# r4ds-ggplot2

2023-12-10

# Contents

# R for data science workshop

This website us the ggplot visulization sections of Hadley Wickham's book 2nd edition of "R for Data Science".

# Chapter 1

# Communication

## 1.1 Introduction

For publishing purposes, you need to *communicate* your understanding to others. Your audience will likely not share your background knowledge and will not be deeply invested in the data. To help others quickly build up a good mental model of the data, you will need to invest considerable effort in making your plots as self-explanatory as possible. In this chapter, you'll learn some of the tools that ggplot2 provides to do so.

This chapter focuses on the tools you need to create good graphics. We assume that you know what you want, and just need to know how to do it. For that reason, we highly recommend pairing this chapter with a good general visualization book. We particularly like The Truthful Art, by Albert Cairo. It doesn't teach the mechanics of creating visualizations, but instead focuses on what you need to think about in order to create effective graphics.

### 1.1.1 Prerequisites

In this chapter, we'll focus once again on ggplot2. We'll also use a little **dplyr** for data manipulation, **scales** to override the default breaks, labels, transformations and palettes, and a few ggplot2 extension packages, including **ggrepel** (https://ggrepel.slowkow.com) by Kamil Slowikowski and **patchwork** (https://patchwork.data-imaginist.com) by Thomas Lin Pedersen. Don't forget that you'll need to install those packages with `install.packages()` if you don't already have them.

Uncomment the following line if you need to install any of these packages (you should have `tidyverse`:

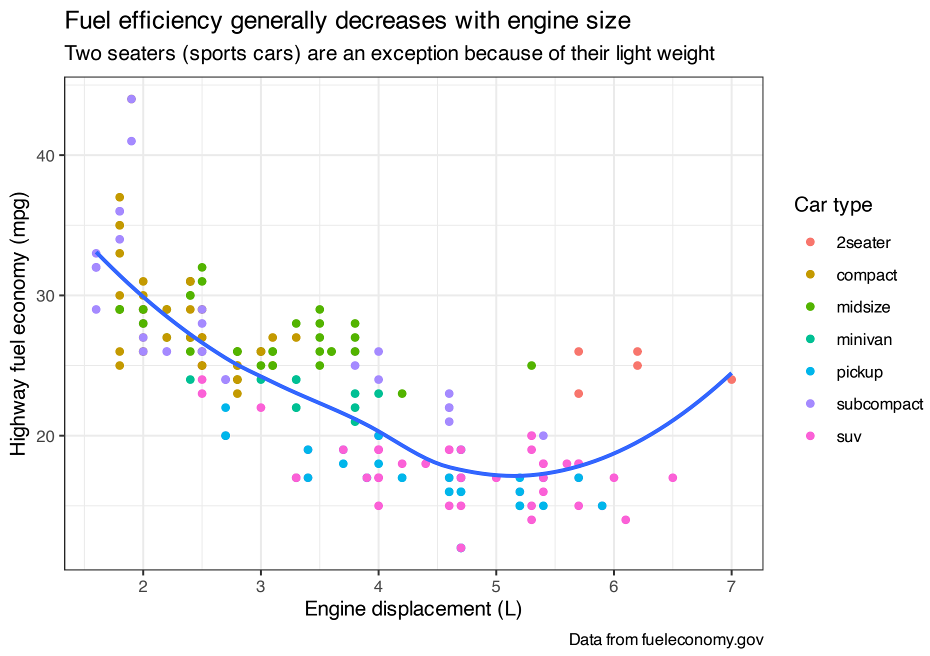
```
# install.packages(c('ggrepel','patchwork','scales','ggthemes'))
```

```
library(tidyverse)
library(scales)
library(ggrepel)
library(patchwork)
library(ggthemes)
```

## 1.2 Labels

The easiest place to start when turning an exploratory graphic into an expository graphic is with good labels. You add labels with the `labs()` function.

```
# | fig-alt: | | Scatterplot of highway fuel efficiency
# versus engine size of cars, where | points are colored
# according to the car class. A smooth curve following |
# the trajectory of the relationship between highway fuel
# efficiency versus | engine size of cars is overlaid. The
# x-axis is labelled 'Engine | displacement (L)' and the
# y-axis is labelled 'Highway fuel economy (mpg)'.  | The
# legend is labelled 'Car type'. The plot is titled 'Fuel
# efficiency | generally decreases with engine size'. The
# subtitle is 'Two seaters | (sports cars) are an exception
# because of their light weight' and the | caption is 'Data
# from fueleconomy.gov'.

ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = class)) +
    geom_smooth(se = FALSE) + labs(x = "Engine displacement (L)",
    y = "Highway fuel economy (mpg)", color = "Car type", title = "Fuel efficiency gene
    subtitle = "Two seaters (sports cars) are an exception because of their light weig
    caption = "Data from fueleconomy.gov")
```

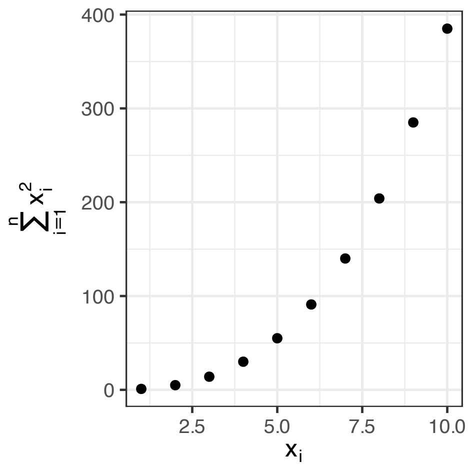


The purpose of a plot title is to summarize the main finding. Avoid titles that just describe what the plot is, e.g., "A scatterplot of engine displacement vs. fuel economy".

If you need to add more text, there are two other useful labels: `subtitle` adds additional detail in a smaller font beneath the title and `caption` adds text at the bottom right of the plot, often used to describe the source of the data. You can also use `labs()` to replace the axis and legend titles. Note the replacement of the legend title by "Car type" in the previous plot. It's usually a good idea to replace short variable names with more detailed descriptions, and to include the units.

It's possible to use mathematical equations instead of text strings. Just switch `""` out for `quote()` and read about the available options in `?plotmath`:

```r
df <- tibble(x = 1:10, y = cumsum(x^2))

ggplot(df, aes(x, y)) + geom_point() + labs(x = quote(x[i]),
    y = quote(sum(x[i]^2, i == 1, n)))
```



### Exercises 1.2.1

1. Create a boxplot using the fuel economy data with x = `hwy` and y = `class` customized `title`, `subtitle`, `caption`, `x`, `y`, and `color` labels.

**Ans-1.2.1.1:**

```
# enter your answer here
```

2. Recreate the following plot using the fuel economy data. Note that both the colors and shapes of points vary by type of drive train.

**Ans-1.2.1.2**

## 1.3 Annotations

In addition to labelling major components of your plot, it's often useful to label individual observations or groups of observations. The first tool you have at your disposal is `geom_text()`. `geom_text()` is similar to `geom_point()`, but it has an additional aesthetic: `label`. This makes it possible to add textual labels to your plots.

There are two possible sources of labels. First, you might have a tibble that provides labels. In the following plot we pull out the cars with the highest engine size in each drive type and save their information as a new data frame called `label_info`.

```
# slice_head operates by group, so when n = 1 it means the
# size of the group case_when where the left hand side
# determines which values match this case and the right
# hand side provides the replacement value.
```

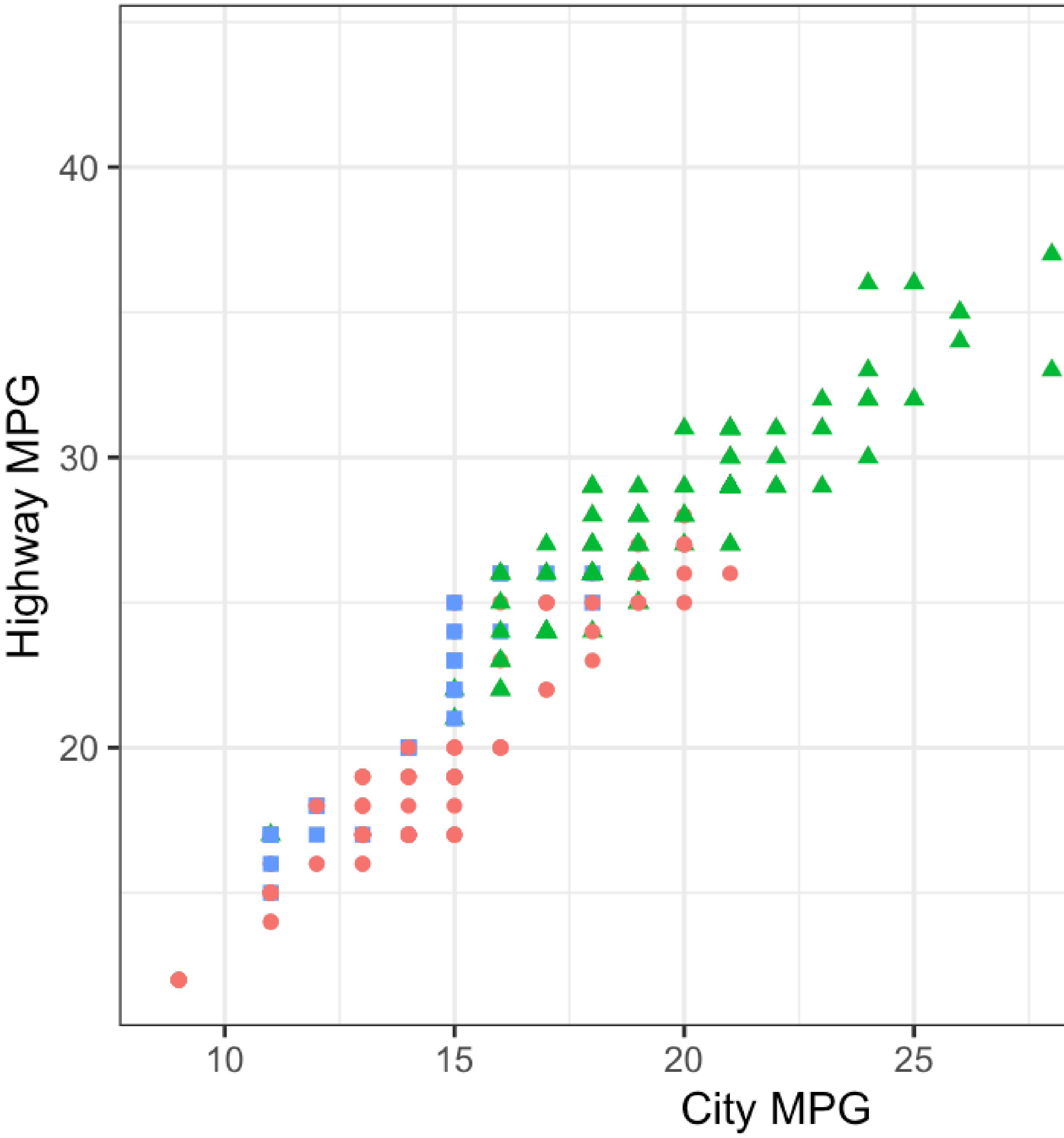


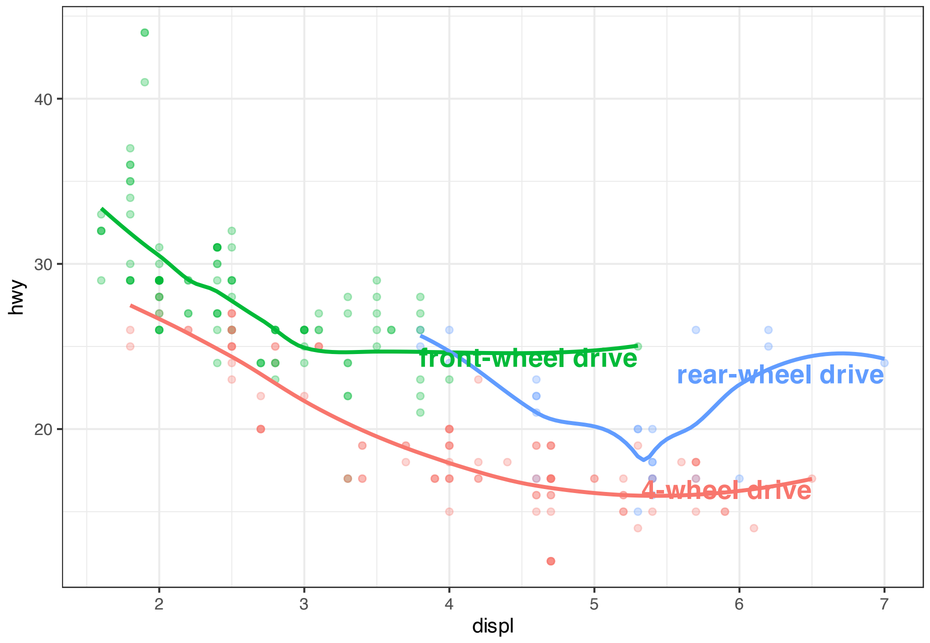
Figure 1.1: Scatterplot of highway versus city fuel efficiency.

```
label_info <- mpg |>
    group_by(drv) |>
    arrange(desc(displ)) |>
    slice_head(n = 1) |>
    mutate(drive_type = case_when(drv == "f" ~ "front-wheel drive",
        drv == "r" ~ "rear-wheel drive", drv == "4" ~ "4-wheel drive")) |>
    select(displ, hwy, drv, drive_type)

label_info
## # A tibble: 3 x 4
## # Groups:   drv [3]
##   displ   hwy drv   drive_type
##   <dbl> <int> <chr> <chr>
## 1   6.5    17 4     4-wheel drive
## 2   5.3    25 f     front-wheel drive
## 3   7      24 r     rear-wheel drive
```

**Notice that `displ` values in `label_info` have captured the right-most point for each drive type.**

Then, we use this new data frame to directly label the three groups to replace the legend with labels placed directly on the plot. Using the `fontface` and `size` arguments we can customize the look of the text labels. They're larger than the rest of the text on the plot and bolded. (`theme(legend.position = "none")` turns all the legends off — we'll talk about it more shortly.)

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) + geom_point(alpha = 0.3) +
    geom_smooth(se = FALSE) + geom_text(data = label_info, aes(x = displ,
    y = hwy, label = drive_type), fontface = "bold", size = 5,
    hjust = "right", vjust = "top") + theme(legend.position = "none")
```

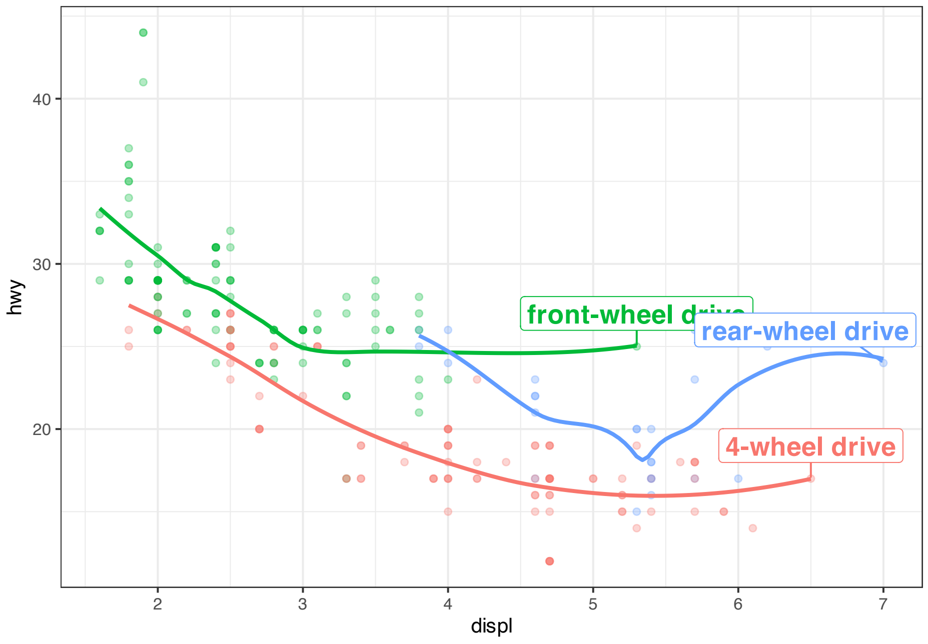


**Once you see the placement you can use `nudge_x` and `nudge_y` to set it so that it doesn't overlap.**

Note the use of `hjust` (horizontal justification) and `vjust` (vertical justification) to control the alignment of the label.

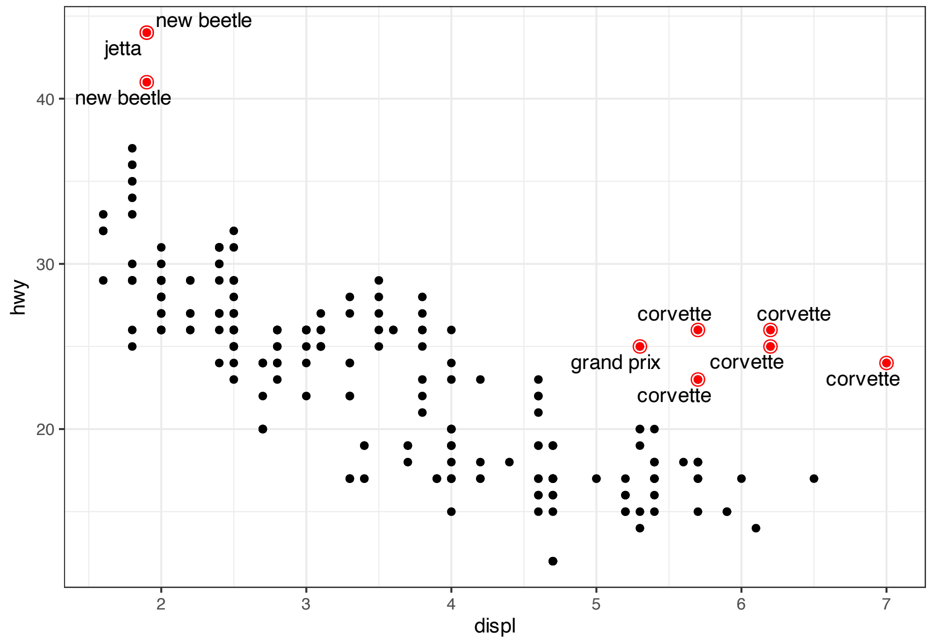
However the annotated plot we made above is hard to read because the labels overlap with each other, and with the points. We can use the `geom_label_repel()` function from the ggrepel package to address both of these issues. This useful package will automatically adjust labels so that they don't overlap:

```r
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) + geom_point(alpha = 0.3) +
    geom_smooth(se = FALSE) + geom_label_repel(data = label_info,
    aes(x = displ, y = hwy, label = drive_type), fontface = "bold",
    size = 5, nudge_y = 2) + theme(legend.position = "none")
```



You can also use the same idea to highlight certain points on a plot with `geom_text_repel()` from the ggrepel package. Note another handy technique used here: we added a second layer of large, hollow points to further highlight the labelled points.

```
potential_outliers <- mpg |>
    filter(hwy > 40 | (hwy > 20 & displ > 5))

ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + geom_text_repel(data = potential_
    aes(label = model)) + geom_point(data = potential_outliers,
    color = "red") + geom_point(data = potential_outliers, color = "red",
    size = 3, shape = "circle open")
```



Remember, in addition to `geom_text()` and `geom_label()`, you have many other geoms in ggplot2 available to help annotate your plot. A couple ideas:

- Use `geom_hline()` and `geom_vline()` to add reference lines. We often make them thick (`linewidth = 2`) and white (`color = white`), and draw them underneath the primary data layer. That makes them easy to see, without drawing attention away from the data.
- Use `geom_rect()` to draw a rectangle around points of interest. The boundaries of the rectangle are defined by aesthetics `xmin`, `xmax`, `ymin`, `ymax`.
- Use `geom_segment()` with the `arrow` argument to draw attention to a point with an arrow. Use aesthetics `x` and `y` to define the starting location, and `xend` and `yend` to define the end location.

Another handy function for adding annotations to plots is `annotate()`. As a rule of thumb, geoms are generally useful for highlighting a subset of the data while `annotate()` is useful for adding one or few annotation elements to a plot.

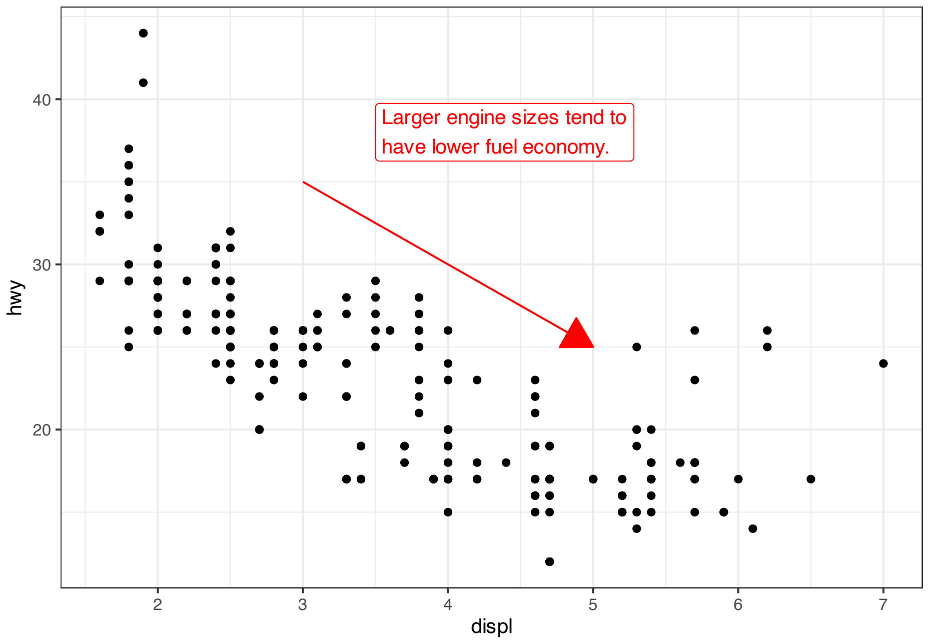
To demonstrate using `annotate()`, let's create some text to add to our plot. The text is a bit long, so we'll use `stringr::str_wrap()` to automatically add line breaks to it given the number of characters you want per line:

```
trend_text <- "Larger engine sizes tend to have lower fuel economy." |>
    str_wrap(width = 30)

trend_text
```

```
## [1] "Larger engine sizes tend to\nhave lower fuel economy."
```

Then, we add two layers of annotation: one with a label geom and the other with a segment geom. The `x` and `y` aesthetics in both define where the annotation should start, and the `xend` and `yend` aesthetics in the segment annotation define the end location of the segment. Note also that the segment is styled as an arrow.

```
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + annotate(geom = "label",
    x = 3.5, y = 38, label = trend_text, hjust = "left", color = "red") +
    annotate(geom = "segment", x = 3, y = 35, xend = 5, yend = 25,
        color = "red", arrow = arrow(type = "closed"))
```



Annotation is a powerful tool for communicating main takeaways and interesting features of your visualizations. The only limit is your imagination (and your patience with positioning annotations to be aesthetically pleasing)!

## Exercises 1.3.1

1. Use `geom_text()` with infinite positions to place text at the four corners of the plot. **Hint: construct a `data.frame` similar to `label_info` with the coordinate positions.**

**Ans-1.3.1.1:**

2. Use `annotate()` to add a point geom in the middle of the following plot without having to create a tibble. Customize the shape, size, or color of the point.

