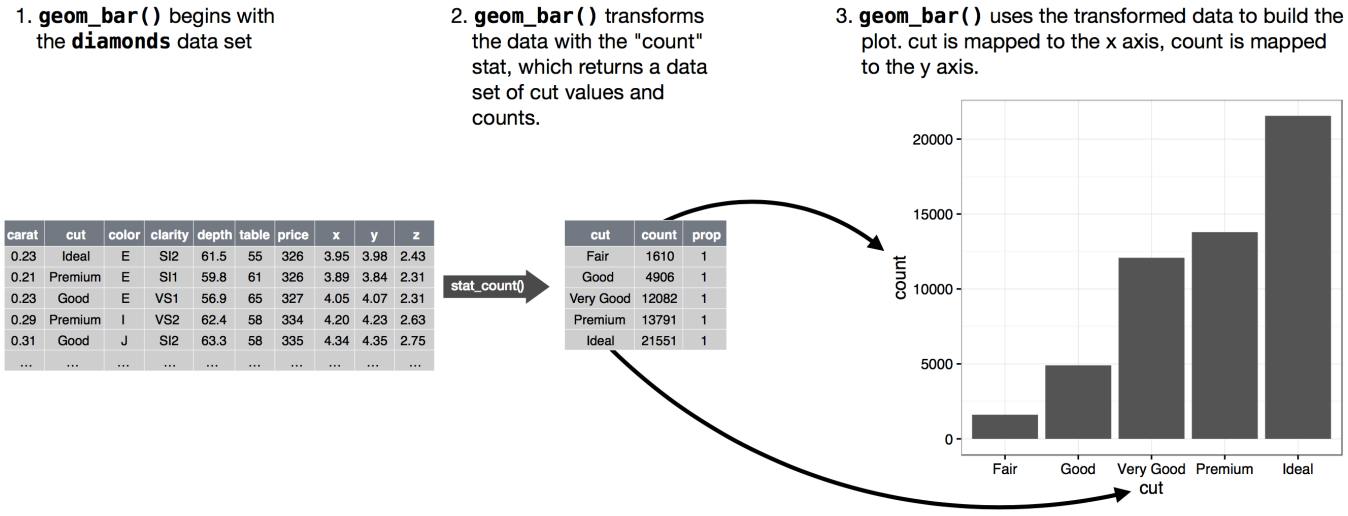
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```



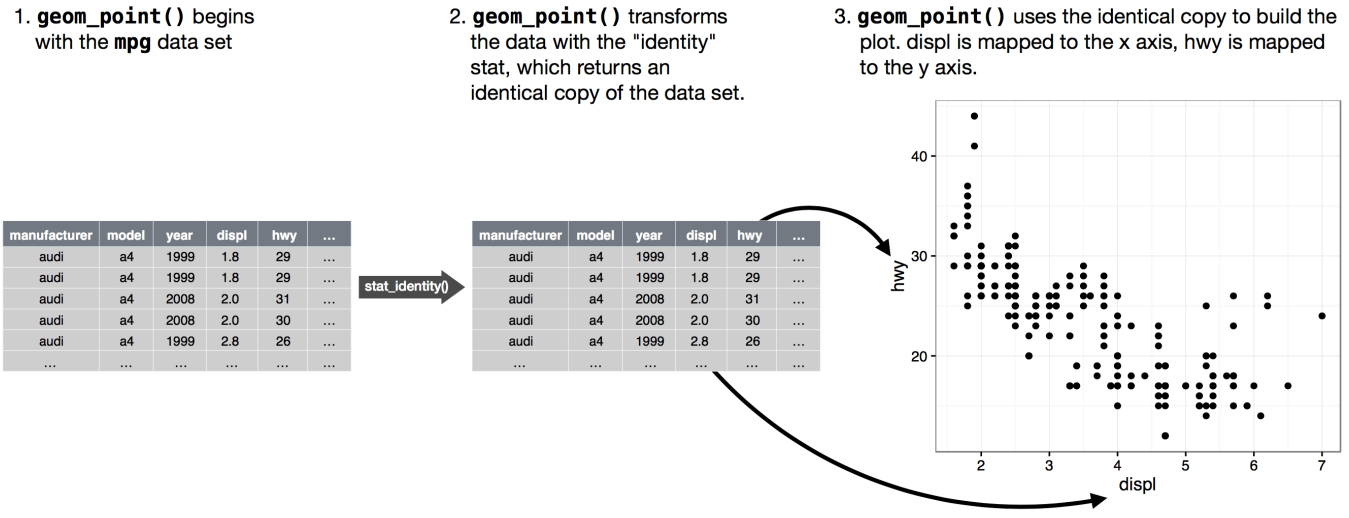
On the x-axis, the chart displays `cut`, a variable from `diamonds`. On the y-axis, it displays count, but count is not a variable in `diamonds`! Where does count come from? Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

- **bar charts**, **histograms**, and **frequency polygons** bin your data and then plot bin counts, the number of points that fall in each bin.
- **smoothers** fit a model to your data and then plot predictions from the model.
- **boxplots** calculate the quartiles of your data and then plot the quartiles as a box.

ggplot2 calls the algorithm that a graph uses to calculate new values a **stat**, which is short for statistical transformation. Each geom in ggplot2 is associated with a default stat that it uses to calculate values to plot. The figure below describes how this process works with `geom_bar()`.



A few geoms, like `geom_point()`, plot your raw data as it is. These geoms also apply a transformation to your data, the identity transformation, which returns the data in its original state. Now we can say that *every* geom uses a stat.