```r
ggplot(mpg, aes(displ, hwy)) + geom_point()
```

**Ans-1.3.1.2:**

3.a. How do labels with `geom_text()` interact with faceting? Try it by adding a geom_text element to:

```r
g <- ggplot(mpg, aes(displ, hwy)) + geom_point()
```

**Ans-1.3.1.3a:**

b. How can you add a label to a single facet?

**Ans-1.3.1.3b:**

c. How can you put a different label in each facet? (Hint: Think about the dataset that is being passed to `geom_text()`.)

**Ans-1.3.1.3c:**

4. What arguments to `geom_label()` control the appearance of the background box?

**Ans-1.3.1.4:**

5. What are the four arguments to `arrow()`? How do they work? Create a series of plots that demonstrate the most important options.

**Ans-1.3.1.5:**

## 1.4 Scales

The third way you can make your plot better for communication is to adjust the scales. Scales control how the aesthetic mappings manifest visually.

### 1.4.1 Default scales

Normally, ggplot2 automatically adds scales for you. For example, when you type:

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

ggplot2 automatically adds default scales behind the scenes:

```r
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = class)) +
    scale_x_continuous() + scale_y_continuous() + scale_color_discrete()
```

Note the naming scheme for scales: `scale_` followed by the name of the aesthetic, then `_`, then the name of the scale. The default scales are named according to the type of variable they align with: continuous, discrete, datetime, or date. `scale_x_continuous()` puts the numeric values from `displ` on a continuous number line on the x-axis, `scale_color_discrete()` chooses colors for each of the `class` of car, etc. There are lots of non-default scales which you'll learn about below.

The default scales have been carefully chosen to do a good job for a wide range of inputs. Nevertheless, you might want to override the defaults for two reasons:

- You might want to tweak some of the parameters of the default scale. This allows you to do things like change the breaks on the axes, or the key labels on the legend.

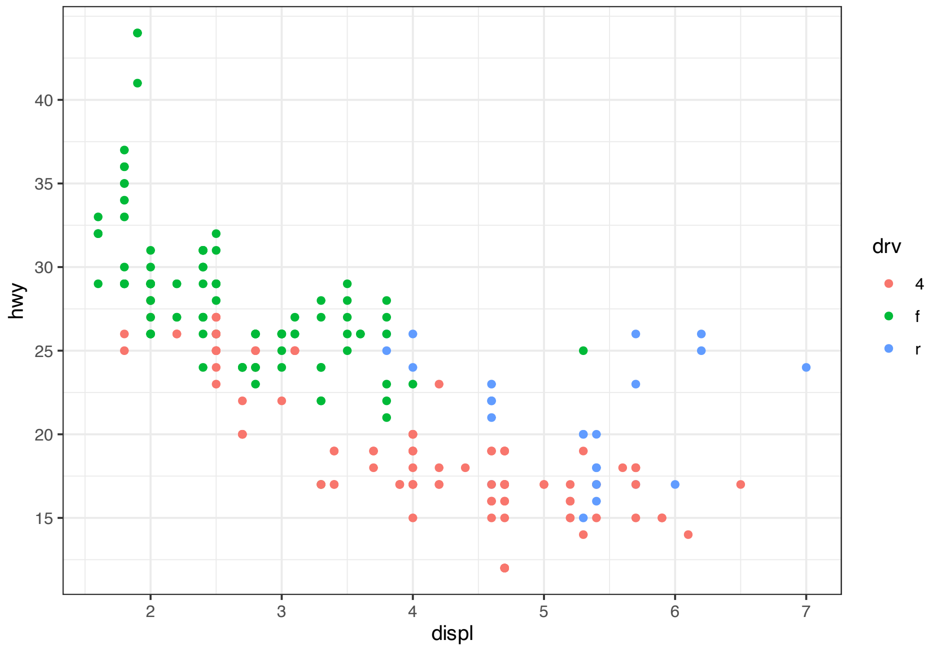
- You might want to replace the scale altogether, and use a completely different algorithm. Often you can do better than the default because you know more about the data.

### 1.4.2 Axis ticks and legend keys

Collectively axes and legends are called **guides**. Axes are used for x and y aesthetics; legends are used for everything else.

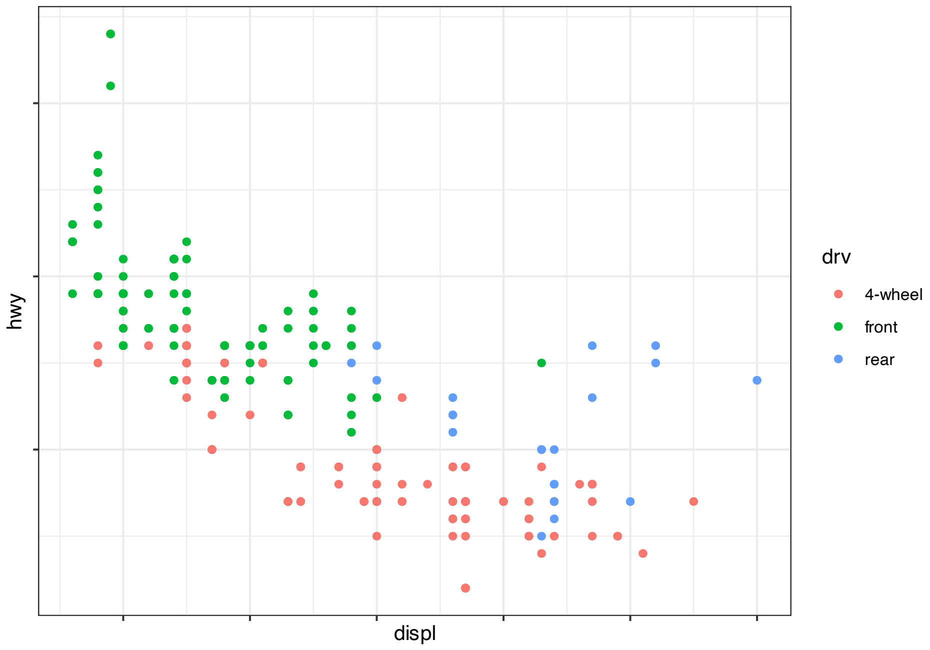
There are two primary arguments that affect the appearance of the ticks on the axes and the keys on the legend: `breaks` and `labels`. Breaks controls the position of the ticks, or the values associated with the keys. Labels controls the text label associated with each tick/key. The most common use of `breaks` is to override the default choice:

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) + geom_point() +
    scale_y_continuous(breaks = seq(15, 40, by = 5))
```



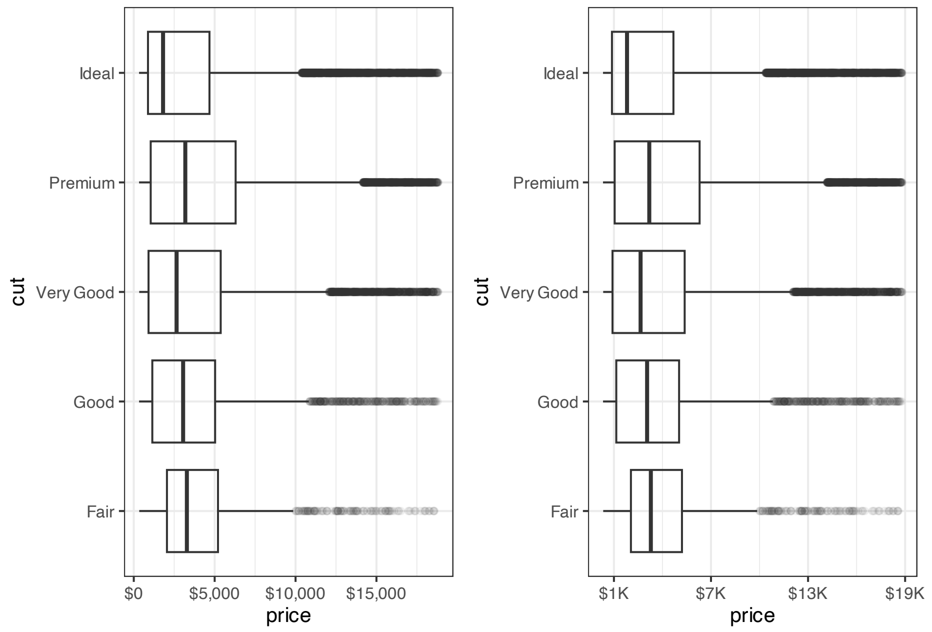
You can use `labels` in the same way (a character vector the same length as `breaks`), but you can also set it to `NULL` to suppress the labels altogether. This can be useful for maps, or for publishing plots where you can't share the absolute numbers. You can also use `breaks` and `labels` to control the appearance of legends. For discrete scales for categorical variables, `labels` can be a named list of the existing levels names and the desired labels for them.

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) + geom_point() +
    scale_x_continuous(labels = NULL) + scale_y_continuous(labels = NULL) +
    scale_color_discrete(labels = c(`4` = "4-wheel", f = "front",
        r = "rear"))
```



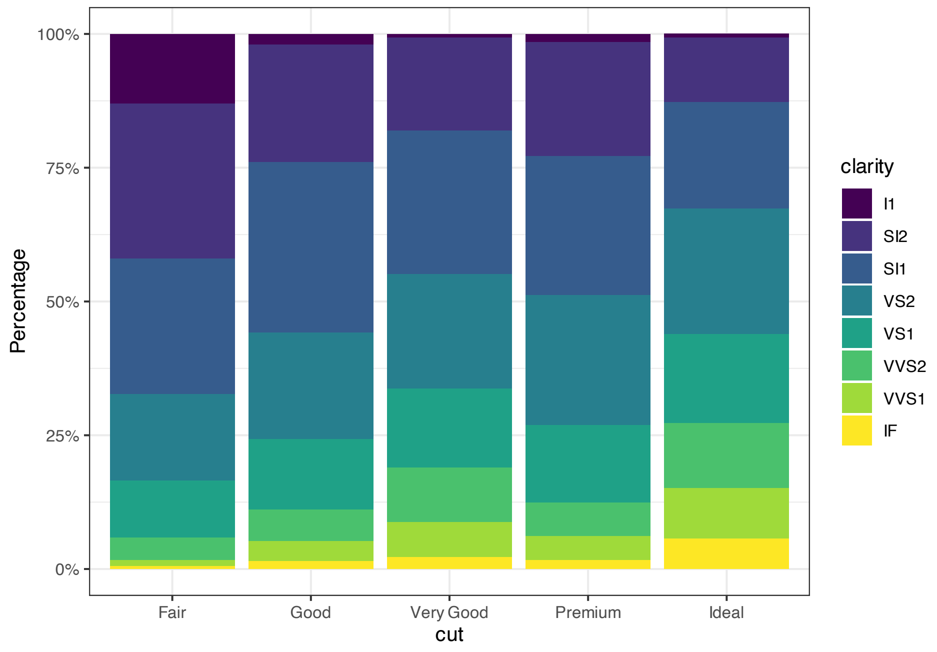
The `labels` argument coupled with labeling functions from the scales package is also useful for formatting numbers as currency, percent, etc. The plot on the left shows default labeling with `label_dollar()` from the `scales` package, which adds a dollar sign as well as a thousand separator comma. The plot on the right adds further customization by dividing dollar values by 1,000 and adding a suffix “K” (for “thousands”) as well as adding custom breaks. Note that `breaks` is in the original scale of the data.

```
# | fig-width: 4 | fig-alt: | | Two side-by-side box plots
# of price versus cut of diamonds. The outliers | are
# transparent. On both plots the x-axis labels are
# formatted as dollars. | The x-axis labels on the plot
# start at $0 and go to $15,000, increasing | by $5,000.
# The x-axis labels on the right plot start at $1K and go
# to | $19K, increasing by $6K.

# Left-# Right
ggplot(diamonds, aes(x = price, y = cut)) + geom_boxplot(alpha = 0.05) +
    scale_x_continuous(labels = label_dollar()) + ggplot(diamonds,
    aes(x = price, y = cut)) + geom_boxplot(alpha = 0.05) + scale_x_continuous(labels
    suffix = "K"), breaks = seq(1000, 19000, by = 6000))
```



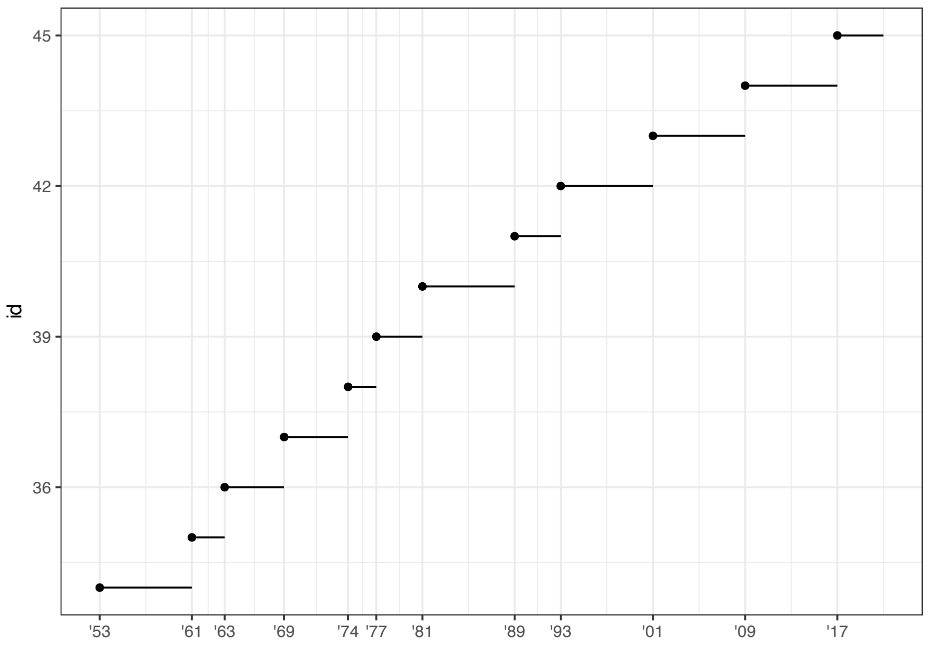
Another handy label function is `label_percent()`:

```r
ggplot(diamonds, aes(x = cut, fill = clarity)) + geom_bar(position = "fill") +
    scale_y_continuous(name = "Percentage", labels = label_percent())
```



Another use of `breaks` is when you have relatively few data points and want to highlight exactly where the observations occur. For example, take this plot that shows when each US president started and ended their term.

```
# here `id` is the ordinal number of the president, e.g.
# Trump is in the 12th row + 33 = 45th president
presidential |>
    mutate(id = 33 + row_number()) |>
    ggplot(aes(x = start, y = id)) + geom_point() + geom_segment(aes(xend = end,
    yend = id)) + scale_x_date(name = NULL, breaks = presidential$start,
    date_labels = "'%y")
```



Note that for the `breaks` argument we pulled out the `start` variable as a vector with `presidential$start` because we can't do an aesthetic mapping for this argument. Also note that the specification of breaks and labels for date and datetime scales is a little different:

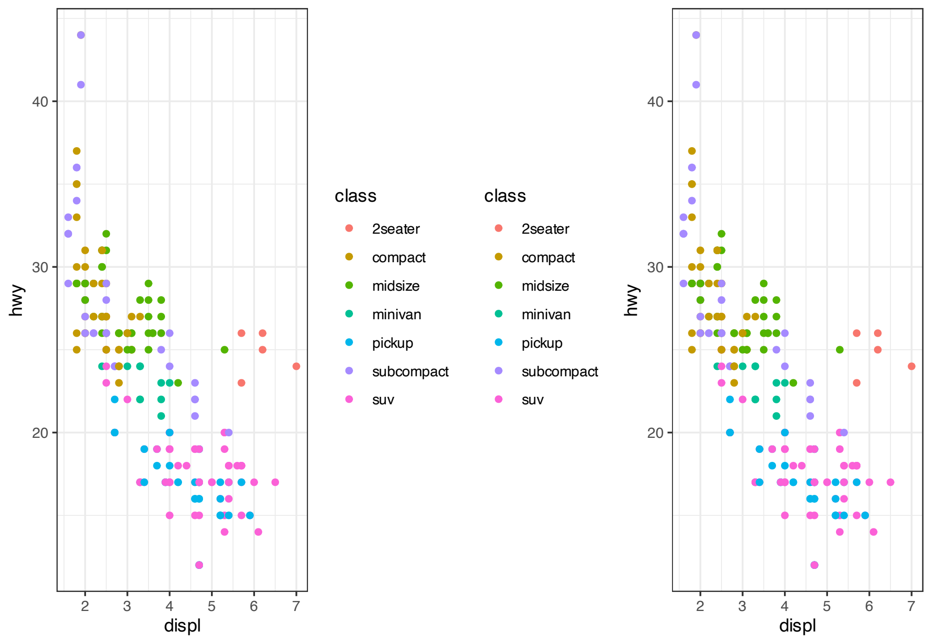
- `date_labels` takes a format specification, in the same form as `parse_datetime()`.
- `date_breaks` (not shown here), takes a string like "2 days" or "1 month".

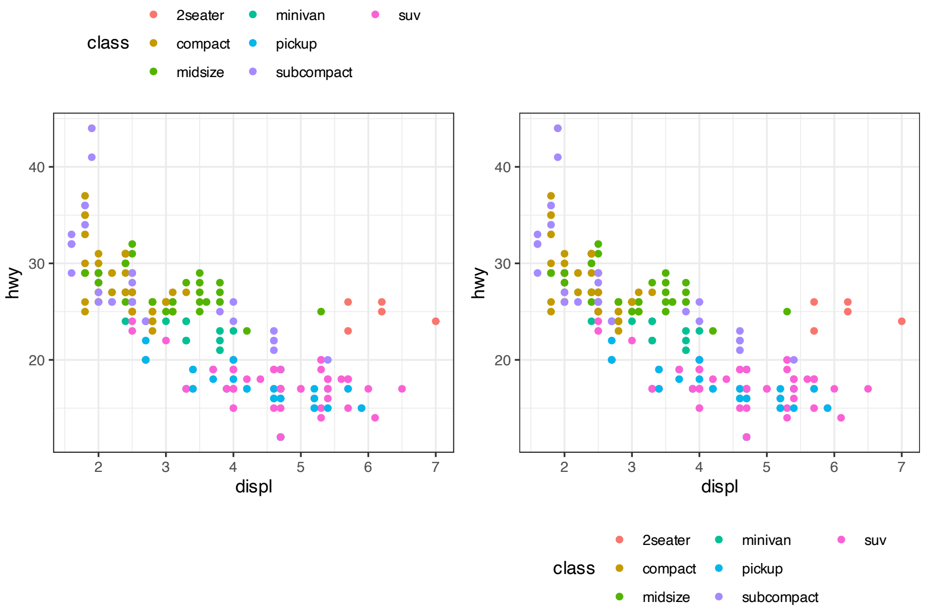
### 1.4.3 Legend layout

You will most often use `breaks` and `labels` to tweak the axes. While they both also work for legends, there are a few other techniques you are more likely to use.

To control the overall position of the legend, you need to use a `theme()` setting. We'll come back to themes at the end of the chapter, but in brief, they control the non-data parts of the plot. The theme setting `legend.position` controls where the legend is drawn:

```r
base <- ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = class))

# the default
base + theme(legend.position = "right") + base + theme(legend.position = "left")

base + theme(legend.position = "top") + guides(color = guide_legend(nrow = 3)) +
    base + theme(legend.position = "bottom") + guides(color = guide_legend(nrow = 3))
```

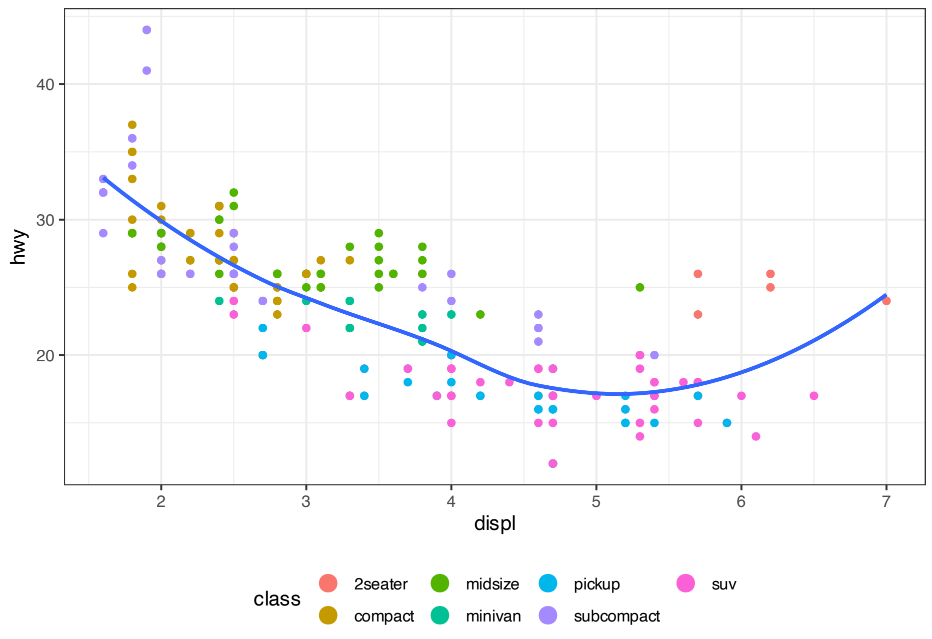




If your plot is short and wide, place the legend at the top or bottom, and if it's tall and narrow, place the legend at the left or right. You can also use `legend.position = "none"` to suppress the display of the legend altogether.

To control the display of individual legends, use `guides()` along with `guide_legend()` or `guide_colorbar()`. The following example shows two important settings: controlling the number of rows the legend uses with `nrow`, and overriding one of the aesthetics to make the points bigger. This is particularly useful if you have used a low `alpha` to display many points on a plot.