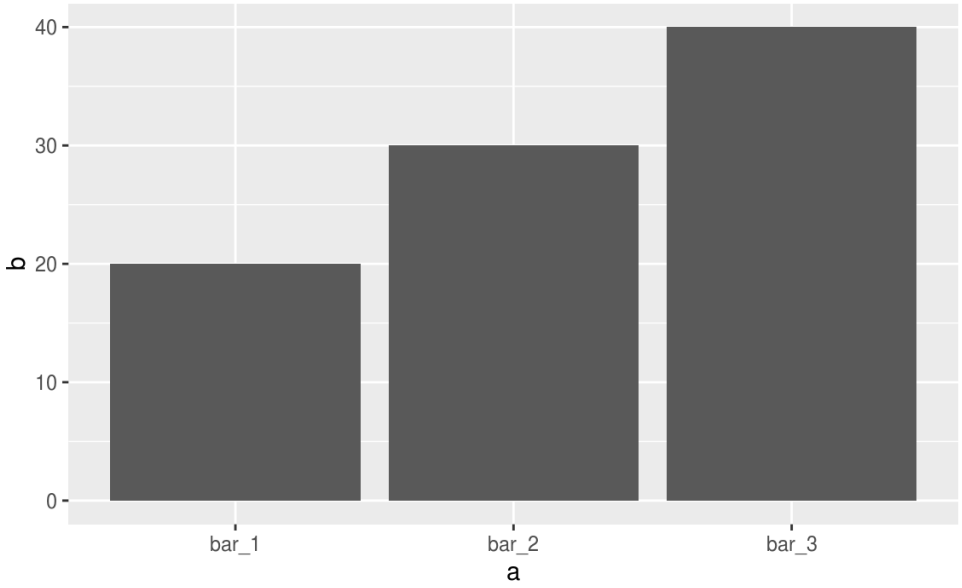


You can learn which stat a geom uses, as well as what variables it computes by visiting the geom's help page. For example, the help page of `geom_bar()` shows that it uses the count stat and that the count stat computes two new variables, `count` and `prop` .

Stats are the most subtle part of plotting because you can't see them directly. ggplot2 applies the transformation and stores the results behind the scenes. You only see impact in the final plot. Generally, you don't need to think about stats: the defaults work away on your behalf to summarise your data as needed for a particular plot. However, there are two cases where you might need to know about it:

1. You might want to override the default stat. In the code below, I change the stat of `geom_bar()` from count (the default) to identity. This lets me map to the height of the bars to the raw values of a y variable.

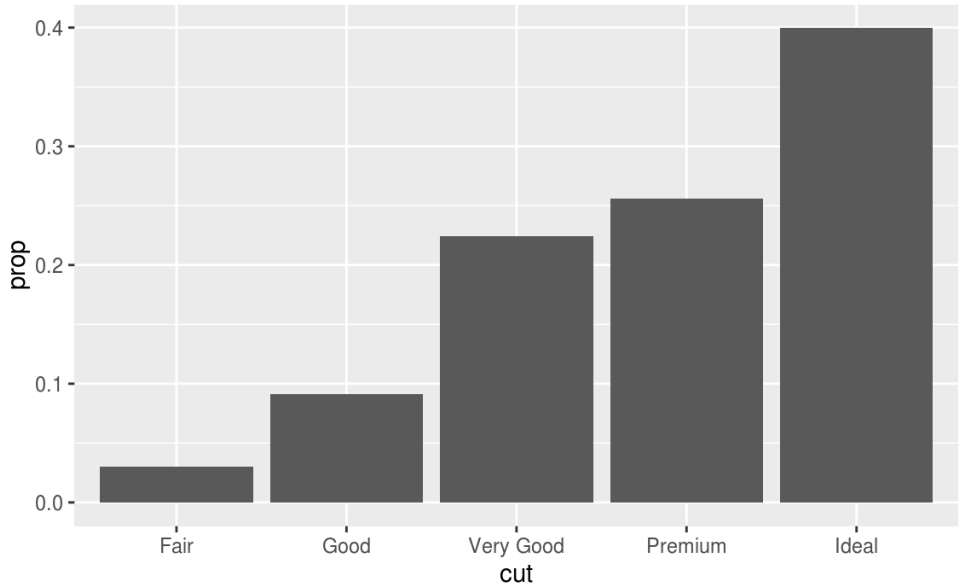
```
demo <- tibble::tibble(
  a = c("bar_1","bar_2","bar_3"),
  b = c(20, 30, 40)
)
demo
#> # A tibble: 3 x 2
#>       a     b
#>   <chr> <dbl>
#> 1 bar_1    20
#> 2 bar_2    30
#> 3 bar_3    40

ggplot(data = demo) +
  geom_bar(mapping = aes(x = a, y = b), stat = "identity")
```



(Unfortunately when people talk about bar charts casually, they might be referring to this type of bar chart, where the height of the bar is already present in the data, or the previous bar chart where the height of the bar is generated by counting rows.)

2. You might want to override the default mapping from transformed variables to aesthetics. For example, you might want to display a bar chart of proportion, rather than count:

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



The help page of `?geom_bar` reveals that the sum stat creates two variables, `count` and `prop`. By default, `geom_bar()` maps `y` to `count`, but you can ask it to use `prop` instead with `aes(y = ..prop..)`.The two dots that surround prop notify ggplot2 that the `prop` variable appears in the transformed dataset not in the raw dataset.

ggplot2 provides over 20 stats for you to use. Each stat is saved as a function, which provides a convenient way to access a stat's help page, e.g. `?stat_identity`. The table below describes each stat in ggplot2 and lists the parameters that the stat takes, as well as the variables that the stat makes.

Stats

A stat transforms raw data into new variables to plot.
Each stat is listed with the parameters it takes
and the variables it creates (surrounded by ..).

stat_bin() - Bins and counts continuous data.
bins, binwidth, breaks, drop, origin, right, width
..count.., ..ncount.., ..density.., ..ndensity..

stat_bin_2d() - Bins and counts 2D continuous data.
bins, binwidth, drop
..count.., ..density..

stat_bin_hex() - 2D bins with hexagonal shape.
bins, binwidth
..count.., ..density..

stat_boxplot() - Computes boxplot parameters.
coef
..lower.., ..middle.., ..notchlower.., ..notchupper..,
..upper.., ..width.., ..ymin.., ..ymax..

stat_contour() - Computes contours of 3D surface.
..level..

stat_count() - Counts discrete data.
width
..count.., ..prop..

stat_density() - Computes density kernel estimate.
adjust, kernel, trim
..count.., ..density.., ..scaled..

stat_density_2d() - 2D density kernel estimate.
contour, n, h
..level..

stat_ecdf(n = 40**)** - Computes empirical CDF.
..x.., ..y..

stat_ellipse() - Computes model ellipses.
level, segments, t

stat_function() - Applies a function to x variable.
args, fun, n
..x.., ..y..

stat_identity() - Returns data as is

stat_qq() - Calculates quantile quantile plot.
distribution, dparams
..sample.., ..theoretical..

stat_quantile() - Computes quantiles.
formula, quantiles, method, method.args
..quantile..

stat_smooth() - Computes model line.
formula, fullrange, level, method, n, se, span
..se.., ..x.., ..y.., ..ymin.., ..ymax..

stat_sum() - Counts discrete data.
..n.., ..prop..

stat_summary() - Applies summary to groups of unique x values.
fun.args, fun.data, fun.y, fun.ymin, fun.ymax

stat_summary_2d() - Applies summary to groups of 2D binned values.
bins, binwidth, drop, fun, fun.args
..value..

stat_summary_bin() - Applies summary to groups of binned x values.
fun.args, fun.data, fun.y, fun.ymin, fun.ymax

stat_summary_hex() - Applies summary to groups of 2D hexagonally binned values.
bins, binwidth, drop, fun, fun.args
..value..

stat_unique() - Removes duplicates.

stat_ydensity() - Computes densities for violin plot.
..count.., ..density.., ..n.., ..scaled.., ..violinwidth.., ..width..

3.7.1 Exercises

1. In our proportion barchart, we need to set `group = 1`. Why? In other words, why is this graph not useful?

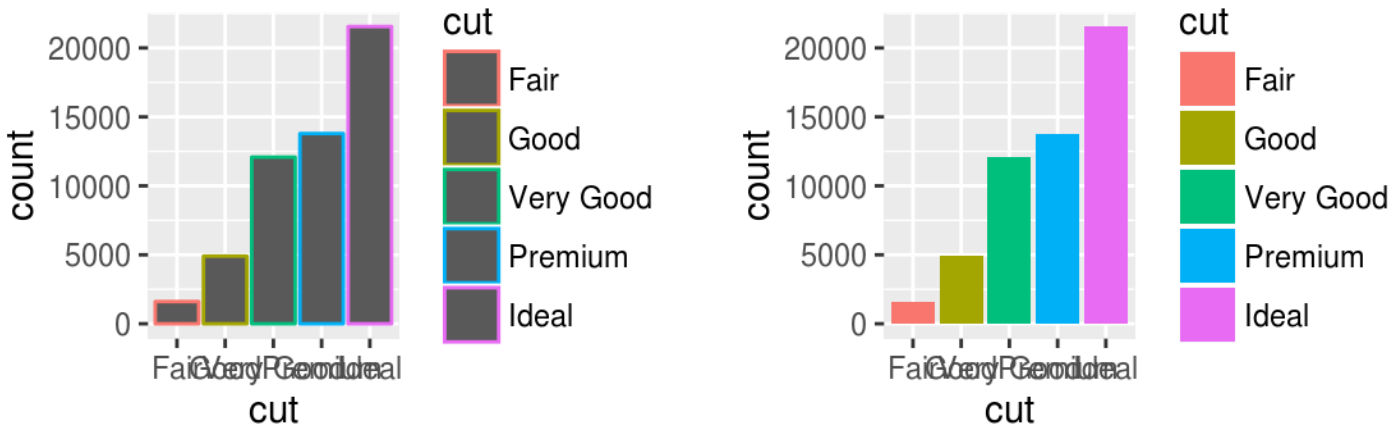
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop..))
```

2. How do you find out the default stat associated with a geom?

3.8 Position adjustments

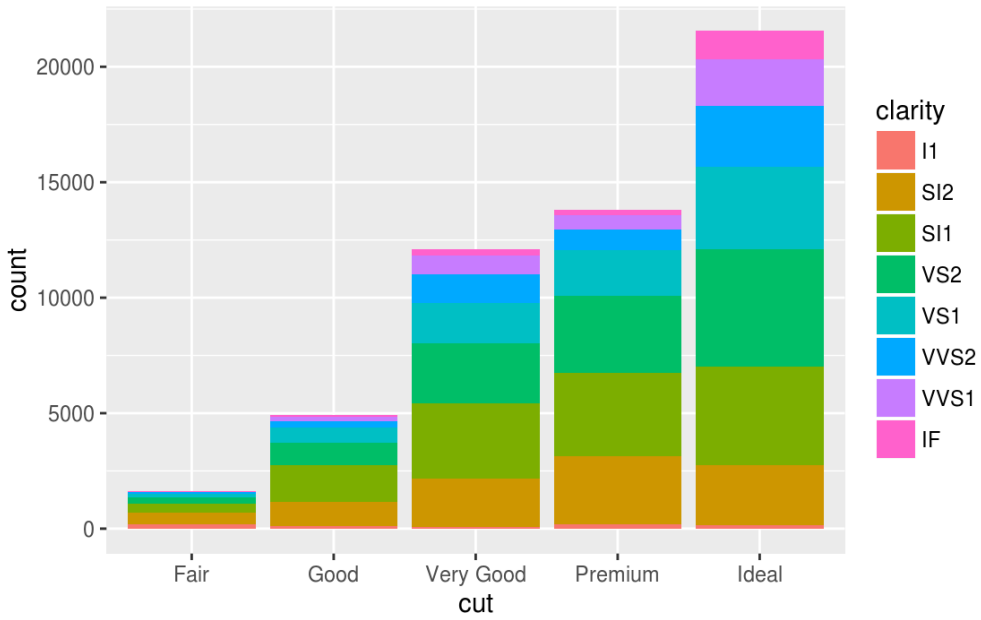
There's one more piece of magic associated with bar charts. You can colour bar chart using either the `colour` aesthetic, or more usefully, `fill`:

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, colour = cut))
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = cut))
```