```
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = class)) +
    geom_smooth(se = FALSE) + theme(legend.position = "bottom") +
    guides(color = guide_legend(nrow = 2, override.aes = list(size = 4)))
```



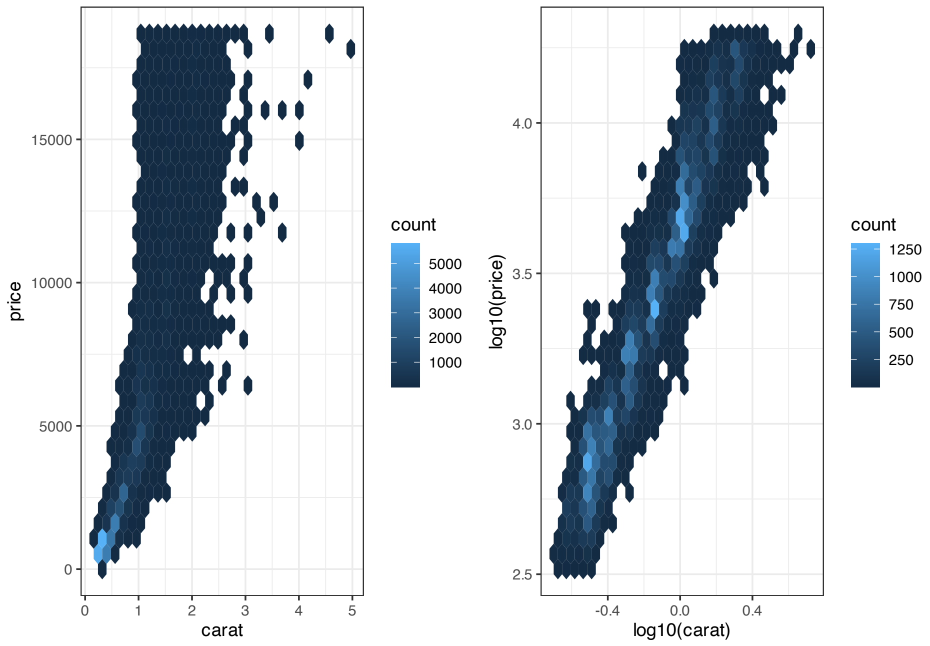
Note that the name of the argument in `guides()` matches the name of the aesthetic, just like in `labs()`.

### 1.4.4 Replacing a scale

Instead of just tweaking the details a little, you can instead replace the scale altogether. There are two types of scales you're mostly likely to want to switch out: continuous position scales and color scales. Fortunately, the same principles apply to all the other aesthetics, so once you've mastered position and color, you'll be able to quickly pick up other scale replacements.

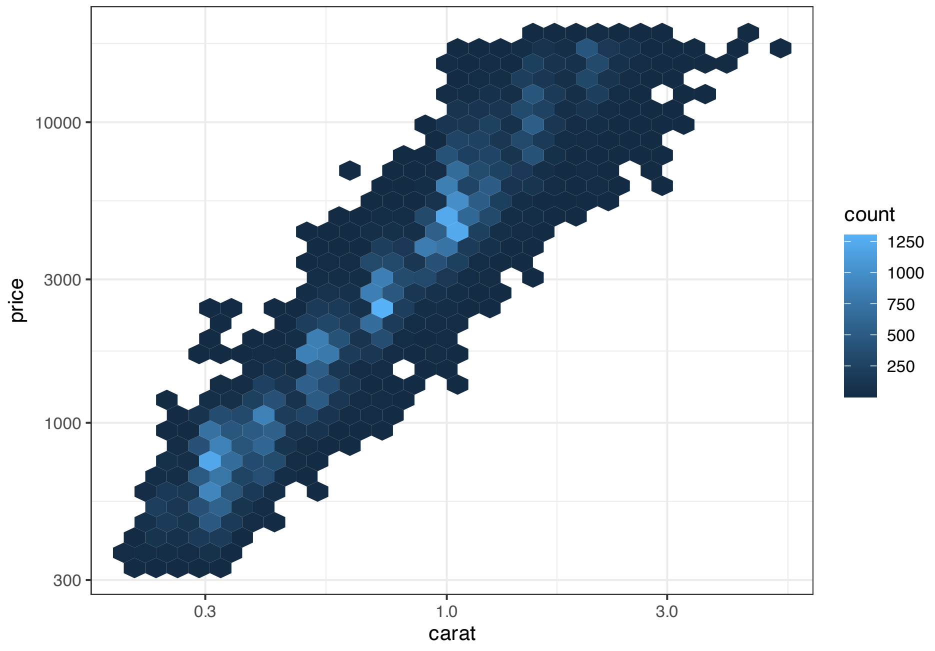
It's very useful to plot transformations of your variable. For example, it's easier to see the precise relationship between `carat` and `price` if we log transform them:

```
# Left-# Right
ggplot(diamonds, aes(x = carat, y = price)) + geom_hex() + ggplot(diamonds,
    aes(x = log10(carat), y = log10(price))) + geom_hex()
```



However, the disadvantage of this transformation is that the axes are now labelled with the transformed values, making it hard to interpret the plot. Instead of doing the transformation in the aesthetic mapping, we can instead do it with the scale. This is visually identical, except the axes are labelled on the original data scale.

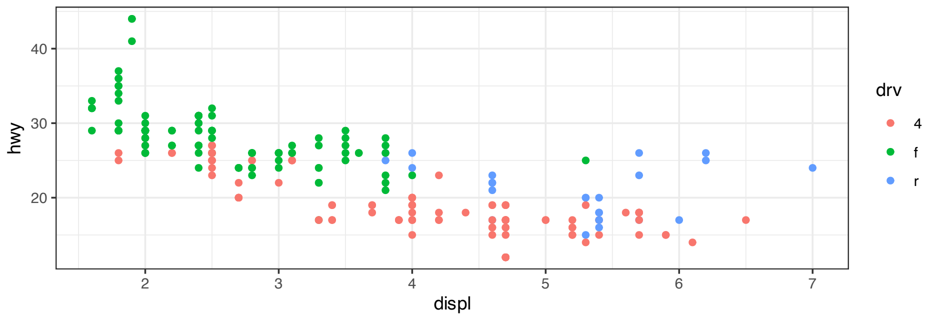
```
ggplot(diamonds, aes(x = carat, y = price)) + geom_hex() + scale_x_log10() +
    scale_y_log10()
```

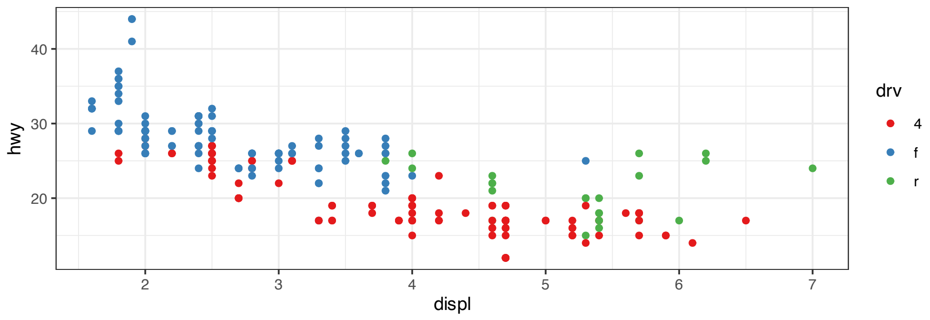


Another scale that is frequently customized is color. The default categorical scale picks colors that are evenly spaced around the color wheel. Useful alternatives are the ColorBrewer scales which have been hand tuned to work better for people with common types of color blindness. The two plots below look similar, but there is enough difference in the shades of red and green that the dots on the right can be distinguished even by people with red-green color blindness.[1]

```
(ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = drv)))/(ggplot(mpg,
    aes(x = displ, y = hwy)) + geom_point(aes(color = drv)) +
    scale_color_brewer(palette = "Set1"))
```

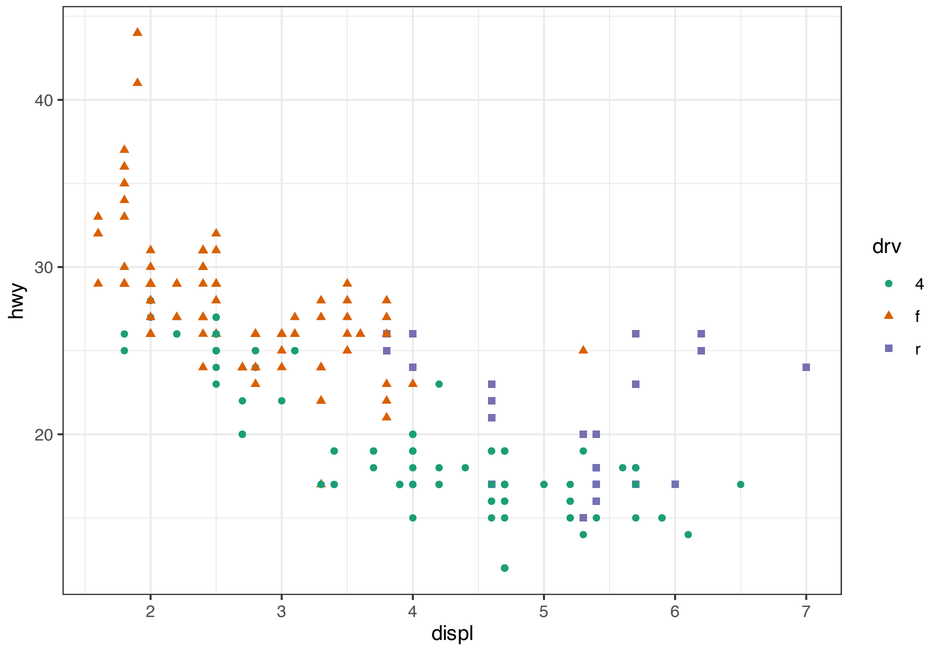
[1]You can use a tool like SimDaltonism to simulate color blindness to test these images.





Don't forget simpler techniques for improving accessibility. If there are just a few colors, you can add a redundant shape mapping. This will also help ensure your plot is interpretable in black and white.

```r
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = drv,
    shape = drv)) + scale_color_brewer(palette = "Dark2")
```



The ColorBrewer scales are documented online at https://colorbrewer2.org/ and made available in R via the **RColorBrewer** package, by Erich Neuwirth. 1.2 shows the complete list of all palettes. The sequential (top) and diverging (bottom) palettes are particularly useful if your categorical values are ordered, or have a "middle". This often arises if you've used `cut()` to make a continuous variable into a categorical variable.

When you have a predefined mapping between values and colors, use `scale_color_manual()`. For example, if we map presidential party to color, we want to use the standard mapping of red for Republicans and blue for Democrats. One approach for assigning these colors is using hex color codes:

```
presidential |>
    mutate(id = 33 + row_number()) |>
    ggplot(aes(x = start, y = id, color = party)) + geom_point() +
    geom_segment(aes(xend = end, yend = id)) + scale_color_manual(values = c(Republican = "#E81B2
    Democratic = "#00AEF3"))
```

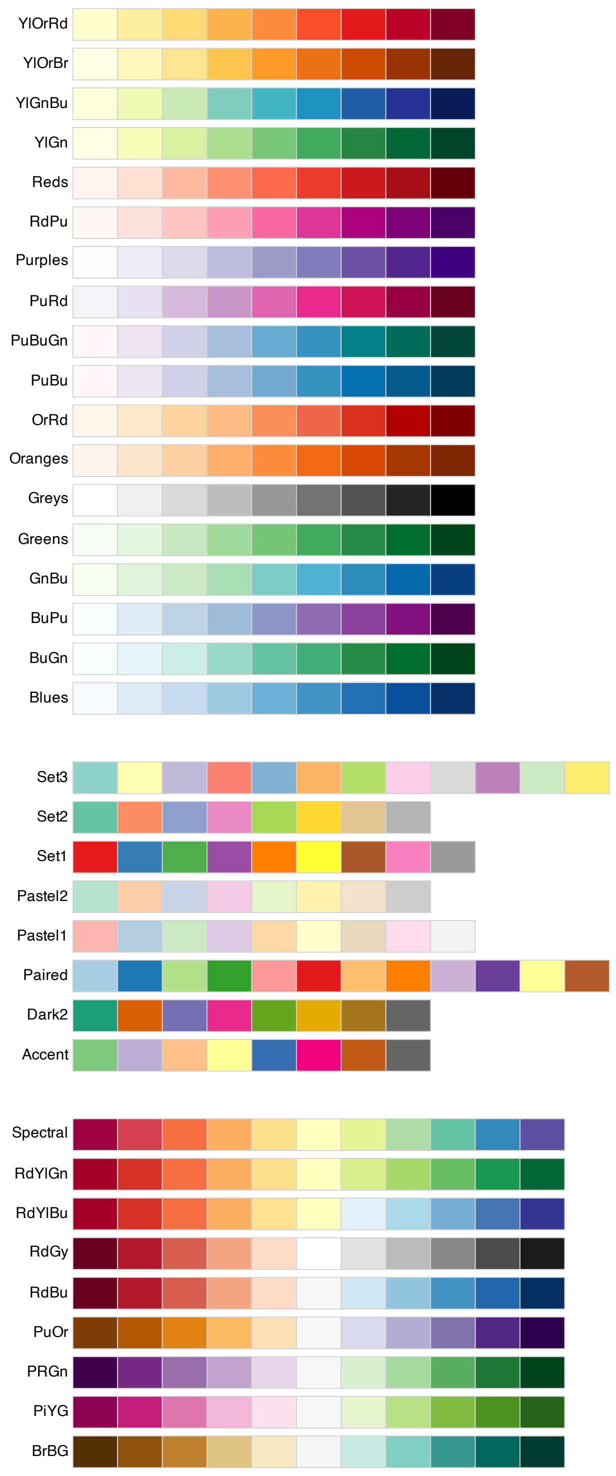
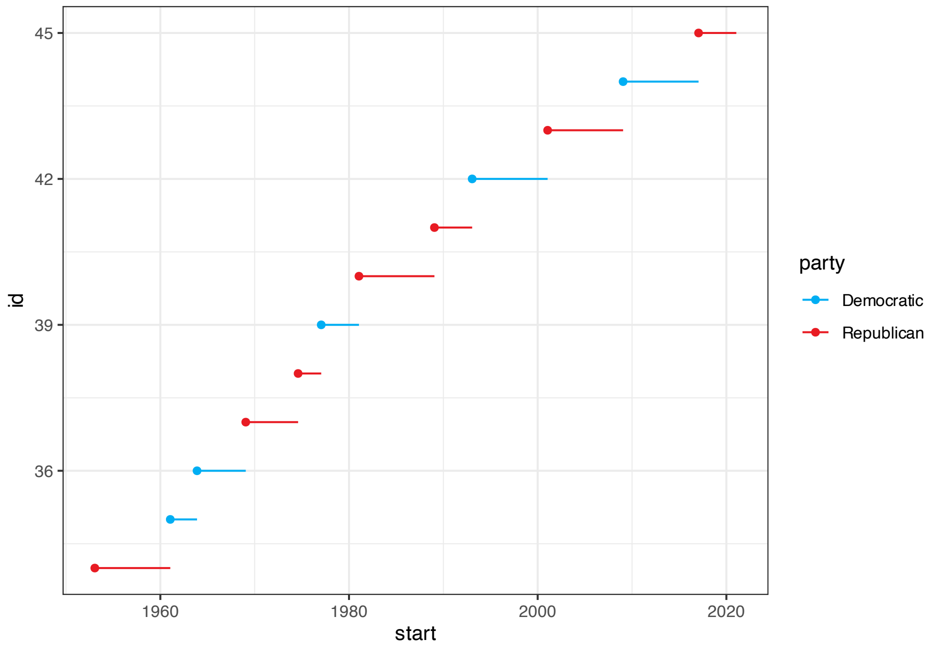


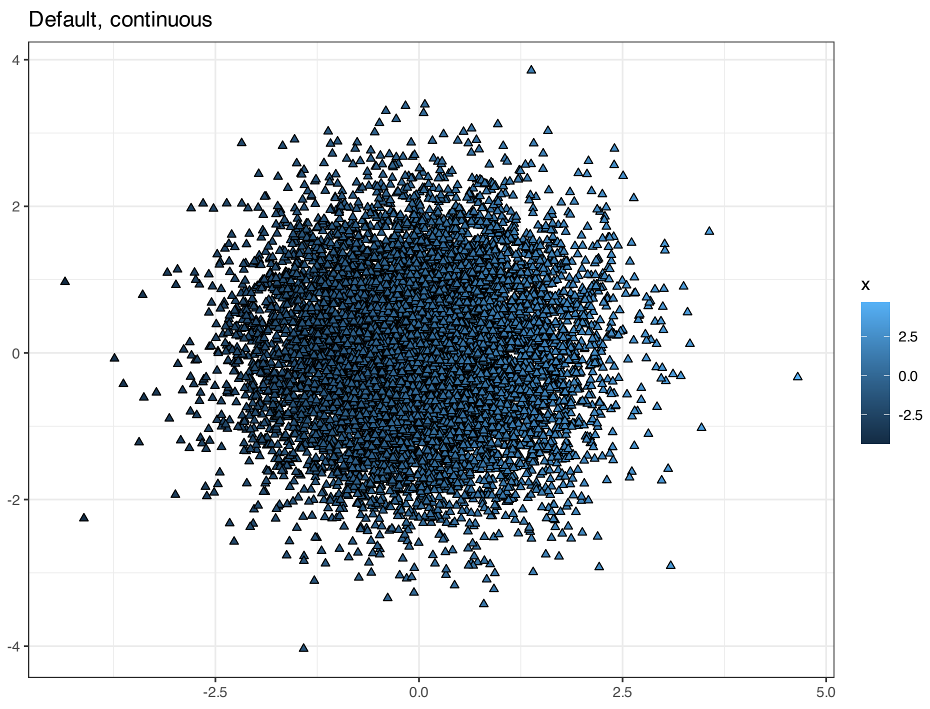
Figure 1.2: All colorBrewer scales.

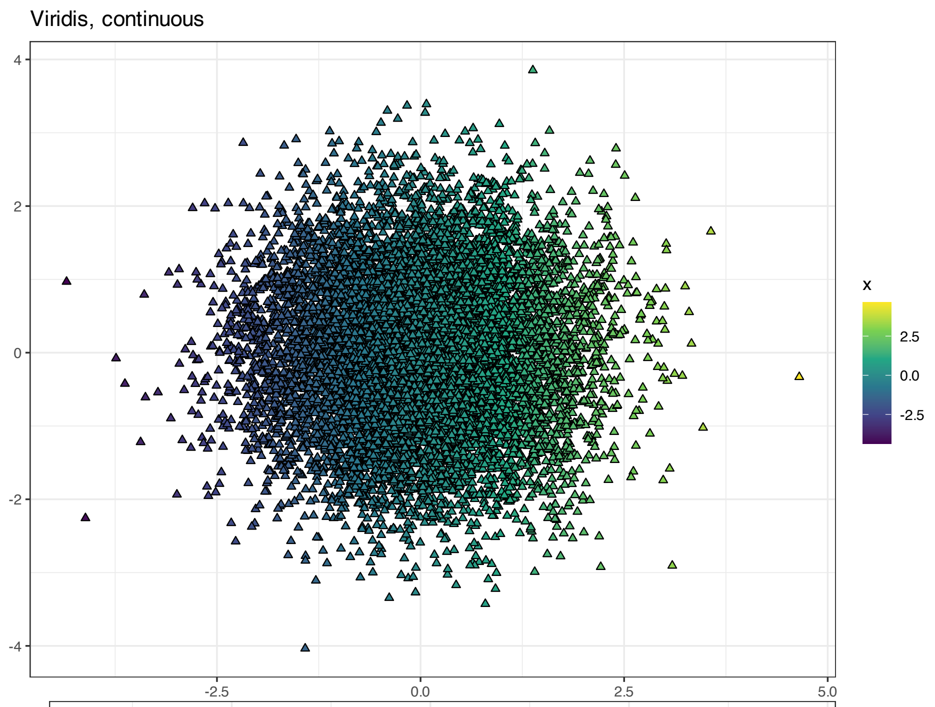


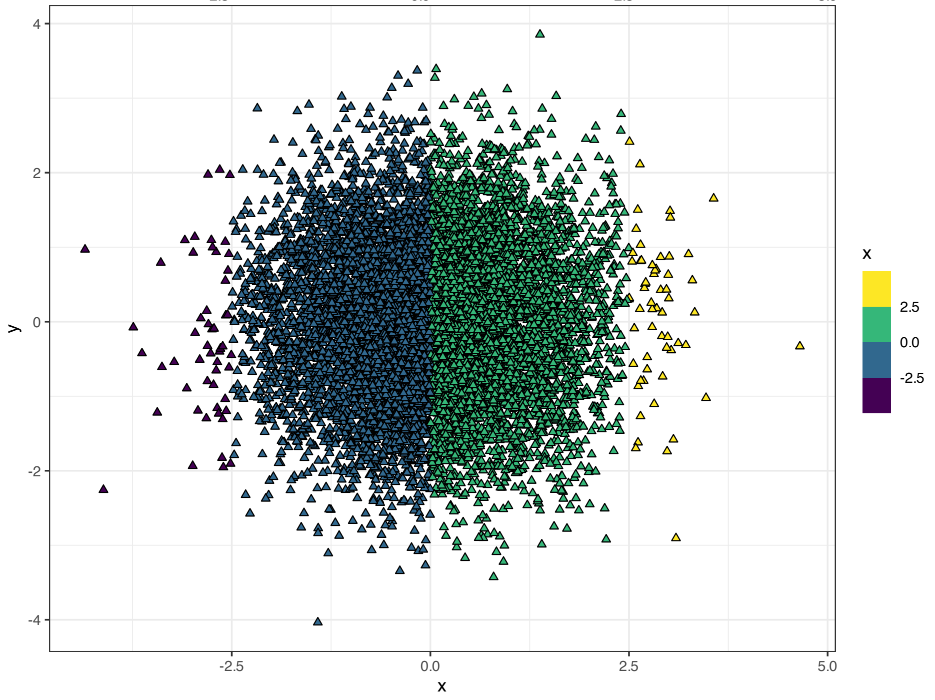
For continuous color, you can use the built-in `scale_color_gradient()` or `scale_fill_gradient()`. If you have a diverging scale, you can use `scale_color_gradient2()`. That allows you to give, for example, positive and negative values different colors. That's sometimes also useful if you want to distinguish points above or below the mean.

Another option is to use the viridis color scales. The designers, Nathaniel Smith and Stéfan van der Walt, carefully tailored continuous color schemes that are perceptible to people with various forms of color blindness as well as perceptually uniform in both color and black and white. These scales are available as continuous (`c`), discrete (`d`), and binned (`b`) palettes in ggplot2.