Note what happens if you map the fill aesthetic to another variable, like `clarity`: the bars are automatically stacked. Each colored rectangle represents a combination of `cut` and `clarity`.

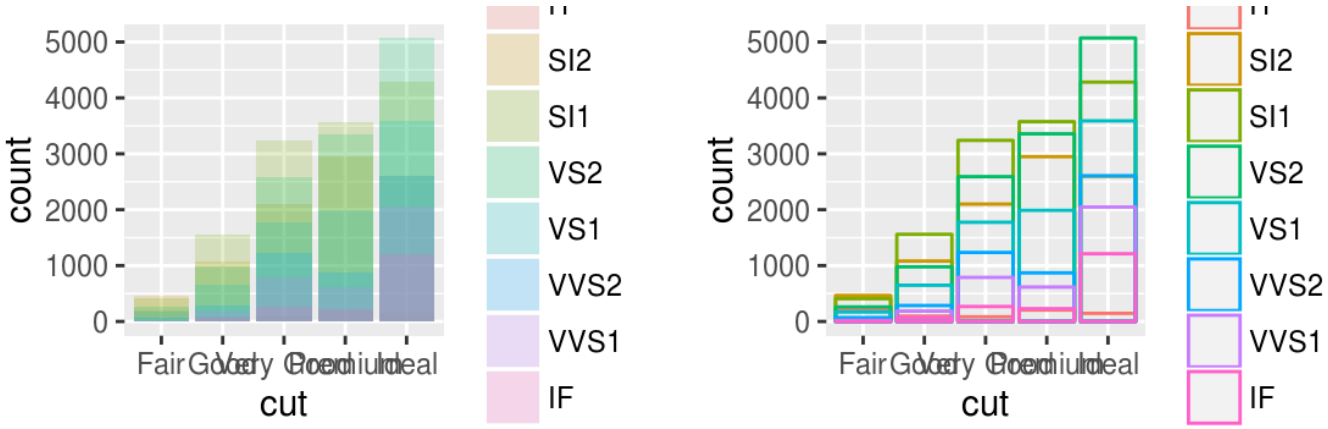
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity))
```



The stacking is performed automatically by the **position adjustment** specified by the `position` argument. If you don't want a stacked bar chart, you can use one of three other options: `"identity"`, `"dodge"` or `"fill"`.

- `position = "identity"` will place each object exactly where it falls in the context of the graph. This is not very useful for bars, because it overlaps them. To see that overlapping we either need to make the bars slightly transparent by setting `alpha` to a small value, or completely transparent by setting `fill = NA`.

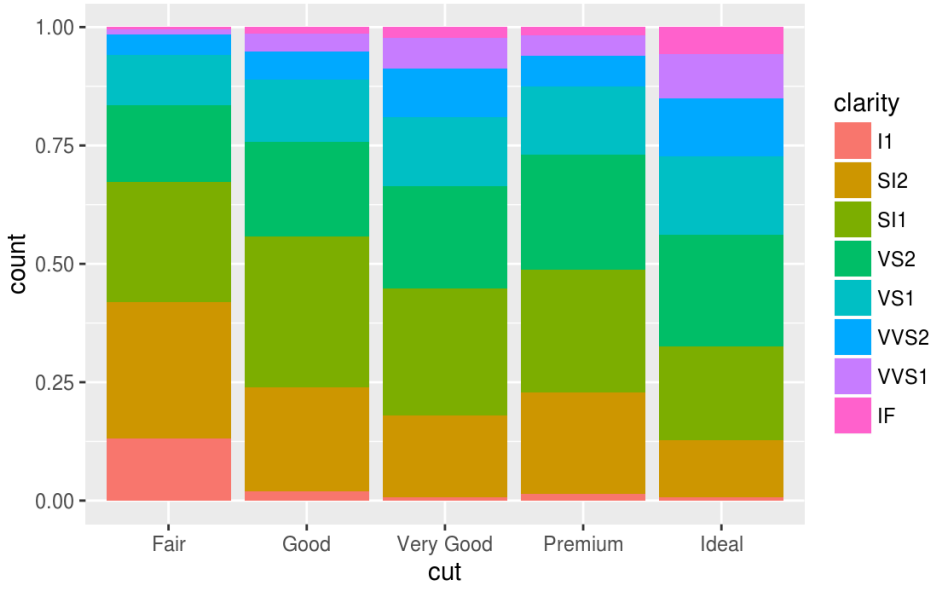
```
ggplot(data = diamonds, mapping = aes(x = cut, fill = clarity)) +
  geom_bar(alpha = 1/5, position = "identity")
ggplot(data = diamonds, mapping = aes(x = cut, colour = clarity)) +
  geom_bar(fill = NA, position = "identity")
```



The identity position adjustment is more useful for 2d geoms, like points, where it is the default.

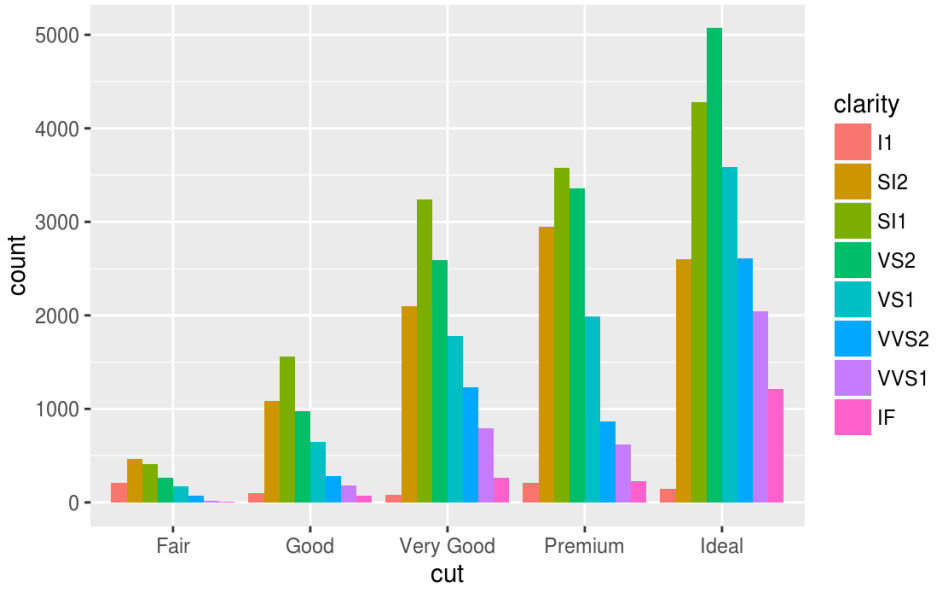
- `position = "fill"` work like stacking, but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "fill")
```

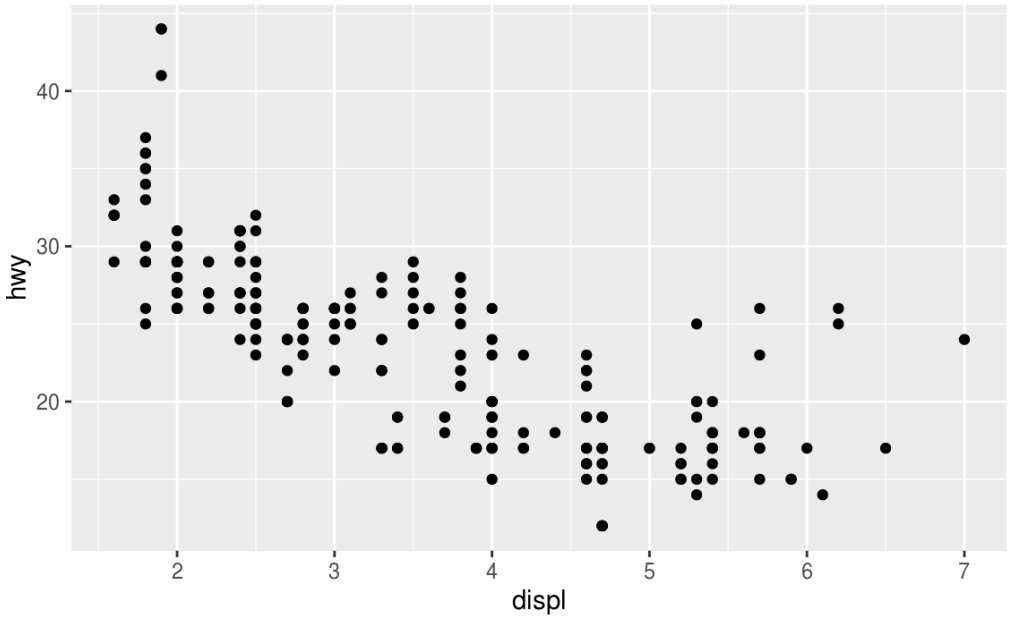


- `position = "dodge"` places overlapping objects directly *beside* one another. This makes it easier to compare individual values.