```r
df <- tibble(x = rnorm(10000), y = rnorm(10000))

ggplot(df, aes(x, y, fill = x)) + geom_point(shape = 24) + labs(title = "Default, continuous",
    x = NULL, y = NULL)

ggplot(df, aes(x, y, fill = x)) + geom_point(shape = 24) + scale_fill_viridis_c() +
    labs(title = "Viridis, continuous", x = NULL, y = NULL)

ggplot(df, aes(x, y, fill = x)) + geom_point(shape = 24) + scale_fill_viridis_b()
labs(title = "Viridis, binned", x = NULL, y = NULL)
## $x
## NULL
##
## $y
## NULL
##
## $title
## [1] "Viridis, binned"
##
## attr(,"class")
## [1] "labels"
```







Note that all color scales come in two varieties: `scale_color_*()` and `scale_fill_*()` for the `color` and `fill` aesthetics respectively (the color scales are available in both UK and US spellings).

### 1.4.5 Zooming

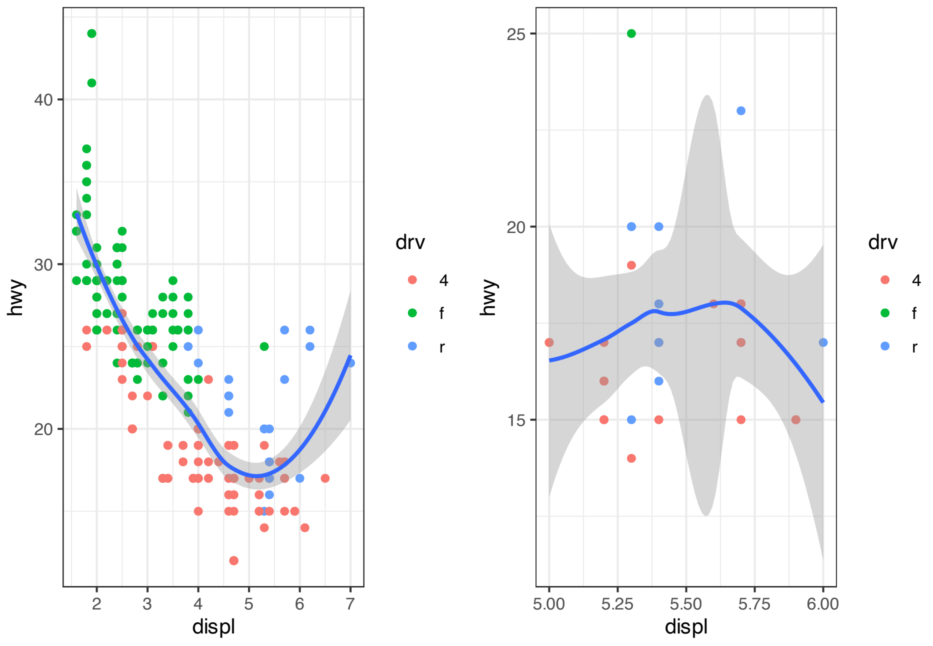
There are three ways to control the plot limits:

1. Adjusting what data are plotted.
2. Setting the limits in each scale.
3. Setting `xlim` and `ylim` in `coord_cartesian()`.

We'll demonstrate these options in a series of plots. The plot on the left shows the relationship between engine size and fuel efficiency, colored by type of drive train. The plot on the right shows the same variables, but subsets the data that are plotted. Subsetting the data has affected the x and y scales as well as the smooth curve.

1. Adjusting what data are plotted:

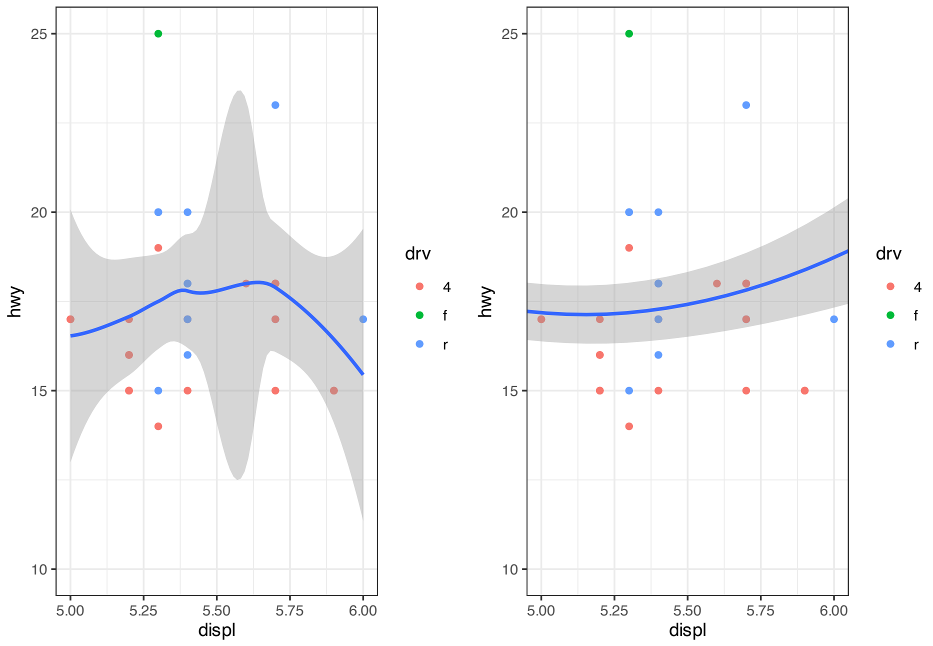
```
# | fig-width: 4

# | fig-alt: | | On the left, scatterplot of highway
# mileage vs. displacement, with | displacement. The smooth
# curve overlaid shows a decreasing, and then | increasing
# trend, like a hockey stick. On the right, same variables
# | are plotted with displacement ranging only from 5 to 6
# and highway | mileage ranging only from 10 to 25. The
# smooth curve overlaid shows a | trend that's slightly
# increasing first and then decreasing.

# Left-# Right
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = drv)) +
    geom_smooth() + mpg |>
    filter(displ >= 5 & displ <= 6 & hwy >= 10 & hwy <= 25) |>
    ggplot(aes(x = displ, y = hwy)) + geom_point(aes(color = drv)) +
    geom_smooth()
```



Let's compare these to the two plots below where the plot on the left sets the `limits` on individual scales and the plot on the right sets them in `coord_cartesian()`. We can see that reducing the limits is equivalent to subsetting the data. Therefore, to zoom in on a region of the plot, it's generally best to use `coord_cartesian()`.

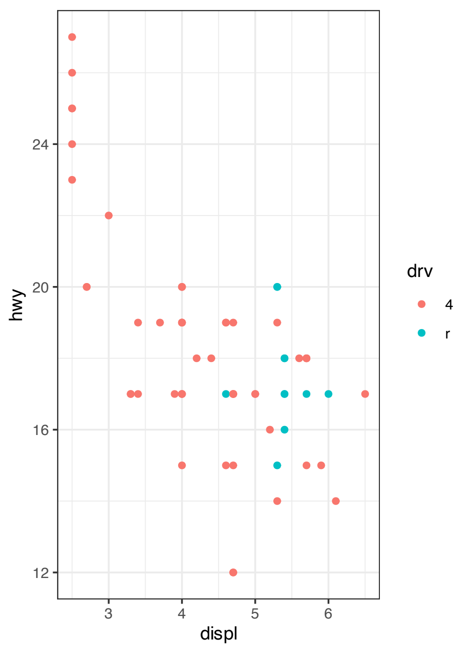
2. Setting the limits in each scale (left)
3. Setting `xlim` and `ylim` in `coord_cartesian()` (right).

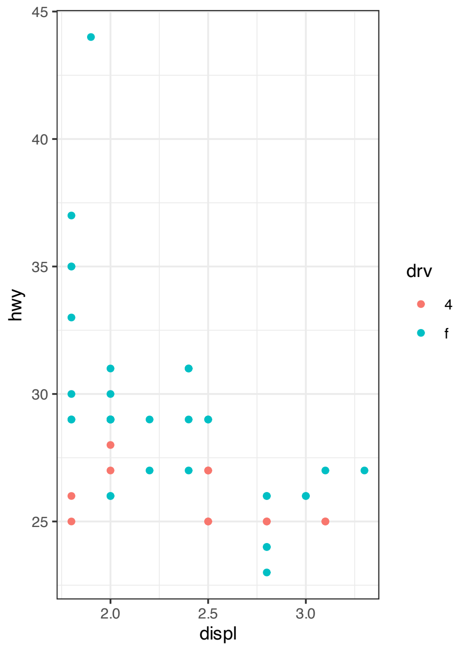
```
# | fig-alt: | | On the left, scatterplot of highway
# mileage vs. displacement, with | displacement ranging
# from 5 to 6 and highway mileage ranging from | 10 to 25.
# The smooth curve overlaid shows a trend that's slightly |
# increasing first and then decreasing. On the right, same
# variables | are plotted with the same limits, however the
# smooth curve overlaid | shows a relatively flat trend
# with a slight increase at the end.

# Left-# Right
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = drv)) +
    geom_smooth() + scale_x_continuous(limits = c(5, 6)) + scale_y_continuous(limits = c(10,
    25)) + ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = drv)) +
    geom_smooth() + coord_cartesian(xlim = c(5, 6), ylim = c(10,
    25))
```



On the other hand, setting the `limits` on individual scales is generally more useful if you want to *expand* the limits, e.g., to match scales across different plots. For example, if we extract two classes of cars and plot them separately, it's difficult to compare the plots because all three scales (the x-axis, the y-axis, and the color aesthetic) have different ranges.

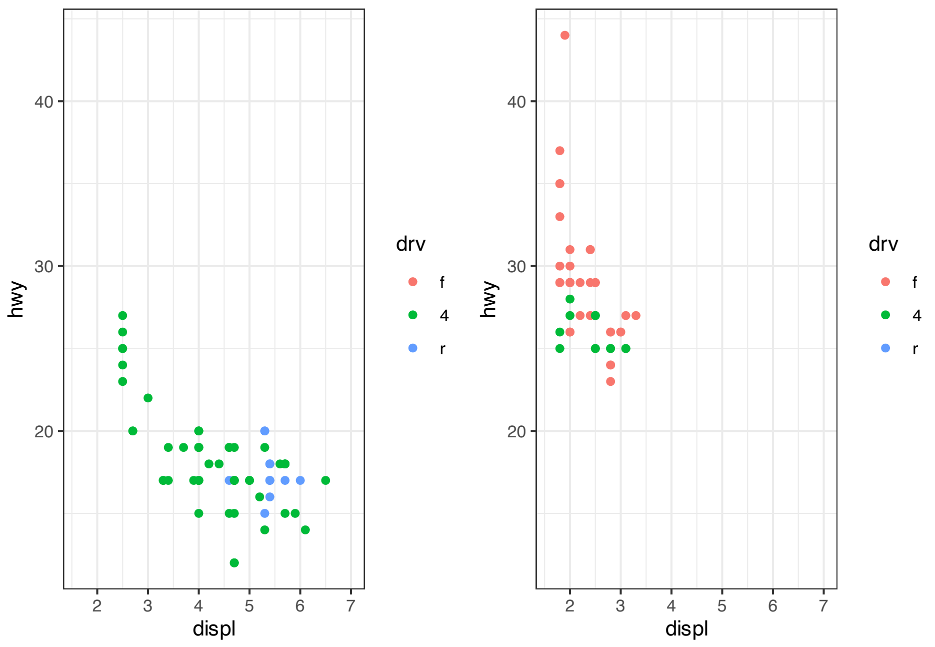
```r
suv <- mpg |>
    filter(class == "suv")
compact <- mpg |>
    filter(class == "compact")

# Left-# Right
ggplot(suv, aes(x = displ, y = hwy, color = drv)) + geom_point() +
    ggplot(compact, aes(x = displ, y = hwy, color = drv)) + geom_point()
```





One way to overcome this problem is to share scales across multiple plots, training the scales with the `limits` of the full data.

```
# | fig-width: 8 | fig-alt: | | On the left, a scatterplot
# of highway mileage vs. displacement of SUVs. | On the
# right, a scatterplot of the same variables for compact
# cars. | Points are colored by drive type for both plots.
# Both plots are plotted | on the same scale for highway
# mileage, displacement, and drive type, | resulting in the
# legend showing all three types (front, rear, and 4-wheel
# | drive) for both plots even though there are no
# front-wheel drive SUVs and | no rear-wheel drive compact
# cars. Since the x and y scales are the same, | and go
# well beyond minimum or maximum highway mileage and
# displacement, | the points do not take up the entire
# plotting area.

x_scale <- scale_x_continuous(limits = range(mpg$displ))
y_scale <- scale_y_continuous(limits = range(mpg$hwy))
col_scale <- scale_color_discrete(limits = unique(mpg$drv))

# Left-# Right
ggplot(suv, aes(x = displ, y = hwy, color = drv)) + geom_point() +
    x_scale + y_scale + col_scale + ggplot(compact, aes(x = displ,
    y = hwy, color = drv)) + geom_point() + x_scale + y_scale +
    col_scale
```

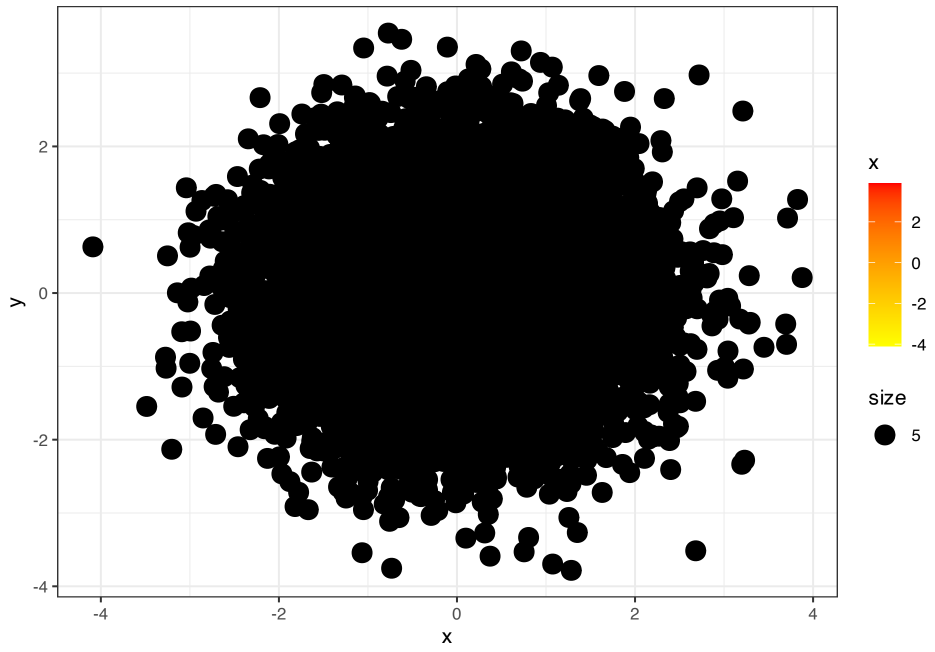


In this particular case, you could have simply used faceting, but this technique is useful more generally, if for instance, you want to spread plots over multiple pages of a report.

## Exercises 1.4.6

1. Why doesn't the following code override the default scale? Fix the plot so that it colors the points with a gradient set by `low = "yellow", high = "red"`.

```
# | fig-show: 'hide'

df <- tibble(x = rnorm(10000), y = rnorm(10000))

ggplot(df) + geom_point(data = df, aes(x = x, y = y, fill = x,
    size = 5)) + scale_fill_gradient(low = "yellow", high = "red")
```



**Ans-1.4.1.1:**

2. What is the first argument to every scale? How does it compare to `labs()`?

**Ans-1.4.1.2:**

3. First, create the following plot. Then, modify the code using `override.aes` to make the legend easier to see.

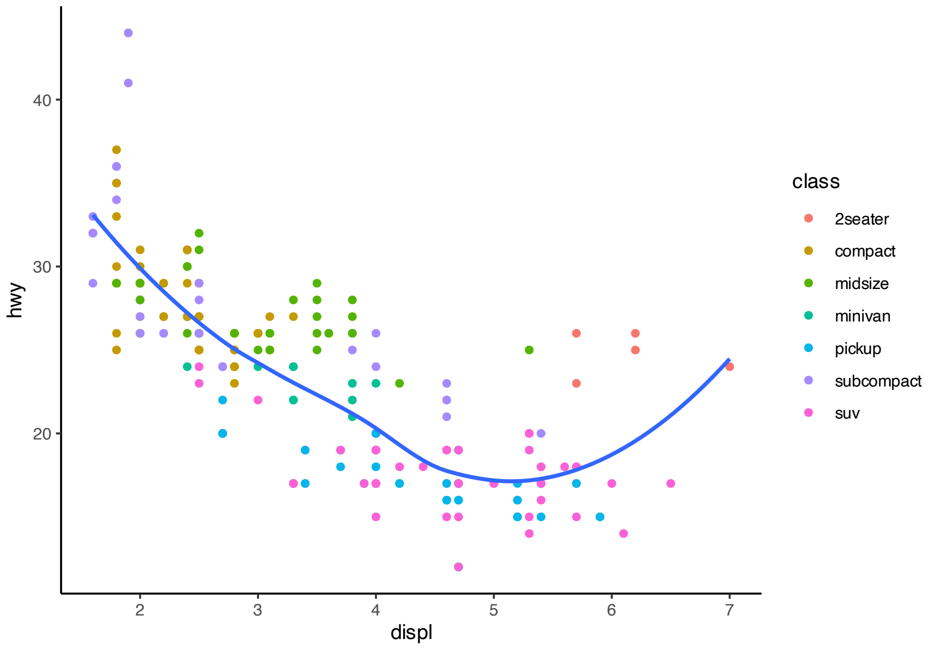
```
ggplot(diamonds, aes(x = carat, y = price)) + geom_point(aes(color = cut),
    alpha = 1/20)
```

**Ans-1.4.1.3:**

## 1.5 Themes

Finally, you can customize the non-data elements of your plot with a theme:

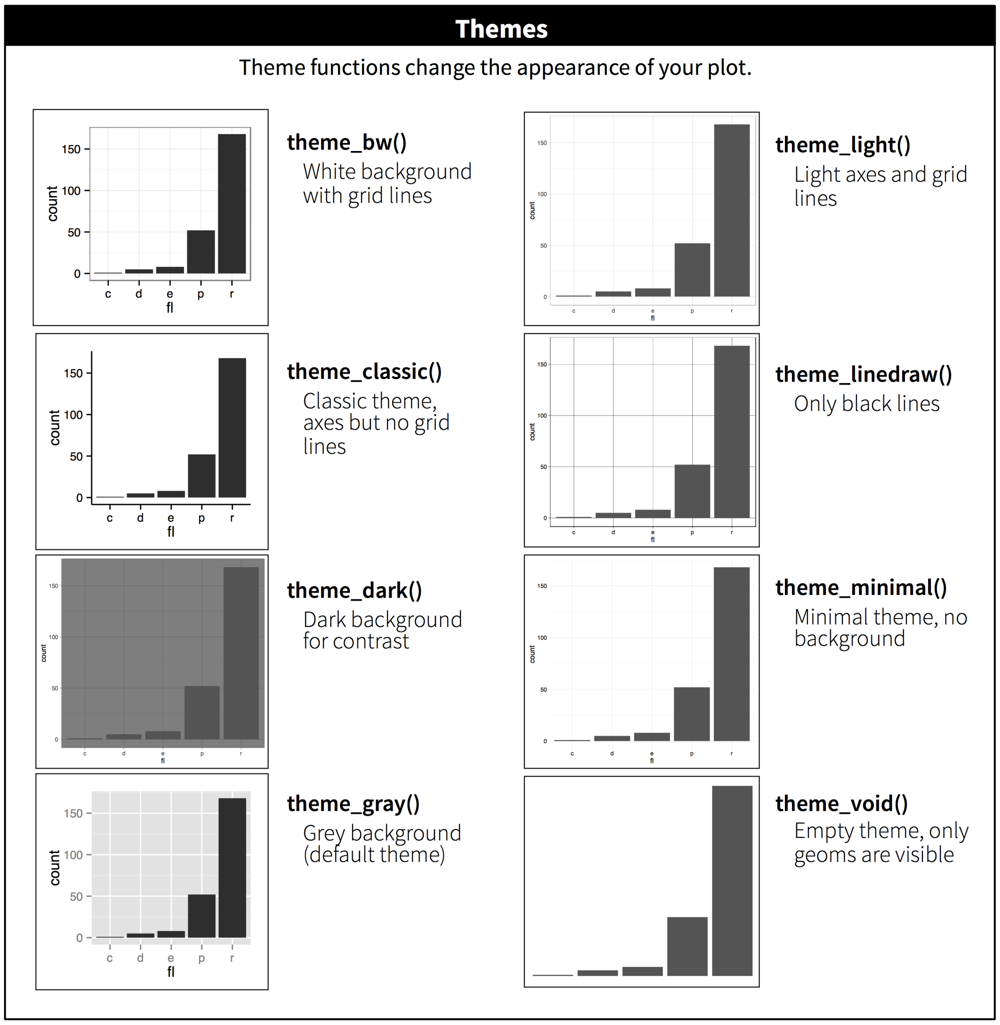
```
# | fig-alt: | | Scatterplot of highway mileage vs.
# displacement of cars, colored by class | of car. The plot
# background is white, with gray grid lines.

ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(aes(color = class)) +
    geom_smooth(se = FALSE) + theme_classic()
```



ggplot2 includes the eight themes shown in 1.3, with `theme_gray()` as the default.[^communication-2] Many more are included in add-on packages like **ggthemes** (https://jrnold.github.io/ggthemes), by Jeffrey Arnold. You can also create your own themes, if you are trying to match a particular corporate or journal style.

Many people wonder why the default theme has a gray background. This was a deliberate choice because it puts the data forward while still making the grid lines visible. The white grid lines are visible (which is important because they significantly aid position judgments), but they have little visual impact and we can easily tune them out. The gray background gives the plot a similar typographic color to the text, ensuring that the graphics fit in with the flow of a document without jumping out with a bright white background. Finally, the gray background creates a continuous field of color which ensures that the plot is perceived as a single visual entity.

It's also possible to control individual components of each theme, like the size and color of the font used for the y axis. We've already seen that `legend.position` controls where the legend is drawn. There are many other aspects of the legend that can be customized with `theme()`. For example, in the plot below we change the direction of the legend as well as put a black border around it. Note that customization of the legend box and plot title elements of the theme are done with `element_*()` functions. These functions specify the styling of non-data components, e.g., the title text is bolded in the `face` argument of `element_text()` and the legend border color is defined in the `color` argument of `element_rect()`. The theme elements that control



**Themes**

Theme functions change the appearance of your plot.

**theme_bw()**
White background with grid lines

**theme_light()**
Light axes and grid lines

**theme_classic()**
Classic theme, axes but no grid lines

**theme_linedraw()**
Only black lines

**theme_dark()**
Dark background for contrast

**theme_minimal()**
Minimal theme, no background

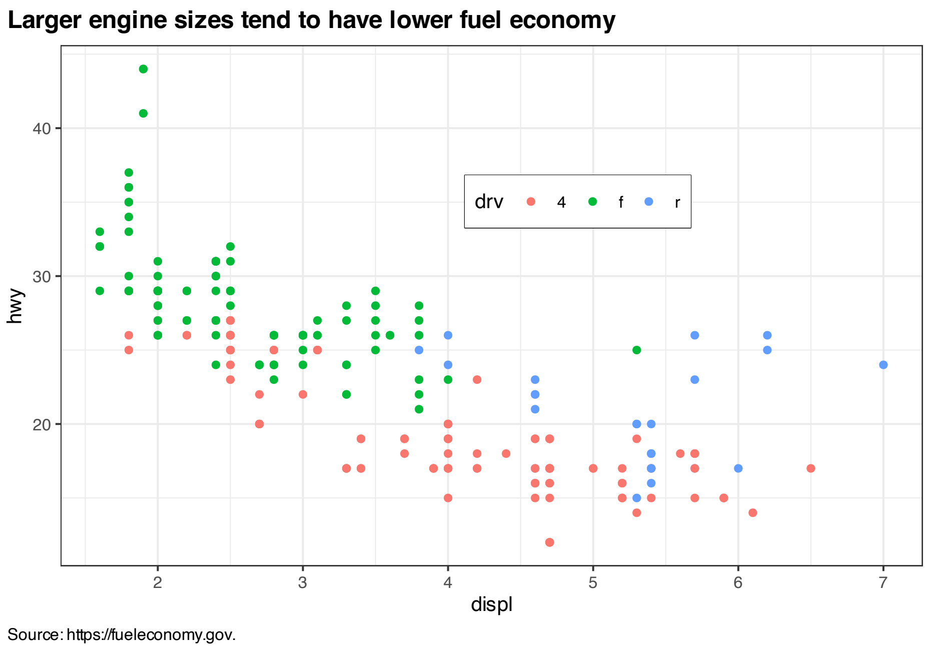
**theme_gray()**
Grey background (default theme)

**theme_void()**
Empty theme, only geoms are visible

Figure 1.3: The eight themes built-in to ggplot2.

the position of the title and the caption are `plot.title.position` and `plot.caption.position`, respectively. In the following plot these are set to `"plot"` to indicate these elements are aligned to the entire plot area, instead of the plot panel (the default). A few other helpful `theme()` components are used to change the placement for format of the title and caption text.