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "dodge")
```



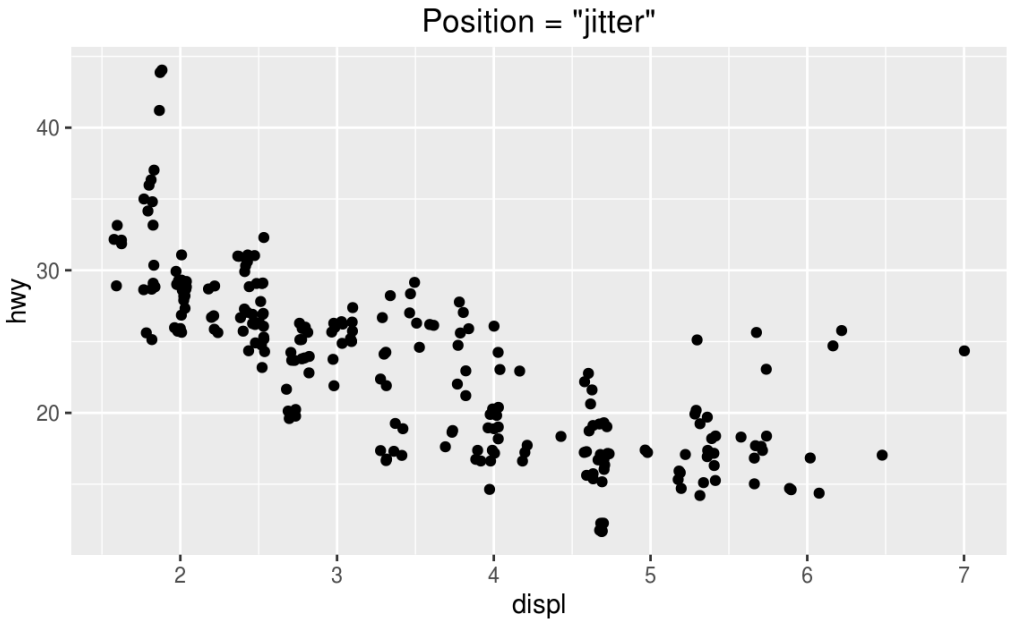
There's one other type of adjustment that's not useful for bar charts, but it can be very useful for scatterplots. Recall our first scatterplot. Did you notice that the plot displays only 126 points, even though there are 234 observations in the dataset?



The values of `hwy` and `displ` are rounded so the points appear on a grid and many points overlap each other. This problem is known as **overplotting**. This arrangement makes it hard to see where the mass of the data is. Are the data points spread equally throughout the graph, or is there one special combination of `hwy` and `displ` that contains 109 values?

You can avoid this gridding by setting the position adjustment to "jitter". `position = "jitter"` adds a small amount of random noise to each point. This spreads the points out because no two points are likely to receive the same amount of random noise.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), position = "jitter") +
  ggtitle('Position = "jitter"')
```



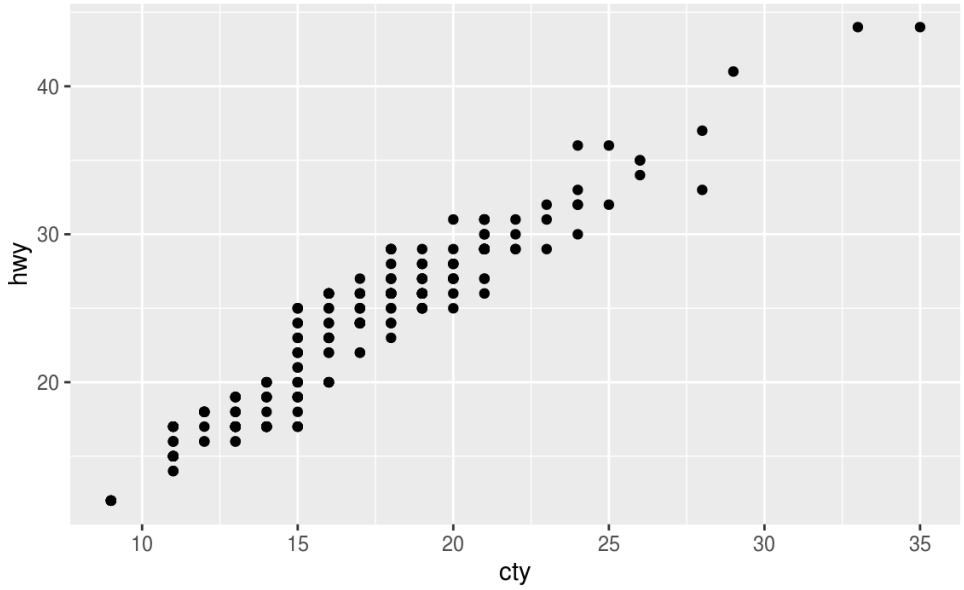
Adding randomness seems like a strange way to improve your plot, but while it makes your graph less accurate at small scales, it makes your graph *more* revealing at large scales. Because this is such a useful operation, ggplot2 comes with a shorthand for `geom_point(position = "jitter")` : `geom_jitter()` .

To learn more about a position adjustment, look up the help page associated with each adjustment: `?position_dodge` , `?position_fill` , `?position_identity` , `?position_jitter` , and `?position_stack` .

3.8.1 Exercises

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

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



2. Compare and contrast `geom_jitter()` with `geom_count()`.

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

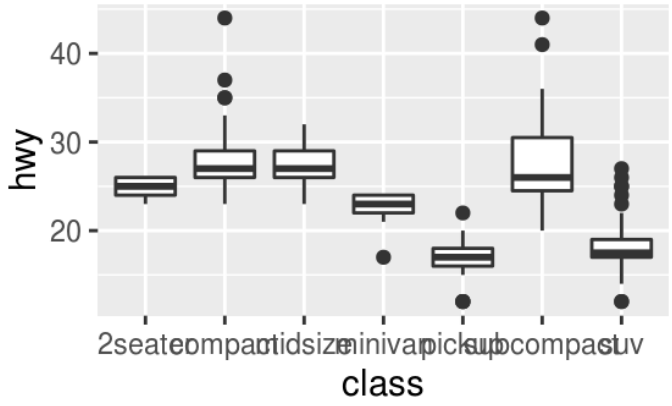
3.9 Coordinate systems

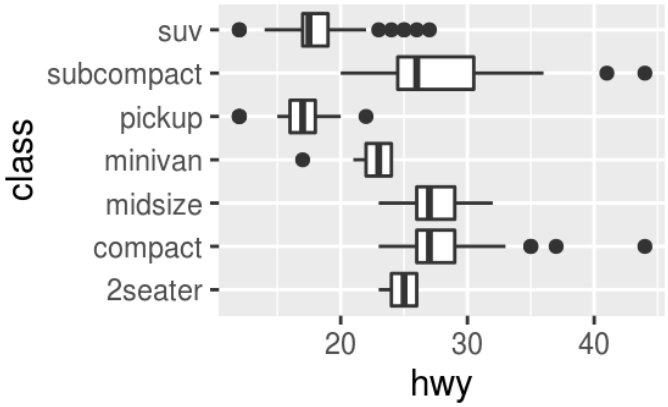
Coordinate systems are probably the most complicated part of ggplot2. The default coordinate system is the Cartesian coordinate system where the x and y position act independently to find the location of each point.