```r
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) + geom_point() +
    labs(title = "Larger engine sizes tend to have lower fuel economy",
        caption = "Source: https://fueleconomy.gov.") + theme(legend.position = c(0.6,
    0.7), legend.direction = "horizontal", legend.box.background = element_rect(color =
    plot.title = element_text(face = "bold"), plot.title.position = "plot",
    plot.caption.position = "plot", plot.caption = element_text(hjust = 0))
```



For an overview of all `theme()` components, see help with `?theme`. The ggplot2 book is also a great place to go for the full details on theming.

## Exercises 1.5.1

1. Pick a theme offered by the ggthemes package and apply it to this plot:

```r
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) + geom_point() +
    labs(title = "Larger engine sizes tend to have lower fuel economy",
        caption = "Source: https://fueleconomy.gov.")
```

**Ans-1.5.1.1:**

2. Make the axis labels of your plot blue and bolded.

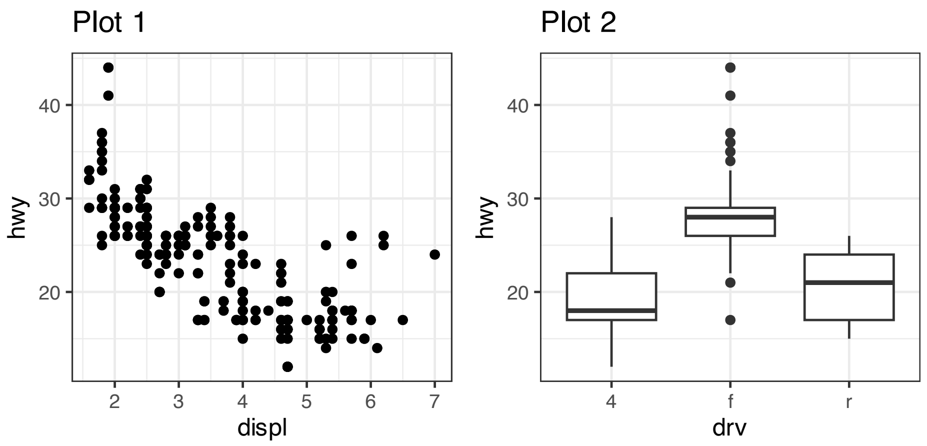
**Ans-1.5.1.2:**

## 1.6 Layout

So far we talked about how to create and modify a single plot. What if you have multiple plots you want to lay out in a certain way? The patchwork package allows you to combine separate plots into the same graphic. We loaded this package earlier in the chapter.

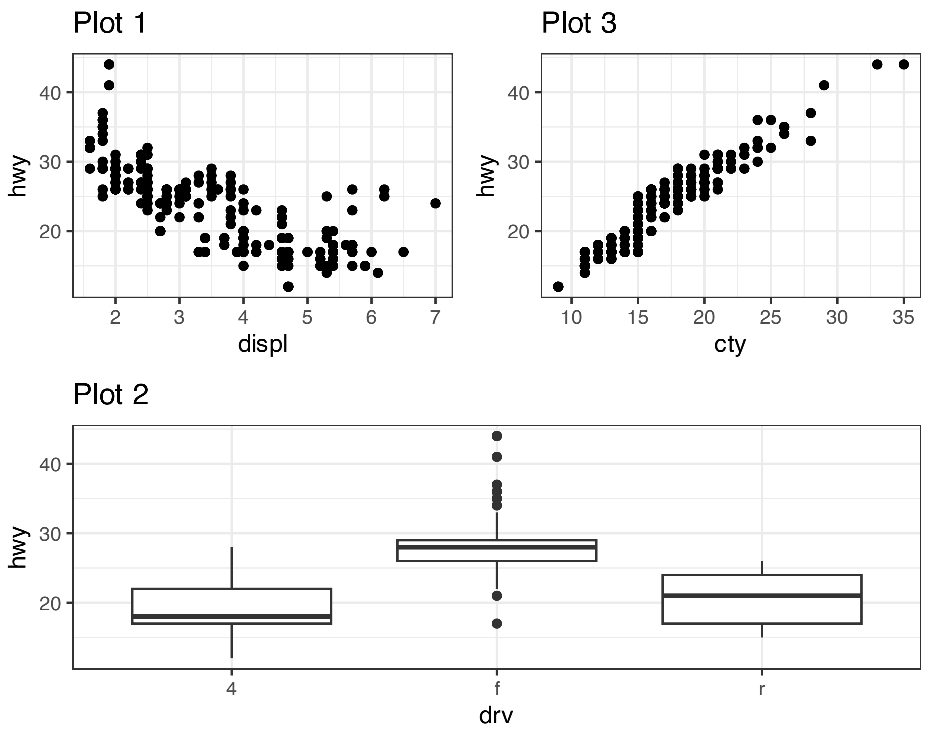
To place two plots next to each other, you can simply add them to each other. Note that you first need to create the plots and save them as objects (in the following example they're called `p1` and `p2`). Then, you place them next to each other with `+`.

```r
p1 <- ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + labs(title = "Plot 1")
p2 <- ggplot(mpg, aes(x = drv, y = hwy)) + geom_boxplot() + labs(title = "Plot 2")
p1 + p2
```



You can also create complex plot layouts using `|` and `/`. In the following, `|` places the `p1` and `p3` next to each other and `/` moves `p2` to the next line.

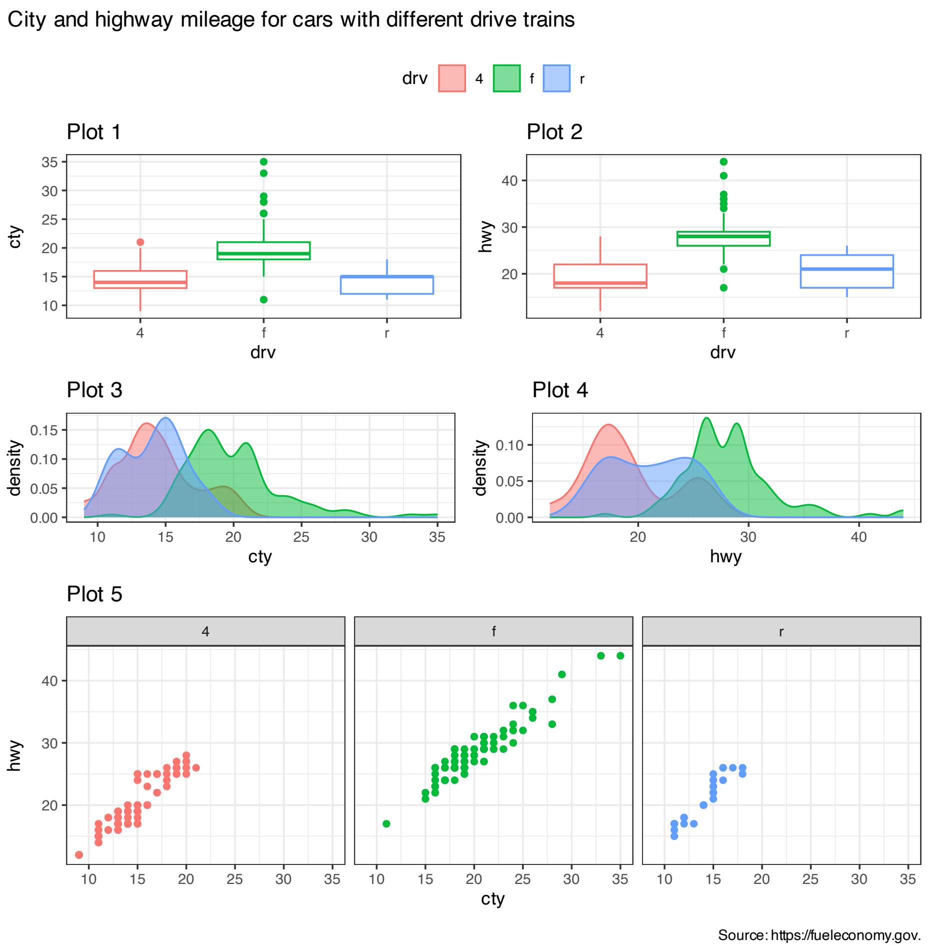
```r
p3 <- ggplot(mpg, aes(x = cty, y = hwy)) + geom_point() + labs(title = "Plot 3")
(p1 | p3)/p2
```



Additionally, patchwork allows you to collect legends from multiple plots into one common legend, customize the placement of the legend as well as dimensions of the plots, and add a common title, subtitle, caption, etc. to your plots. Below we create 5 plots. We have turned off the legends on the box plots and the scatterplot and collected the legends for the density plots at the top of the plot with `& theme(legend.position = "top")`. Note the use of the `&` operator here instead of the usual `+`. This is because we're modifying the theme for the patchwork plot as opposed to the individual ggplots. The legend is placed on top, inside the `guide_area()`. Finally, we have also customized the heights of the various components of our patchwork – the guide has a height of 1, the box plots 3, density plots 2, and the faceted scatterplot 4. Patchwork divides up the area you have allotted for your plot using this scale and places the components accordingly.

```
p1 <- ggplot(mpg, aes(x = drv, y = cty, color = drv)) + geom_boxplot(show.legend = FAL
    labs(title = "Plot 1")

p2 <- ggplot(mpg, aes(x = drv, y = hwy, color = drv)) + geom_boxplot(show.legend = FAL
    labs(title = "Plot 2")

p3 <- ggplot(mpg, aes(x = cty, color = drv, fill = drv)) + geom_density(alpha = 0.5) +
    labs(title = "Plot 3")
```

```
p4 <- ggplot(mpg, aes(x = hwy, color = drv, fill = drv)) + geom_density(alpha = 0.5) +
    labs(title = "Plot 4")

p5 <- ggplot(mpg, aes(x = cty, y = hwy, color = drv)) + geom_point(show.legend = FALSE) +
    facet_wrap(~drv) + labs(title = "Plot 5")

(guide_area()/(p1 + p2)/(p3 + p4)/p5) + plot_annotation(title = "City and highway mileage for car
    caption = "Source: https://fueleconomy.gov.") + plot_layout(guides = "collect",
    heights = c(1, 3, 2, 4)) & theme(legend.position = "top")
```

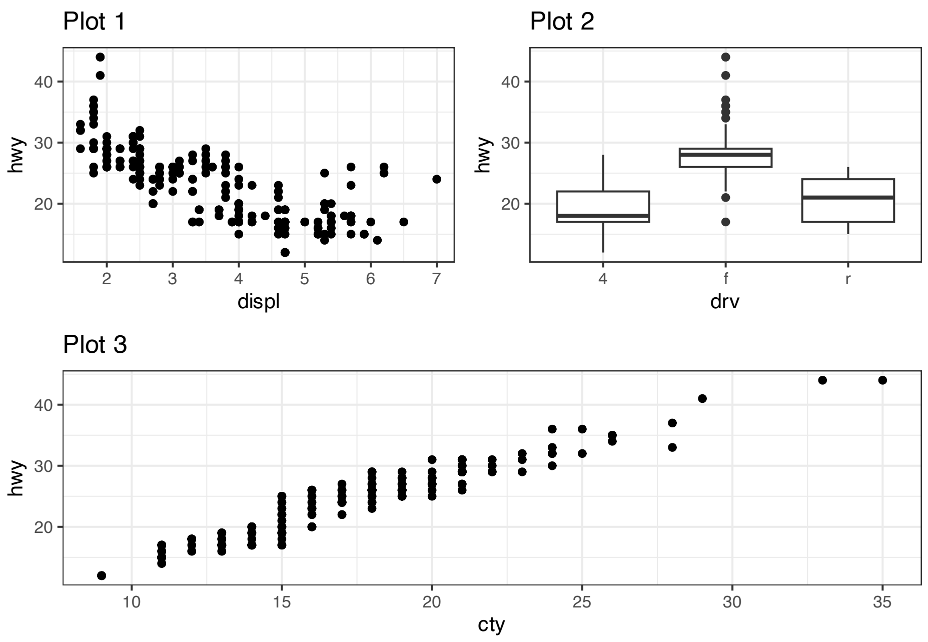


If you'd like to learn more about combining and layout out multiple plots with patchwork, we recommend looking through the guides on the package website: https://patchwork.data-imaginist.com.

### Exercises 1.6.1

1. What happens if you omit the parentheses in the following plot layout. Try it. Can you explain why this happens?

```
p1 <- ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + labs(title = "Plot 1")
p2 <- ggplot(mpg, aes(x = drv, y = hwy)) + geom_boxplot() + labs(title = "Plot 2")
p3 <- ggplot(mpg, aes(x = cty, y = hwy)) + geom_point() + labs(title = "Plot 3")

(p1 | p2)/p3
```



**Ans-1.6.1.1:**

2. Using the three plots from the previous exercise, recreate the following patchwork.

**Ans-1.6.1.2:**

## 1.7 Summary

In this chapter you've learned about adding plot labels such as title, subtitle, caption as well as modifying default axis labels, using annotation to add informational text to your plot or to highlight specific data points, customizing the axis scales, and changing the theme of your plot. You've also learned about combining multiple plots in a single graph using both simple and complex plot layouts.

Fig. A:

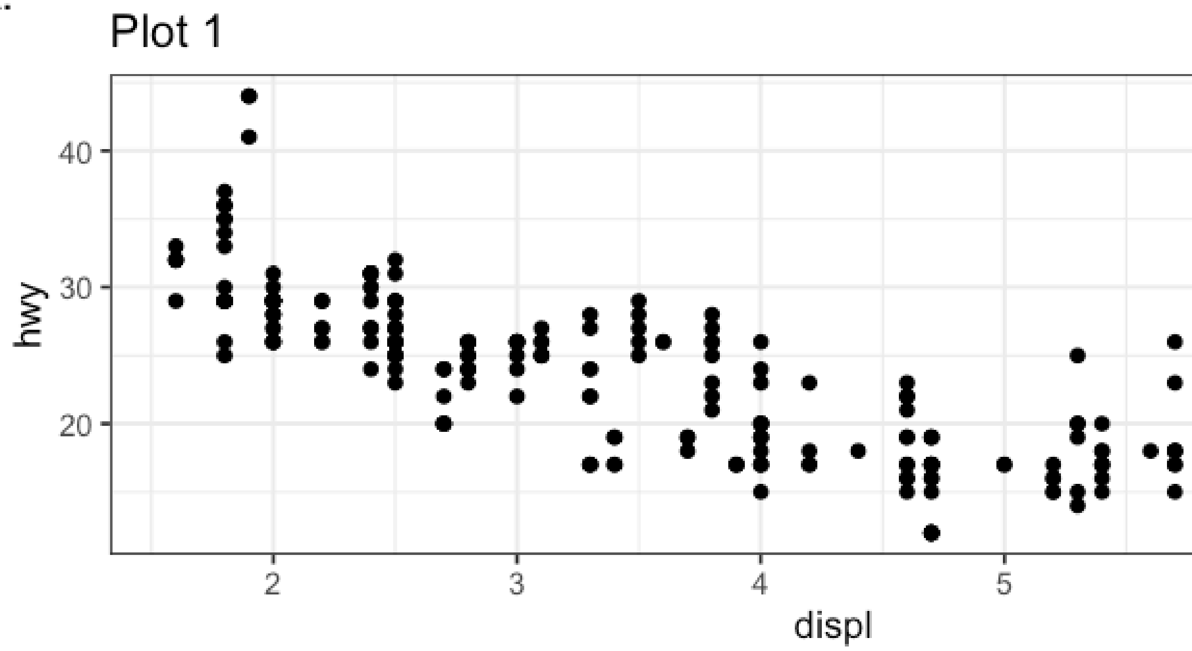


Fig. B:

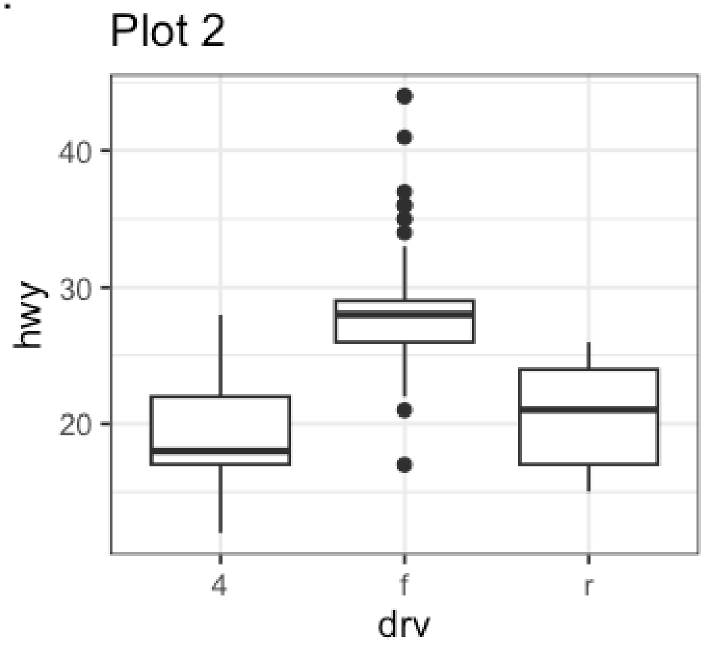


Fig. C:

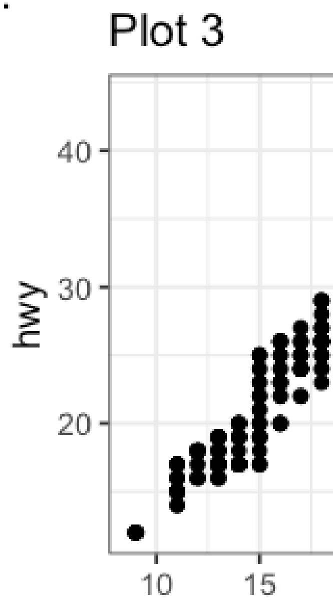


Figure 1.4: Scatterplot of highway versus city fuel efficiency.

While you've so far learned about how to make many different types of plots and how to customize them using a variety of techniques, we've barely scratched the surface of what you can create with ggplot2. If you want to get a comprehensive understanding of ggplot2, we recommend reading the book, *ggplot2: Elegant Graphics for Data Analysis*. Other useful resources are the *R Graphics Cookbook* by Winston Chang and *Fundamentals of Data Visualization* by Claus Wilke.

# Chapter 2

# Layers

```r
# set global options
source("_common.R")
```

## 2.1 Introduction

In this chapter, you'll expand on your ggplot2 foundation as you learn about the layered grammar of graphics. We'll start with a deeper dive into aesthetic mappings, geometric objects, and facets. Then, you will learn about statistical transformations ggplot2 makes under the hood when creating a plot. These transformations are used to calculate new values to plot, such as the heights of bars in a bar plot or medians in a box plot. You will also learn about position adjustments, which modify how geoms are displayed in your plots.

We will not cover every single function and option for each of these layers, but we will walk you through the most important and commonly used functionality provided by ggplot2 as well as introduce you to packages that extend ggplot2.

### 2.1.1 Prerequisites

This chapter focuses on ggplot2. To access the datasets, help pages, and functions used in this chapter, first install the packages `patchwork` and `ggridges` if you need to by uncommenting this code:

```r
# install.packages(c('patchwork','ggridges'))
```

Then load the `tidyverse` and the `patchwork` packages by into your session:

```r
library(tidyverse)
library(patchwork)
```

## 2.2 Aesthetic mappings

The `mpg` data frame is bundled with the ggplot2 package and contains 234 observations on 38 car models.

```
`?`(mpg)