There are a number of other coordinate systems that are occassionally helpful.

- `coord_flip()` switches the x and y axes. This is useful (for example), if you want vertical boxplots.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```



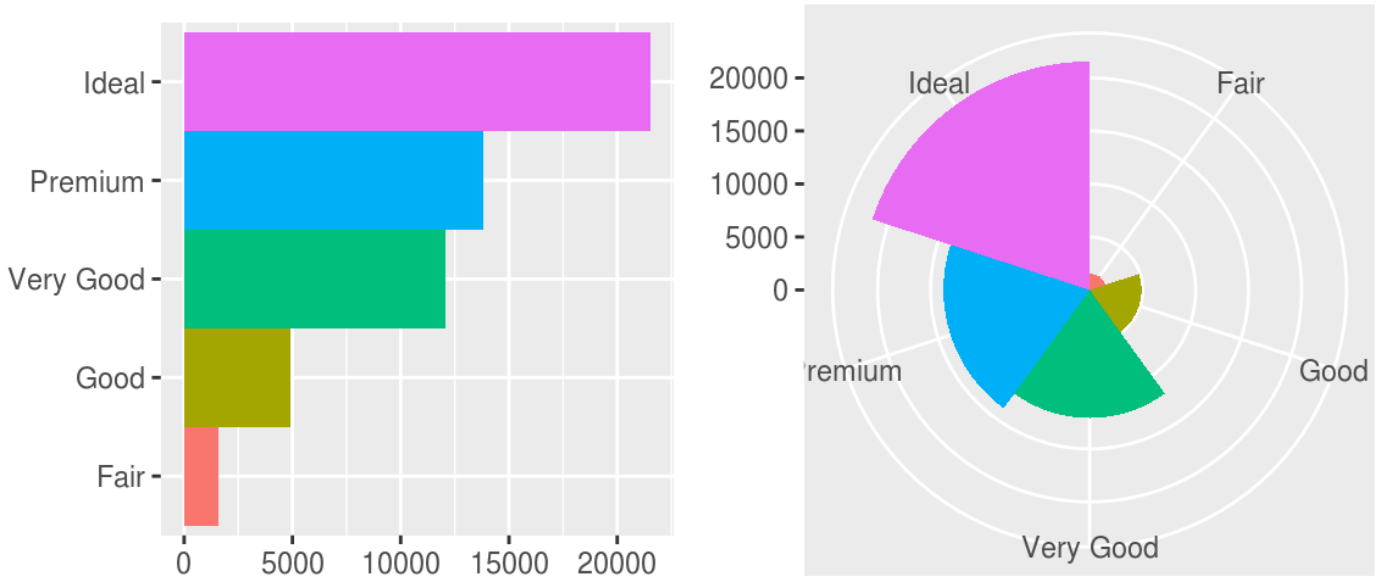


- `coord_quickmap()` sets the aspect ratio correctly for maps. This is very important if you're plotting spatial data with ggplot2 (which unfortunately we don't have the space to cover in this book).

```
nz <- map_data("nz")

ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black")

ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black") +
  coord_quickmap()
```

- `coord_polar()` uses polar coordinates. Polar coordinates reveals an interesting connection between a bar chart and a Coxcomb chart.

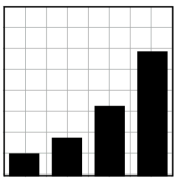
```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  xlab(NULL) +
  ylab(NULL)

bar + coord_flip()
bar + coord_polar()
```



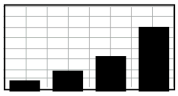
The table below describes each built-in coord. You can learn more about each coordinate system by opening its help page in R, e.g. `?coord_cartesian` .

Coordinate Systems

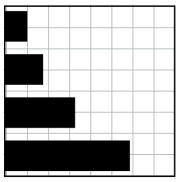
ggplot2 comes with eight coordinate systems to draw plots in.



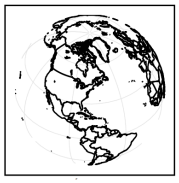
coord_cartesian()
xlim, ylim
Cartesian coordinate system (the default)



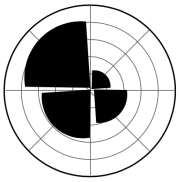
coord_fixed()
ratio, xlim, ylim
Cartesian coordinate system with fixed aspect ratio between x and y units. See also **coord_equal()**