nrow(mpg)
## [1] 234

sapply(mpg, class)
## manufacturer        model         displ         year
##  "character"  "character"    "numeric"    "integer"
##          cyl        trans           drv          cty
##    "integer"  "character"  "character"    "integer"
##          hwy           fl         class
##    "integer"  "character"  "character"
```

Among the variables in `mpg` are:

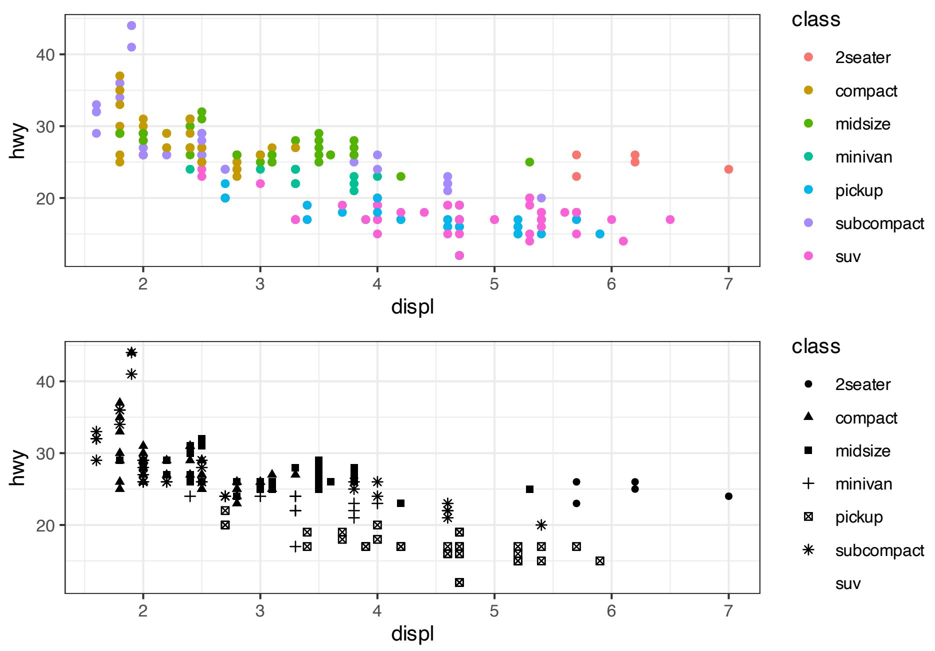
1. `displ`: A car's engine size, in liters. A numerical variable.

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. A numerical variable.

3. `class`: Type of car. A categorical variable.

Let's start by visualizing the relationship between `displ` and `hwy` for various `classes` of cars. We can do this with a scatterplot where the numerical variables are mapped to the `x` and `y` aesthetics and the categorical variable is mapped to an aesthetic like `color` or `shape`.

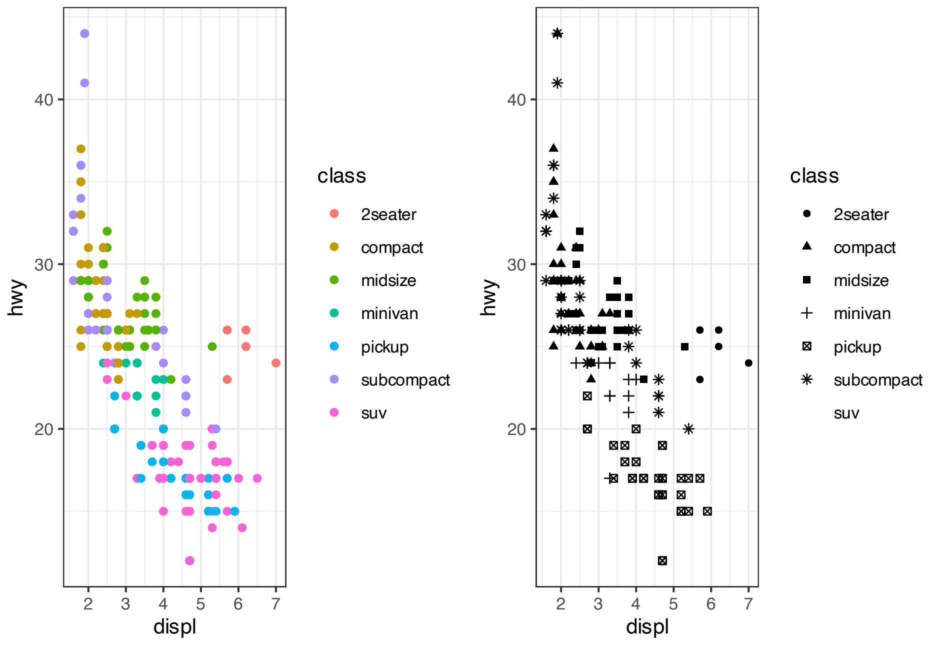
```
ggplot(mpg, aes(x = displ, y = hwy, color = class)) + geom_point() +
    ggplot(mpg, aes(x = displ, y = hwy, shape = class)) + geom_point() +
    plot_layout(nrow = 2)
```



Note the use of `plot_layout` from the `patchwork` package to layout the plots in two rows. You can also use mathematical symbols to layout ggplots:

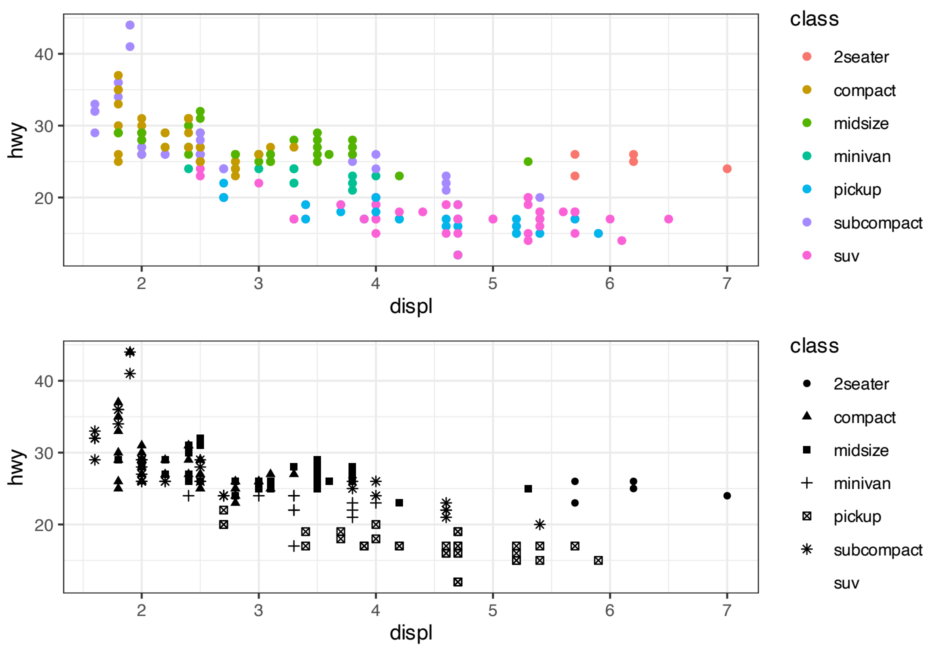
Use `+` to plot them side-by-side:

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



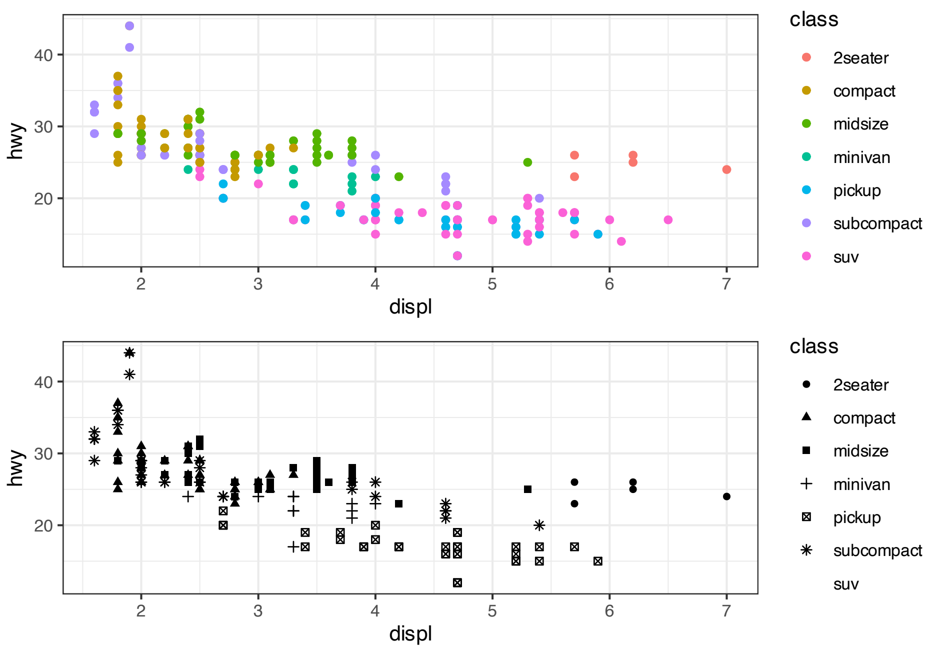
Use '/' to plot them in one column:

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



**Notice that parentheses are required for this operation unless you first save the plots:**

```r
p1 <- ggplot(mpg, aes(x = displ, y = hwy, color = class)) + geom_point()

p2 <- ggplot(mpg, aes(x = displ, y = hwy, shape = class)) + geom_point()

p1/p2
```



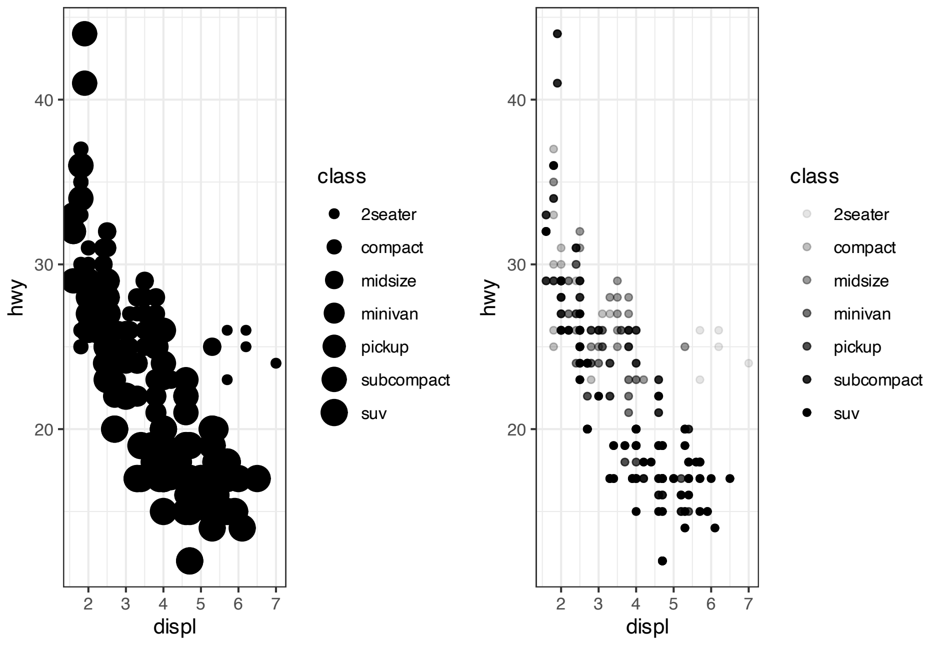
When `class` is mapped to `shape`, we get two warnings:

> 1: The shape palette can deal with a maximum of 6 discrete values because more than 6 becomes difficult to discriminate; you have 7. Consider specifying shapes manually if you must have them.
>
> 2: Removed 62 rows containing missing values (`geom_point()`).

Since ggplot2 will only use six shapes at a time, by default, additional groups will go unplotted when you use the shape aesthetic. The second warning is related – there are 62 SUVs in the dataset and they're not plotted.

Similarly, we can map `class` to `size` or `alpha` aesthetics as well, which control the shape and the transparency of the points, respectively.

```r
ggplot(mpg, aes(x = displ, y = hwy, size = class)) + geom_point() +
    ggplot(mpg, aes(x = displ, y = hwy, alpha = class)) + geom_point() +
    plot_layout(ncol = 2)
```



Both of these produce warnings as well:

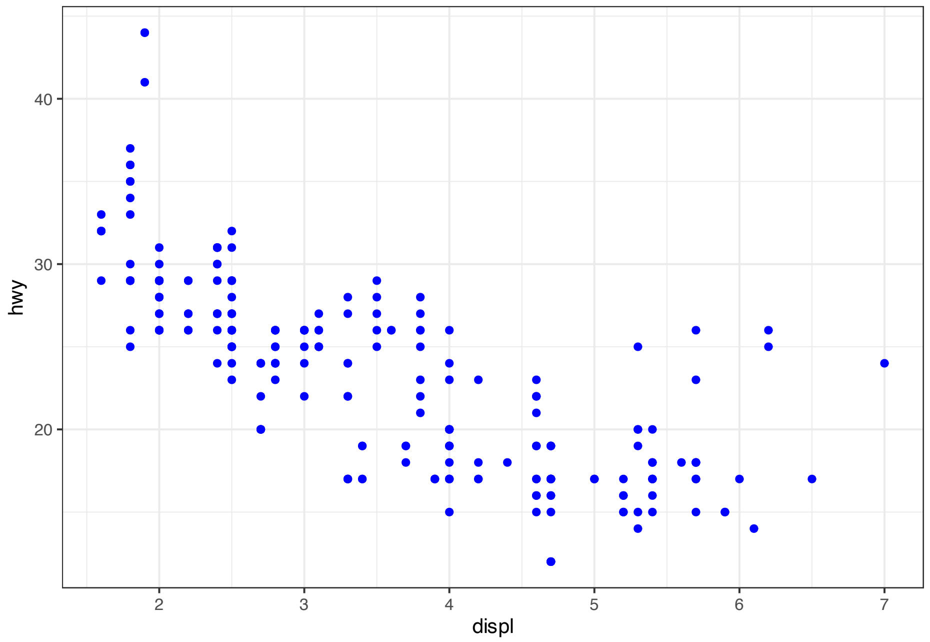
> Using alpha for a discrete variable is not advised.

Mapping an unordered discrete (categorical) variable (`class`) to an ordered aesthetic (`size` or `alpha`) is generally not a good idea because it implies a ranking that does not in fact exist.

Once you map 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 provides the same information as a legend; it explains the mapping between locations and values.

You can also set the visual properties of your geom manually as an argument of your geom function (*outside* of `aes()`) instead of relying on a variable mapping to determine the appearance. For example, we can make all of the points in our plot blue:

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



Here, the color doesn't convey information about a variable, but only changes the appearance of the plot. You'll need to pick a value that makes sense for that aesthetic:

- The name of a color as a character string, e.g., `color = "blue"`
- The size of a point in mm, e.g., `size = 1`
- The shape of a point as a number, e.g, `shape = 1`, as shown in 2.1.

So far we have discussed aesthetics that we can map or set in a scatterplot, when using a point geom. You can learn more about all possible aesthetic mappings in the aesthetic specifications vignette at https://ggplot2.tidyverse.org/articles/ggplot2-specs.html.

The specific aesthetics you can use for a plot depend on the geom you use to represent the data. In the next section we dive deeper into geoms.

## 2.2.1 Exercises

1. Create a scatterplot of `hwy` vs. `displ` where the points are pink filled in triangles.

**Ans-2.2.1.1:**

2. Why did the following code not result in a plot with blue points?

```
# | fig-alt: | | Scatterplot of highway fuel efficiency
# versus engine size of cars | that shows a negative
# association. All points are red and | the legend shows a
```

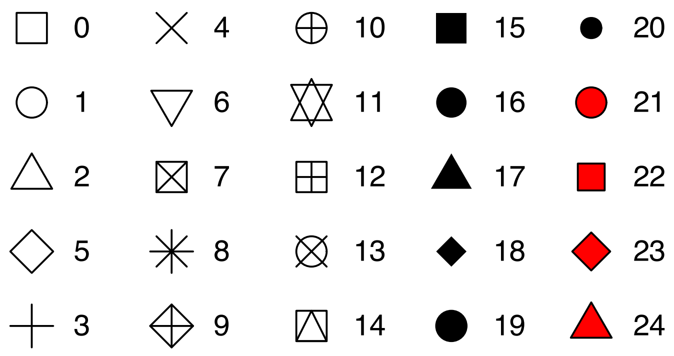


Figure 2.1: R has 25 built-in shapes that are identified by numbers. There are some seeming duplicates: for example, 0, 15, and 22 are all squares. The difference comes from the interaction of the ‘color‘ and ‘fill‘ aesthetics. The hollow shapes (0–14) have a border determined by ‘color‘; the solid shapes (15–20) are filled with ‘color‘; the filled shapes (21–24) have a border of ‘color‘ and are filled with ‘fill‘. Shapes are arranged to keep similar shapes next to each other.

```
# red point that is mapped to the word blue.

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

**Ans-2.2.1.2:**

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

**Ans-2.2.1.3:**

4. What happens if you map an aesthetic to something other than a variable name, like `aes(color = displ < 5)`? Try it by adding color to the following plot.

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

**Ans-2.2.1.4:**

## 2.3 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 geometric object, geom, to represent the 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 the following code:

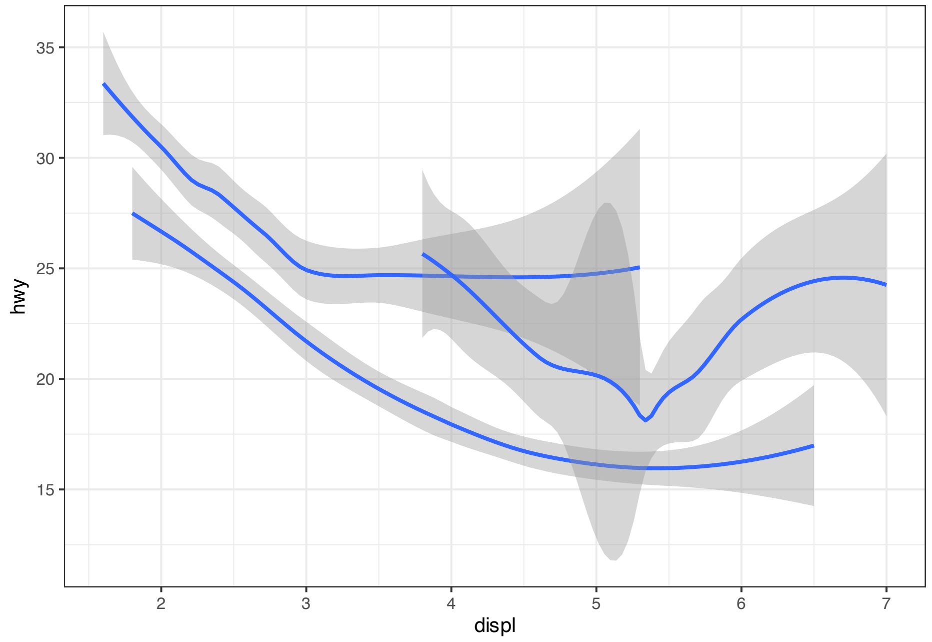
```
# Top
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point()

# Bottom
ggplot(mpg, aes(x = displ, y = hwy)) + geom_smooth()
```

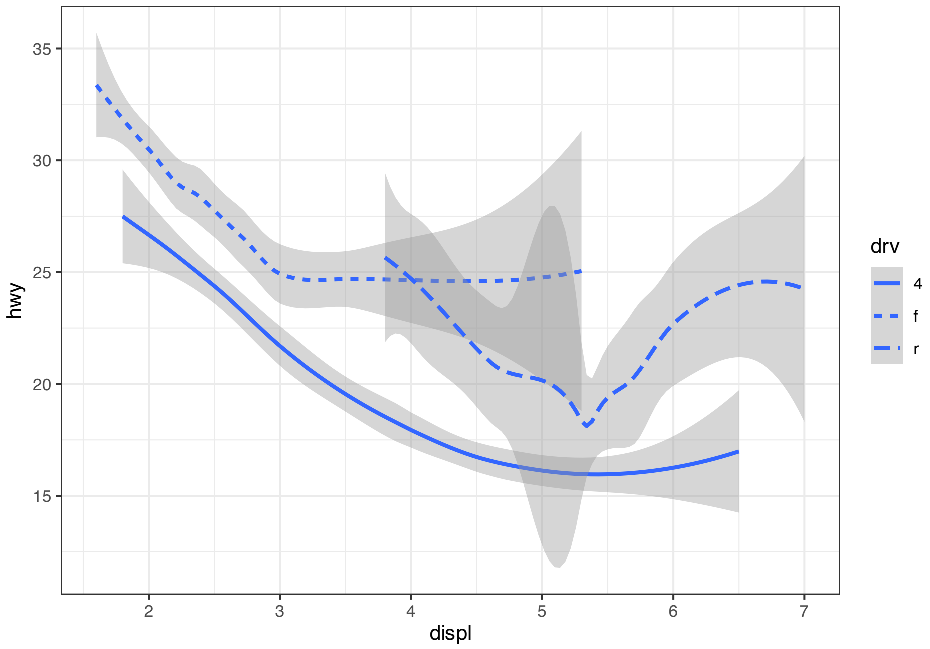
Every geom function in ggplot2 takes a `mapping` argument, either defined locally in the geom layer or globally in the `ggplot()` layer.

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. If you try, ggplot2 will silently ignore that aesthetic mapping. 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.

```
# Top
ggplot(mpg, aes(x = displ, y = hwy, shape = drv)) + geom_smooth()

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

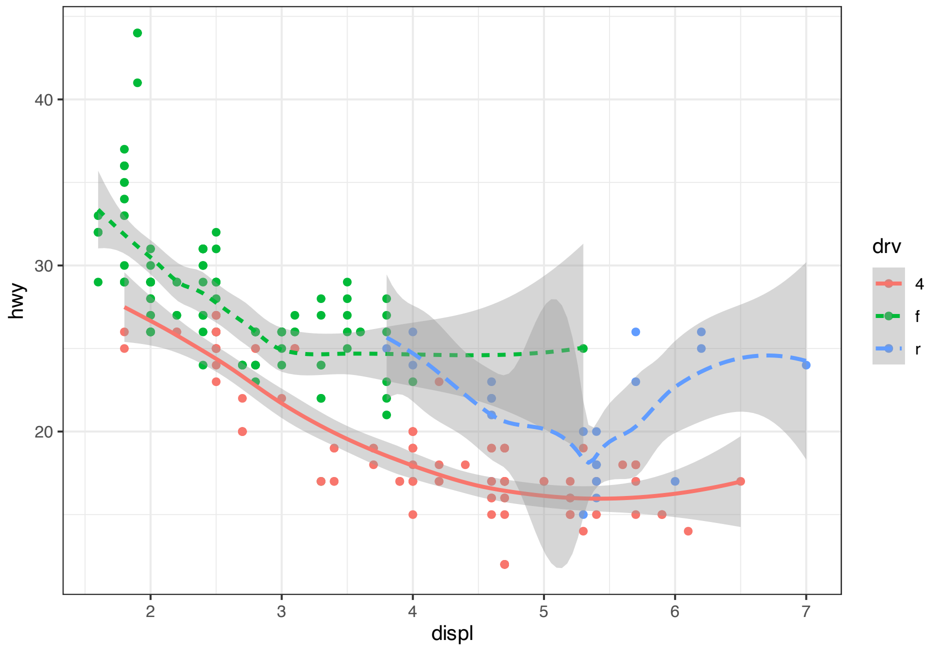




Here, `geom_smooth()` separates the cars into three lines based on their `drv` value, which describes a car's drive train. One line describes all of the points that have a `4` value, one line describes all of the points that have an `f` value, and one line describes all of the points that have an `r` 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 clearer by overlaying the lines on top of the raw data and then coloring everything according to `drv`.

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



Notice that this plot contains two geoms in the same graph.

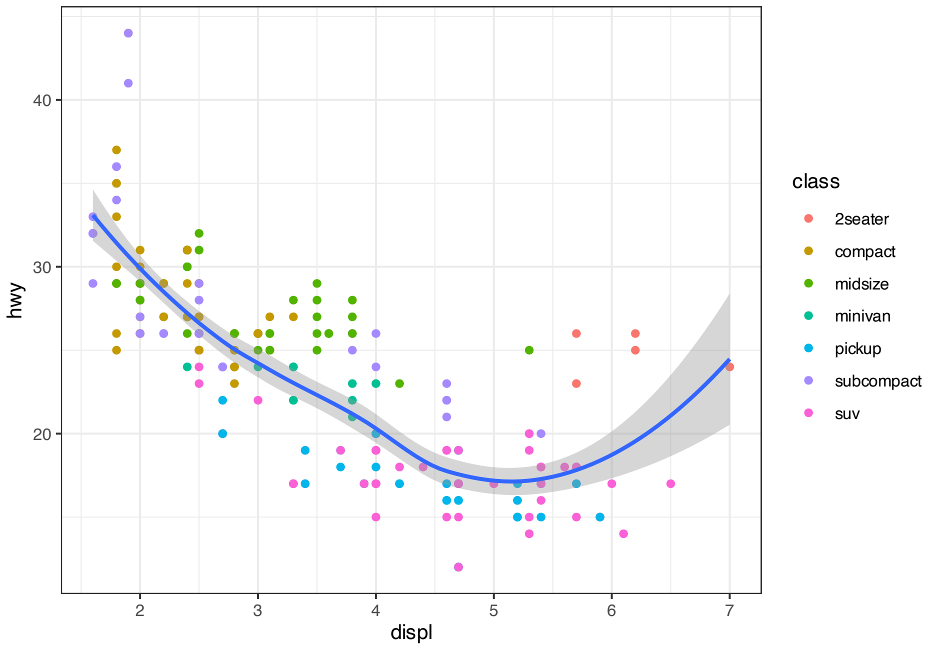
Many geoms, like `geom_smooth()`, use a single geometric object to display multiple rows of data. For these geoms, you can set the `group` aesthetic to a categorical variable to draw multiple objects. ggplot2 will draw a separate object for each unique value of the grouping variable.

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.

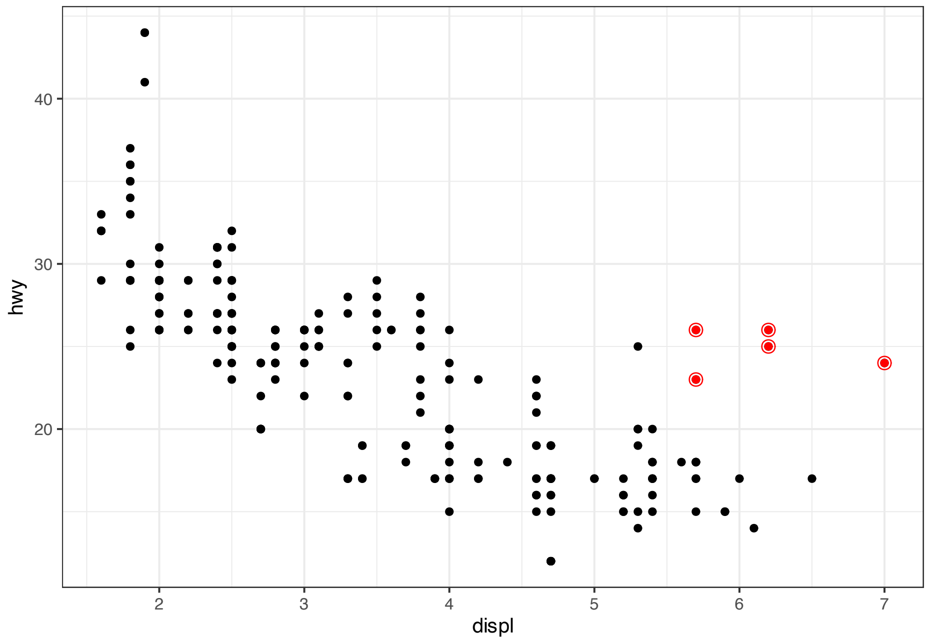
If you place mappings in a geom function, ggplot2 will treat them as local mappings for the layer. It will use these mappings to extend or overwrite the global mappings *for that layer only*. his makes it possible to display different aesthetics in different layers.

```
# | fig-alt: | | Scatterplot of highway fuel efficiency
# versus engine size of cars, where | points are colored
# according to the car class. A smooth curve following |
# the trajectory of the relationship between highway fuel
# efficiency versus | engine size of cars is overlaid along
# with a confidence interval around it.

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



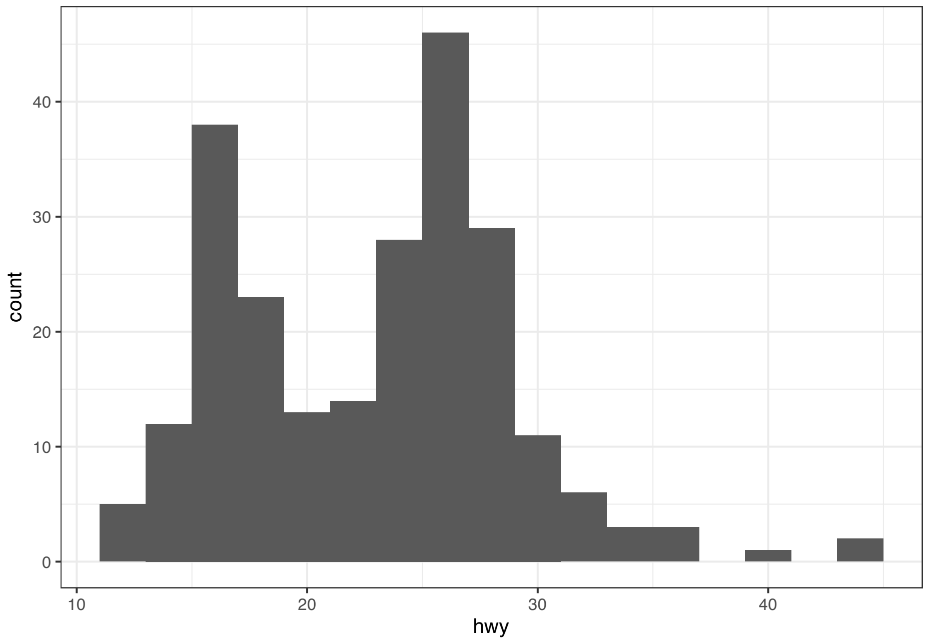
You can use the same idea to specify different `data` for each layer. Here, we use red points as well as open circles to highlight two-seater cars. The local data argument in `geom_point()` overrides the global data argument in `ggplot()` for that layer only.

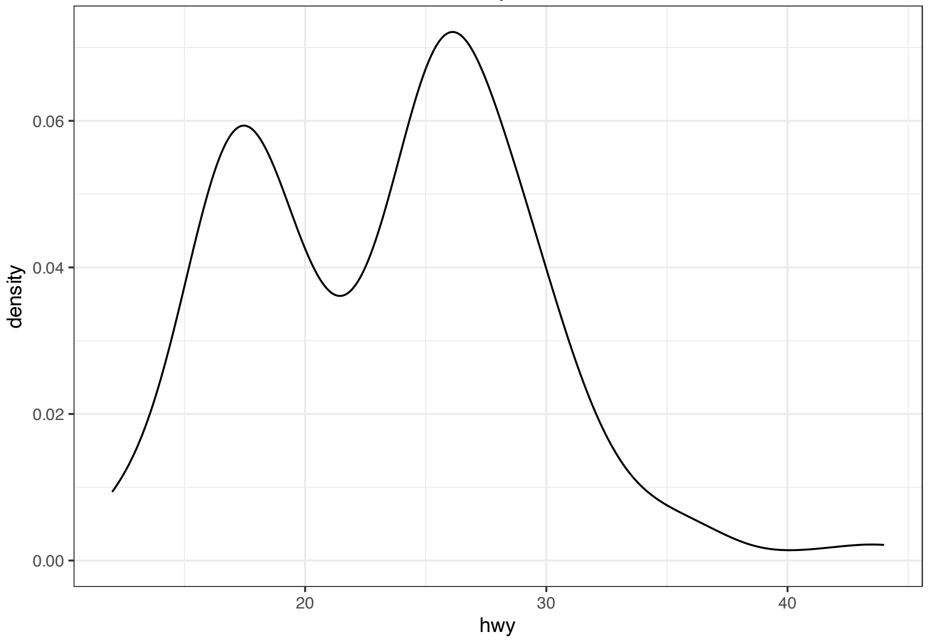
```r
g <- ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + geom_point(data = mpg %>%
    filter(class == "2seater"), color = "red")

g + geom_point(data = mpg %>%
    filter(class == "2seater"), shape = "circle open", size = 3,
    color = "red")
```

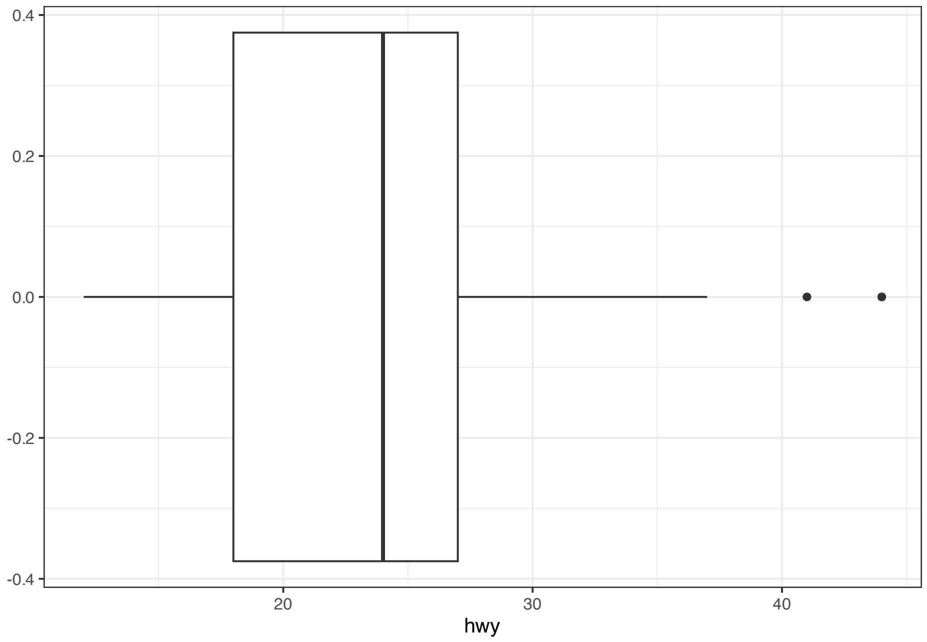


Geoms are the fundamental building blocks of ggplot2. You can completely transform the look of your plot by changing its geom, and different geoms can reveal different features of your data. For example, the histogram and density plot below reveal that the distribution of highway mileage is bimodal and right skewed while the boxplot reveals two potential outliers.

```
# | fig-alt: | | Three plots: histogram, density plot, and
# box plot of highway | mileage.

# Top
ggplot(mpg, aes(x = hwy)) + geom_histogram(binwidth = 2)

# Middle
ggplot(mpg, aes(x = hwy)) + geom_density()

# Bottom
ggplot(mpg, aes(x = hwy)) + geom_boxplot()
```


count
0
10
20
30
40
10
20
30
40
hwy


density
0.00
0.02
0.04
0.06
20
30
40
hwy

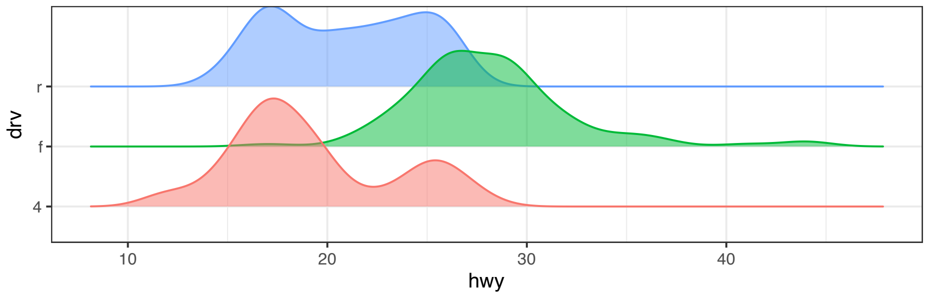


ggplot2 provides more than 40 geoms but these don't cover all possible plots one could make. If you need a different geom, we recommend looking into extension packages first to see if someone else has already implemented it (see https://exts.ggplot2.tidyverse.org/gallery/ for a sampling).

For example, the **ggridges** package (https://wilkelab.org/ggridges) is useful for making ridgeline plots, which can be useful for visualizing the density of a numerical variable for different levels of a categorical variable.

In the following plot not only did we use a new geom (`geom_density_ridges()`), but we have also mapped the same variable to multiple aesthetics (`drv` to `y`, `fill`, and `color`) as well as set an aesthetic (`alpha = 0.5`) to make the density curves transparent.

```r
library(ggridges)

ggplot(mpg, aes(x = hwy, y = drv, fill = drv, color = drv)) +
  geom_density_ridges(alpha = 0.5, show.legend = FALSE)
```



The best place to get a comprehensive overview of all of the geoms ggplot2 offers, as well as all functions in the package, is the reference page: https://ggplot2.tidyverse.org/reference. To learn more about any single geom, use the help (e.g., `?geom_smooth`).

### 2.3.1 Exercises

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

**Ans-2.3.1.1a:**

b. Use the following ggplot to create 4 plots, `geom_line()`, `geom_boxplot()`, `geom_point()` and `geom_area()`. Include a title specifying the geometry for each.

```
df <- data.frame(x = c(3, 1, 5), y = c(2, 4, 6), label = c("a",
    "b", "c"))

p <- ggplot(df, aes(x, y, label = label))
```

**Ans-2.3.1.1b:**

2. Earlier in this chapter we used `show.legend` without explaining it:

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

What does `show.legend = FALSE` do here? What happens if you remove it? Try it.

**Ans-2.3.1.2:**

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

**Ans-2.3.1.3:**

4. Recreate the R code necessary to generate and save the following graph 2.2 and arrange them in a figure with 2 columns. Note that wherever a

categorical variable is used in the plot, it's `drv`. Use the mathematical notation in one, and the function from `patchwork`, `plot_layout` in another for laying out the plot.

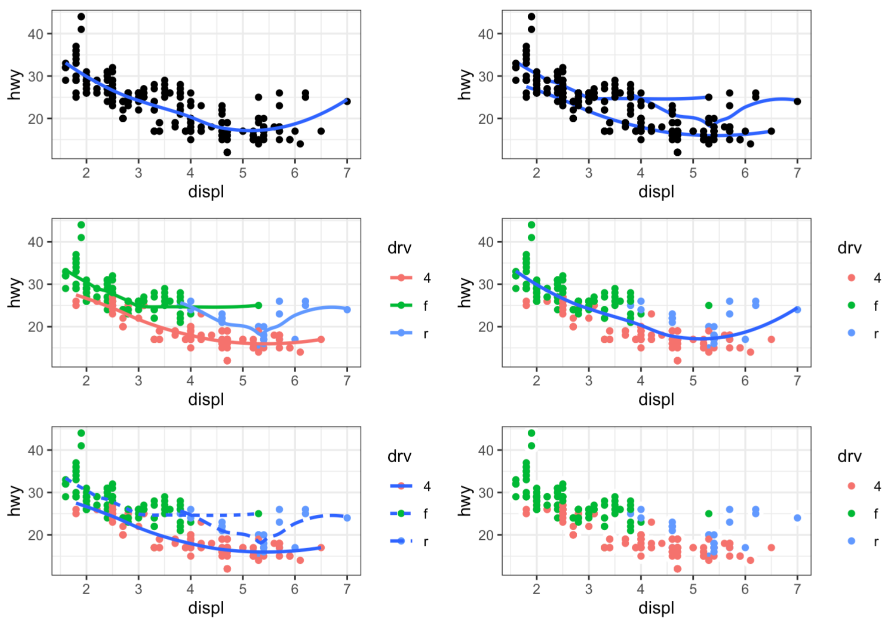


Figure 2.2: Six plots arranged in two columns

**Ans-2.3.1.4:**

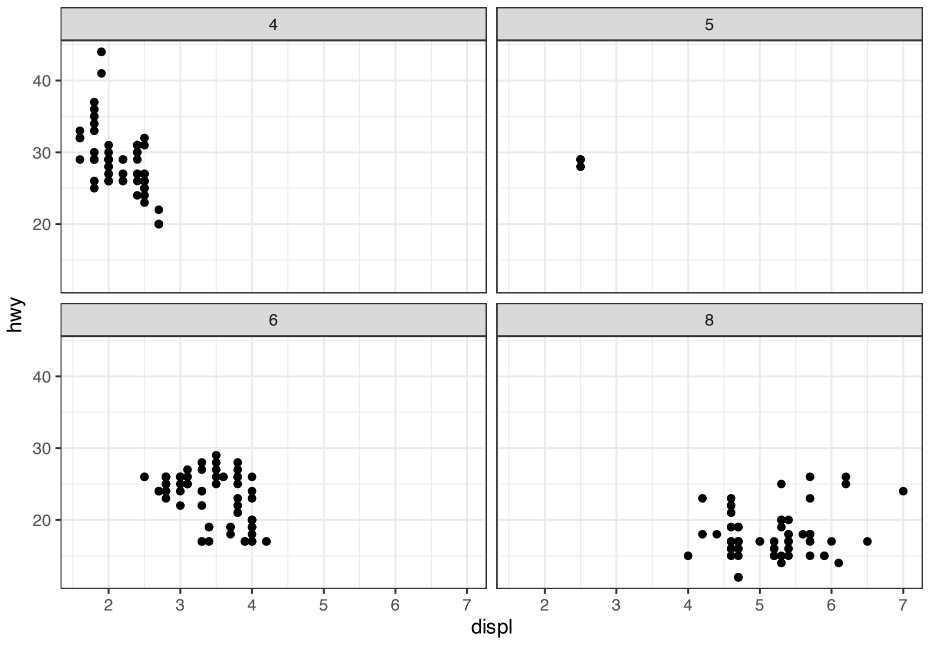
## 2.4 Facets

Another way, which is 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()` is a formula, which you create with `~` followed by a variable name. The variable that you pass to `facet_wrap()` should be categorical.

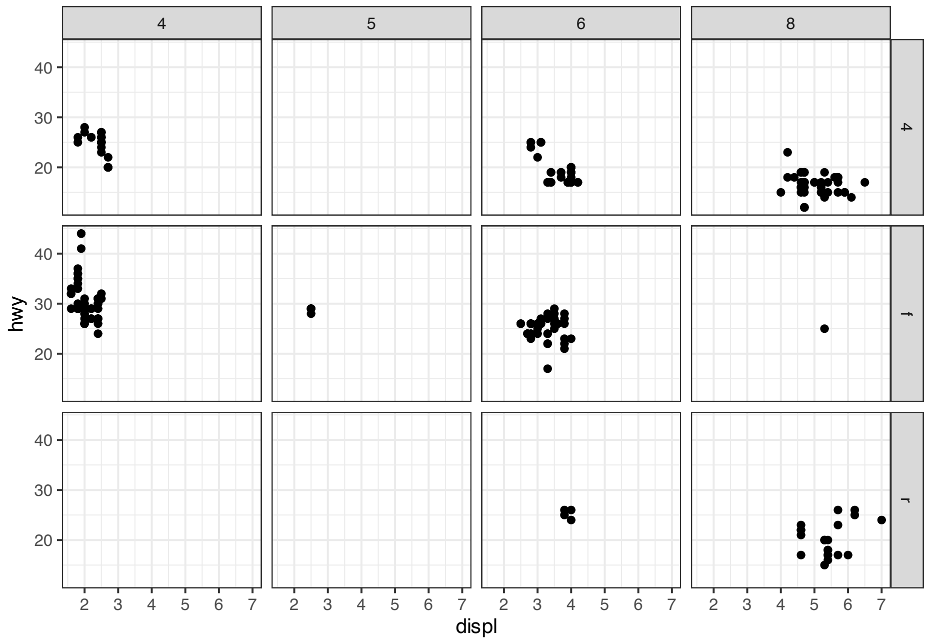
Here "formula" is the name of the thing created by `~`, not a synonym for "equation".

```r
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + facet_wrap(~cyl)
```



To facet your plot with the combination of two variables, switch from `facet_wrap()` to `facet_grid()`. The first argument of `facet_grid()` is also a formula, but now it's a double sided formula: `rows ~ cols`.

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

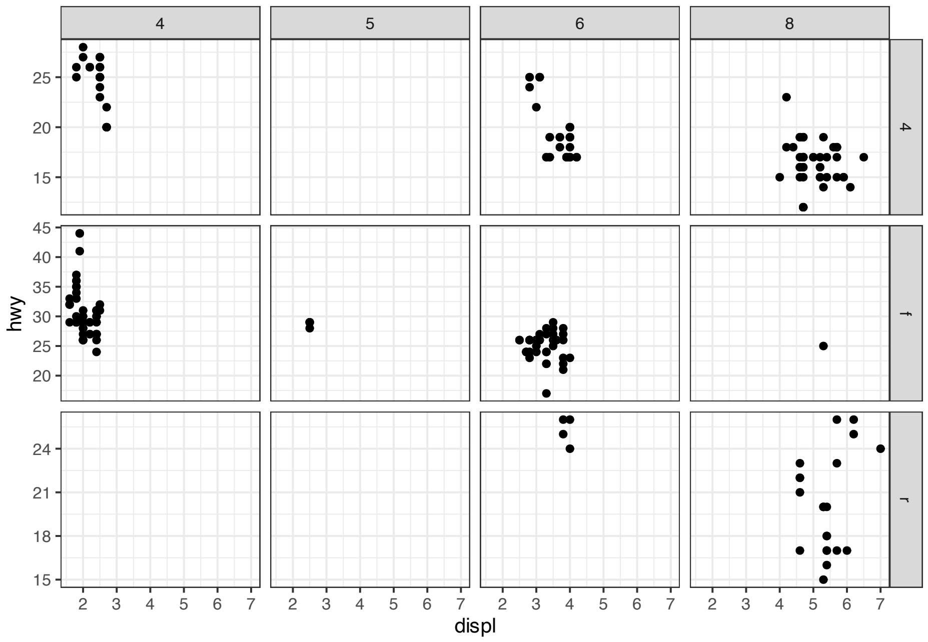


By default each of the facets share the same scale and range for x and y axes.

This is useful when you want to compare data across facets but it can be limiting when you want to visualize the relationship within each facet better.

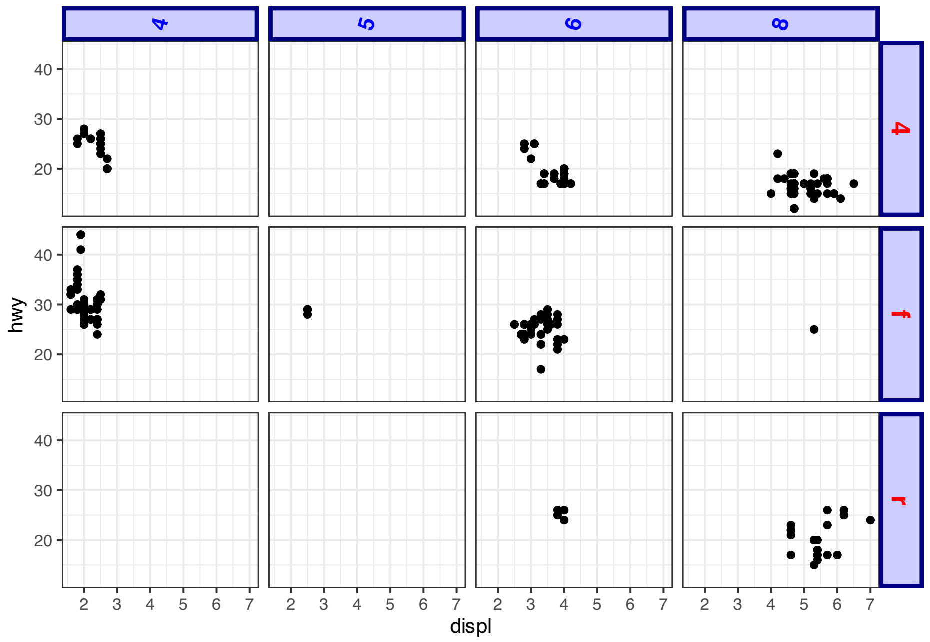
Setting the `scales` argument in a faceting function to `"free"` will allow for different axis scales across both rows and columns, `"free_x"` will allow for different scales across rows, and `"free_y"` will allow for different scales across columns.

```
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + facet_grid(drv ~
    cyl, scales = "free_y")
```



Modifying facet label appearance:

```
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + facet_grid(drv ~
    cyl) + theme(strip.text.x = element_text(size = 12, face = "bold",
    colour = "blue", angle = 75), strip.text.y = element_text(size = 14,
    face = "bold", colour = "red"), strip.background = element_rect(colour = "navy",
    fill = "#CCCCFF", linewidth = 2))
```



### 2.4.1 Exercises

1. What happens if you facet on a continuous variable? Try it on the. plot below and facet on `hwy`.

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

**Ans-2.4.1.1:**

2. Run the following code and facet on drv vs cyl. What do the empty cells mean?

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

**Ans-2.4.1.2:**

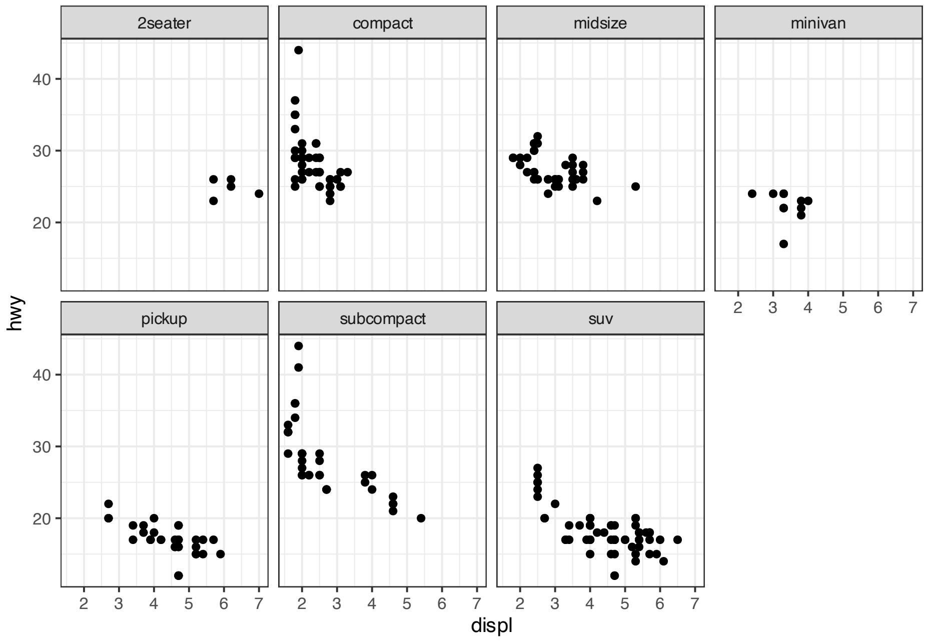
3. What plots does the following code make? What does `.` do?