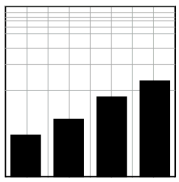
coord_flip()
xlim, ylim
Cartesian coordinate system with x and y axes flipped



coord_map()
projection, orientation, xlim, ylim
Map projections from the mapproj package. See also **coord_quickmap()**



coord_polar()
theta, start, direction
Polar coordinate system

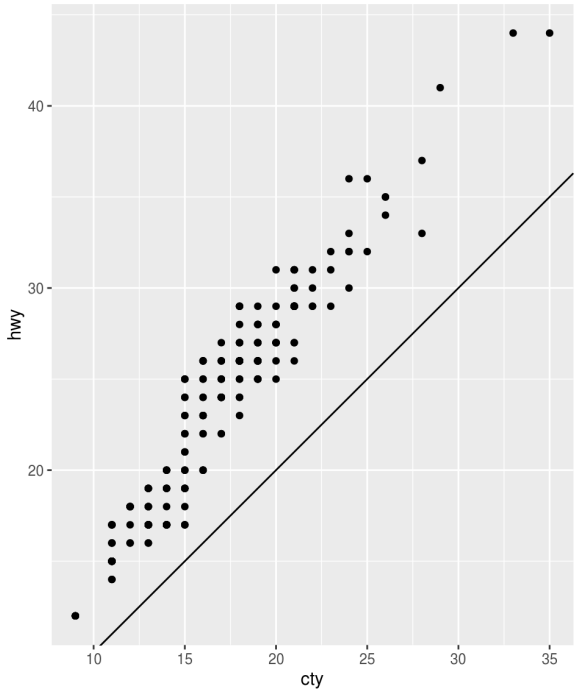


coord_trans()
xtrans, ytrans, limx, limy
Cartesian coordinate system with x and y axes transformed by a function

3.9.1 Exercises

1. Turn a stacked bar chart into a pie chart using `coord_polar()` .
2. What's the difference between `coord_quickmap()` and `coord_map()` ?
3. What does the plot below tell you about the relationship between city and highway mpg? Why is `coord_fixed()` important? What does `geom_abline()` do?

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



3.10 The layered grammar of graphics

In the previous sections, you learned much more than how to make scatterplots, bar charts, and boxplots. You learned a foundation that you can use to make *any* type of plot with ggplot2. To see this, lets add position adjustments, stats, coordinate systems, and faceting to our code template:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>,
     position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

Our new template takes seven parameters, the bracketed words that appear in the template. In practice, you rarely need to supply all seven parameters to make a graph because ggplot2 will provide useful defaults for everything except the data, the mappings, and the geom function.

The seven parameters in the template compose the grammar of graphics, a formal system for building plots. The grammar of graphics is based on the insight that you can uniquely describe *any* plot as a combination of a dataset, a geom, a set of mappings, a stat, a position adjustment, a coordinate system, and a faceting scheme.

To see how this works, consider how you could build a basic plot from scratch: you could start with a dataset and then transform it into the information that you want to display (with a stat).

1. Begin with the **`diamonds`** data set
2. Compute counts for each cut value with **`stat_count()`**.

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
…	…	…	…	…	…	…	…	…	…

stat_count()

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

Next, you could choose a geometric object to represent each observation in the transformed data. You could then use the aesthetic properties of the geoms to represent variables in the data. You would map the values of each variable to the levels of an aesthetic.

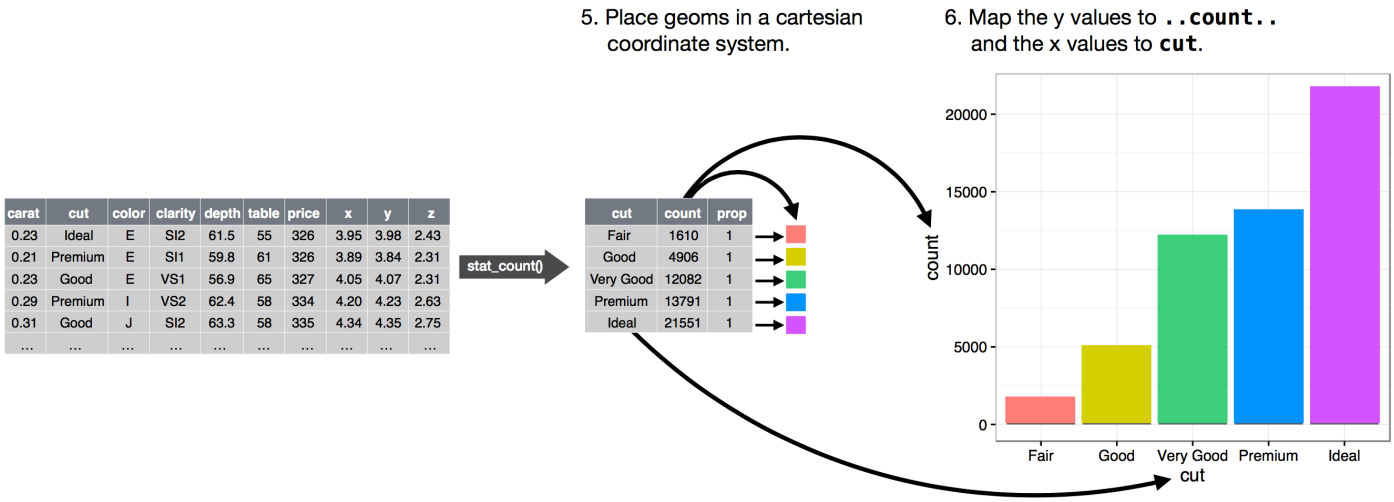
3. Represent each observation with a bar.
4. Map the **`fill`** of each bar to the **`..count..`** variable.

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
…	…	…	…	…	…	…	…	…	…

stat_count()

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

You'd then select a coordinate system to place the geoms into. You'd use the location of the objects (which is itself an aesthetic property) to display the values of the x and y variables. At that point, you would have a complete graph, but you could further adjust the positions of the geoms within the coordinate system (a position adjustment) or split the graph into subplots (faceting). You could also extend the plot by adding one or more additional layers, where each additional layer uses a dataset, a geom, a set of mappings, a stat, and a position adjustment.



You could use this method to build *any* plot that you imagine. In other words, you can use the code template that you've learned in this chapter to build hundreds of thousands of unique plots.