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

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

**Ans-2.4.1.3:**

4. Take the first faceted plot in this section:

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



a. What are the advantages to using faceting instead of the color aesthetic? What are the disadvantages?

**Ans-2.4.1.4a:**

b. What if you were interested in a specific class, e.g. compact cars? How would highlighting that group using only an additional layer added to the plot below?

```r
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(color = "gray")
```

**Ans-2.4.1.4b:**

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` arguments?

**Ans-2.4.1.5:**

6. Which of the following plots makes it easier to compare engine size (`displ`) across cars with different drive trains? What does this say about when to place a faceting variable across rows or columns?

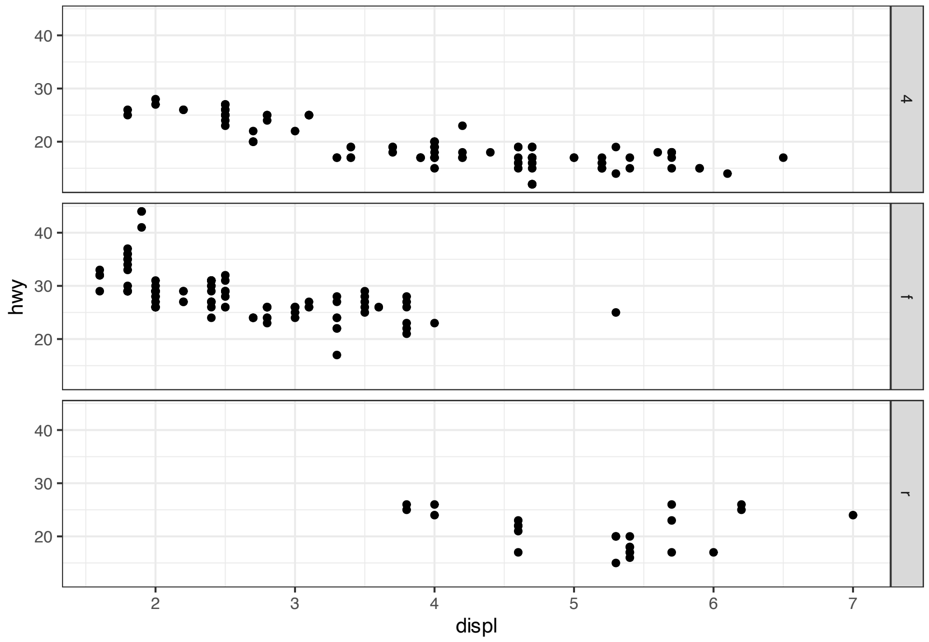
```r
ggplot(mpg, aes(x = displ)) + geom_histogram() + facet_grid(drv ~
    .)
```

```
ggplot(mpg, aes(x = displ)) + geom_histogram() + facet_grid(. ~
    drv)
```

**Ans-2.4.1.6:**

7. Recreate the following plot using `facet_wrap()` instead of `facet_grid()`. How do the positions of the facet labels change?

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



**Ans-2.4.1.7:**

8. Modify the facet label in the following plot so that it has a background color of `navy` with a `red` outline and `white` text.

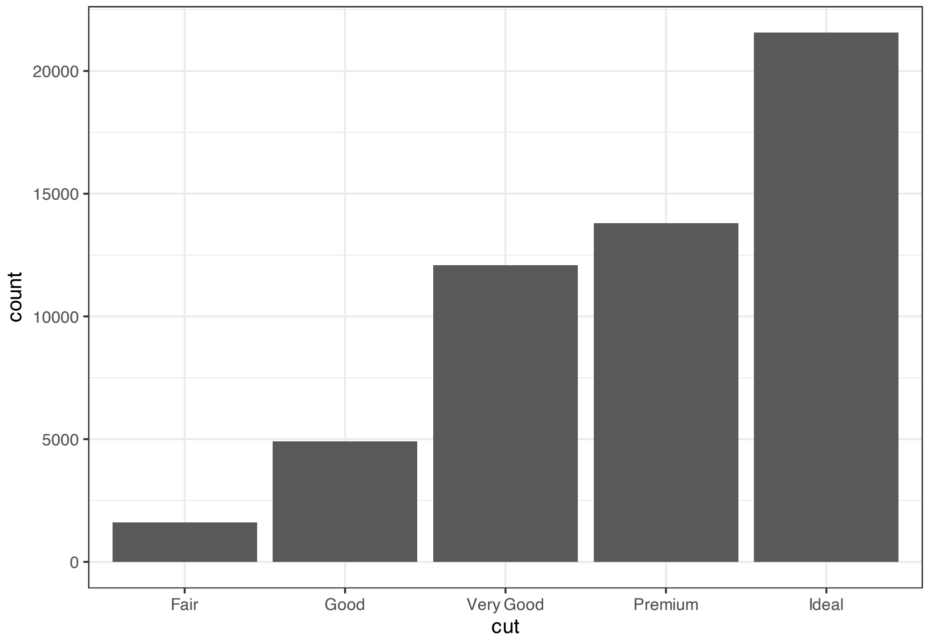
```
ggplot(mpg) + geom_point(aes(x = displ, y = hwy)) + facet_grid(drv ~
    .)
```

**Ans-2.4.1.8:**

## 2.5 Statistical transformations

Consider a basic bar chart, drawn with `geom_bar()` or `geom_col()`. The following chart displays the total number of diamonds in the `diamonds` dataset, grouped by `cut`. The `diamonds` dataset is in the ggplot2 package and contains information on ~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(diamonds, aes(x = cut)) + geom_bar()
```

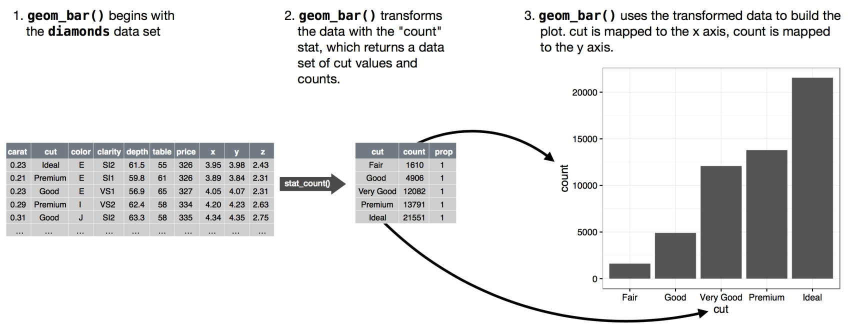


On the x-axis, the chart displays `cut`, a variable from `diamonds`. On the y-axis, it displays count, but count is not a variable in `diamonds`! Where does count come from? Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

- 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 compute the five-number summary of the distribution and then display that summary as a specially formatted box.

The algorithm used to calculate new values for a graph is called a **stat**, short for statistical transformation. 2.3 shows how this process works with `geom_bar()`.

You can learn which stat a geom uses by inspecting the default value for the `stat` argument. For example, `?geom_bar` shows that the default value for `stat` is "count", which means that `geom_bar()` uses `stat_count()`. `stat_count()` is documented on the same page as `geom_bar()`. If you scroll down, the section called "Computed variables" explains that it computes two new variables: `count`


1. geom_bar() begins with the diamonds data set
2. geom_bar() transforms the data with the "count" stat, which returns a data set of cut values and counts.
3. geom_bar() uses the transformed data to build the plot. cut is mapped to the x axis, count is mapped to the y axis.
stat_count()
count
cut
0
5000
10000
15000
20000
Fair
Good
Very Good
Premium
Ideal

| 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 |
| … | … | … | … | … | … | … | … | … | … |

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

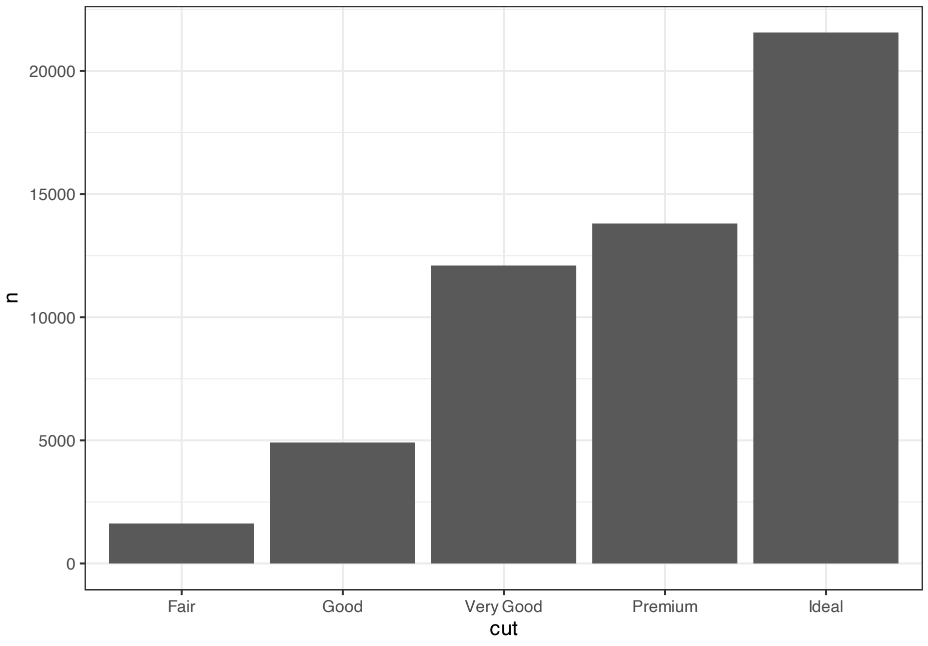
Figure 2.3: When creating a bar chart we first start with the raw data, then aggregate it to count the number of observations in each bar, and finally map those computed variables to plot aesthetics.

and `prop`.

Every geom has a default stat; and every stat has a default geom. This means that you can typically use geoms without worrying about the underlying statistical transformation. However, there are three reasons why you might need to use a stat explicitly:

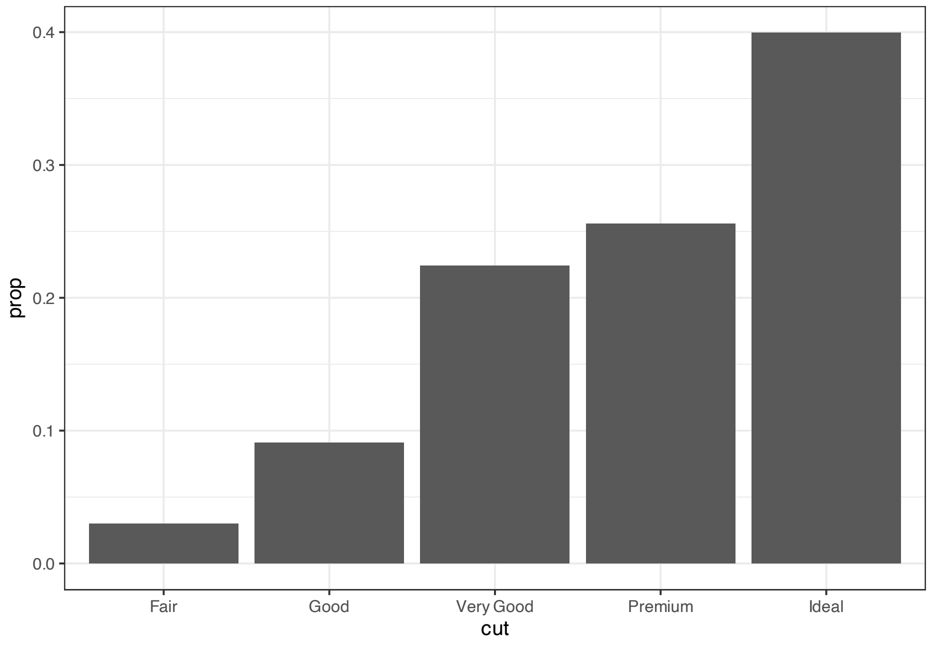
1. You might want to override the default stat. In the code below, we change the stat of `geom_bar()` from count (the default) to identity.This lets us map the height of the bars to the raw values of a y variable.

```r
# | | fig-alt: | | Bar chart of number of each cut of
# diamond. There are roughly 1500 Fair, 5000 Good, 12000
# Very Good, 14000 Premium, and 22000 Ideal cut diamonds.

diamonds %>%
    count(cut) %>%
    ggplot(aes(x = cut, y = n)) + geom_bar(stat = "identity")
```



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 proportions, rather than counts:

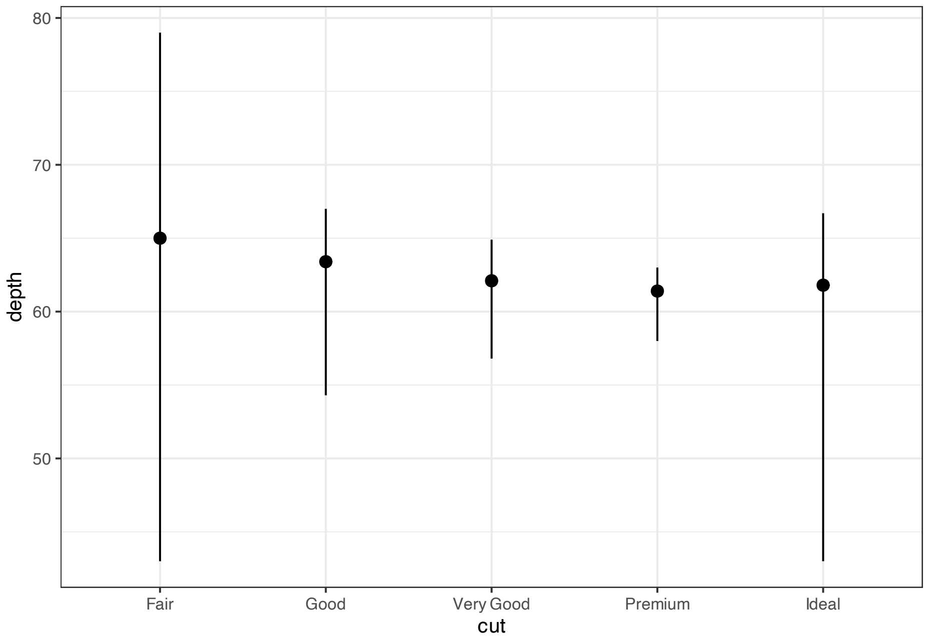
```
# | | fig-alt: | | Bar chart of proportion of each cut of
# diamond. Roughly, Fair | diamonds make up 0.03, Good
# 0.09, Very Good 0.22, Premium 0.26, and | Ideal 0.40.

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



```
To find the possible variables that can be computed by the stat, look for the section titled "com
```

3. You might want to draw greater attention to the statistical transformation in your code. For example, you might use `stat_summary()`, which summarizes the y values for each unique x value, to draw attention to the summary that you're computing:

```
# | fig-alt: | | A plot with depth on the y-axis and cut on
# the x-axis (with levels | fair, good, very good, premium,
# and ideal) of diamonds. For each level | of cut, vertical
# lines extend from minimum to maximum depth for diamonds |
# in that cut category, and the median depth is indicated
# on the line | with a point.


ggplot(diamonds) + stat_summary(aes(x = cut, y = depth), fun.min = min,
    fun.max = max, fun = median)
```



ggplot2 provides more than 20 stats for you to use. Each stat is a function, so you can get help in the usual way, e.g., `?stat_bin`.

### 2.5.1 Exercises

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

**Ans-2.5.1.1:**

2.a. What does `geom_col()` do? How is it different from `geom_bar()`?

**Ans-2.5.1.2a:**

b. Plot mpg and count the classes in drv. Compare `geom_bar` to `geom_col`.

```r
g <- ggplot(mpg, aes(class))
```

**Ans-2.5.1.2b:**

3. Most geoms and stats come in pairs that are almost always used in concert. Make a list of all the pairs. What do they have in common? (Hint: Read through the documentation.). Write the stat used for `geom_bar()`,`geom_smooth()` and `geom_boxplot()`.

**Ans-2.5.1.3:**

4. What variables does `stat_smooth()` compute? What arguments control its behavior?

**Ans-2.5.1.4:**

5.a. In our proportion bar chart, we needed to set `group = 1`. Why? In other words, what is the problem with these two graphs? Replot the two plots below using `group = 1`.

```r
ggplot(diamonds, aes(x = cut, y = after_stat(prop))) + geom_bar()

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

**Ans-2.5.1.5a:**

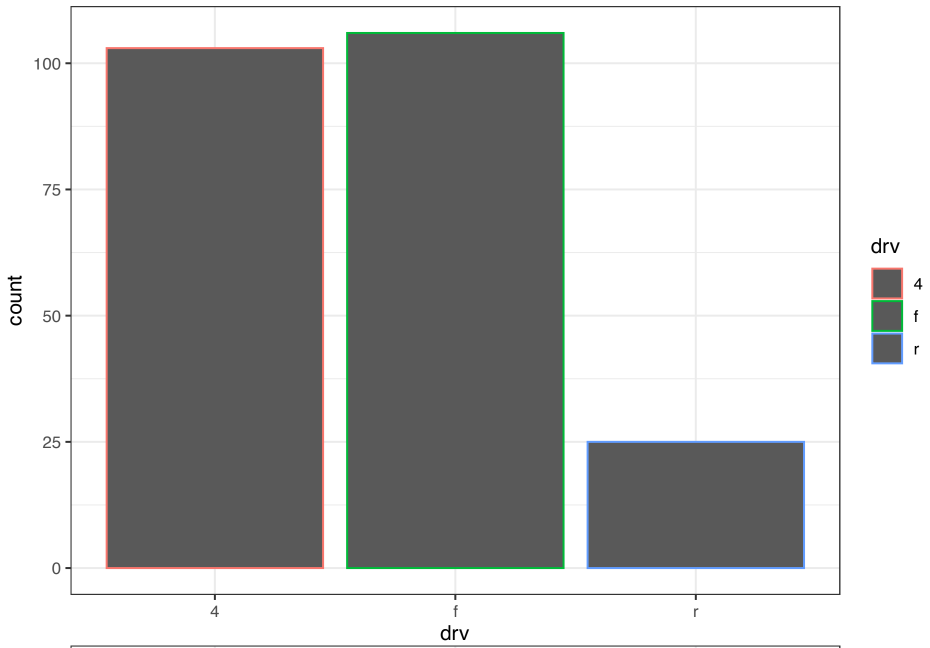
b. Replot the above two plots below without using `group = 1` and use 'after_stat(count)' to calculate proportions instead of 'after_stat(prop)'. Label the y-axis using percentages. Hint: check out the scales package that we loaded.

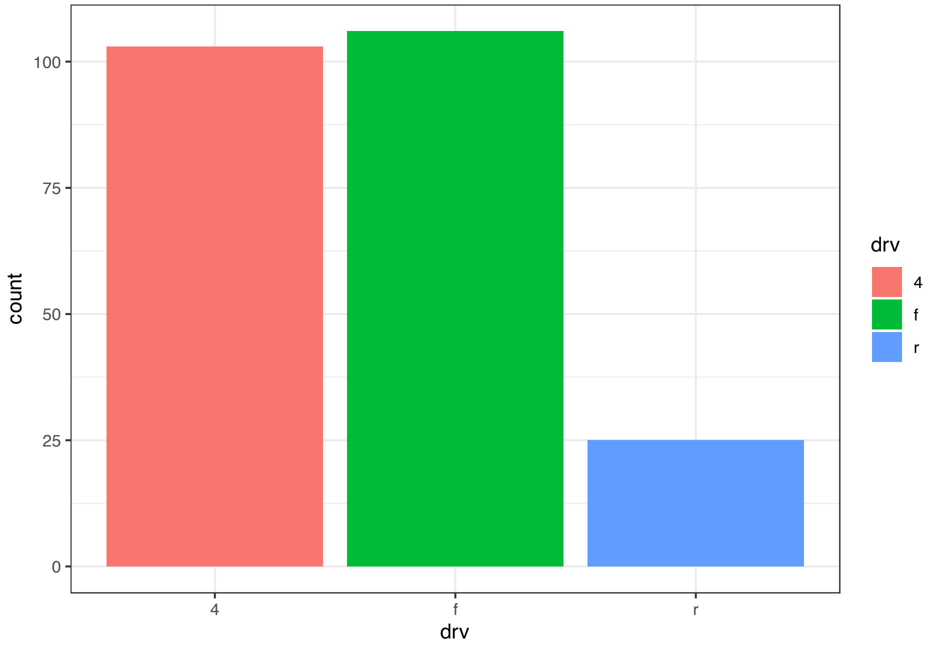
**Ans-2.5.1.5b:**

## 2.6 Position adjustments

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

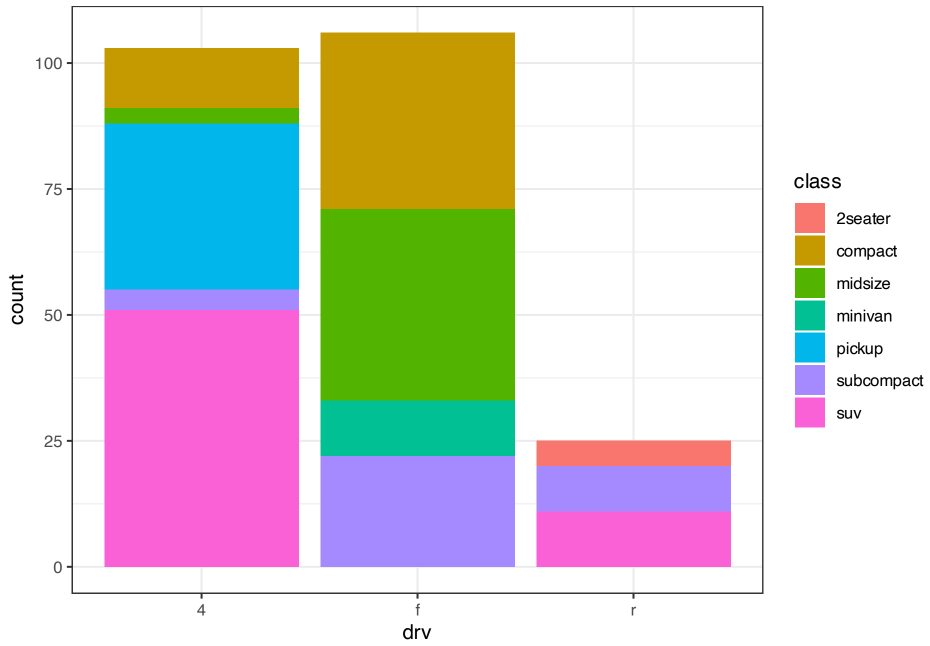
```r
# Top
ggplot(mpg, aes(x = drv, color = drv)) + geom_bar()

# Bottom
ggplot(mpg, aes(x = drv, fill = drv)) + geom_bar()
```





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

```
ggplot(mpg, aes(x = drv, fill = class)) + geom_bar()
```

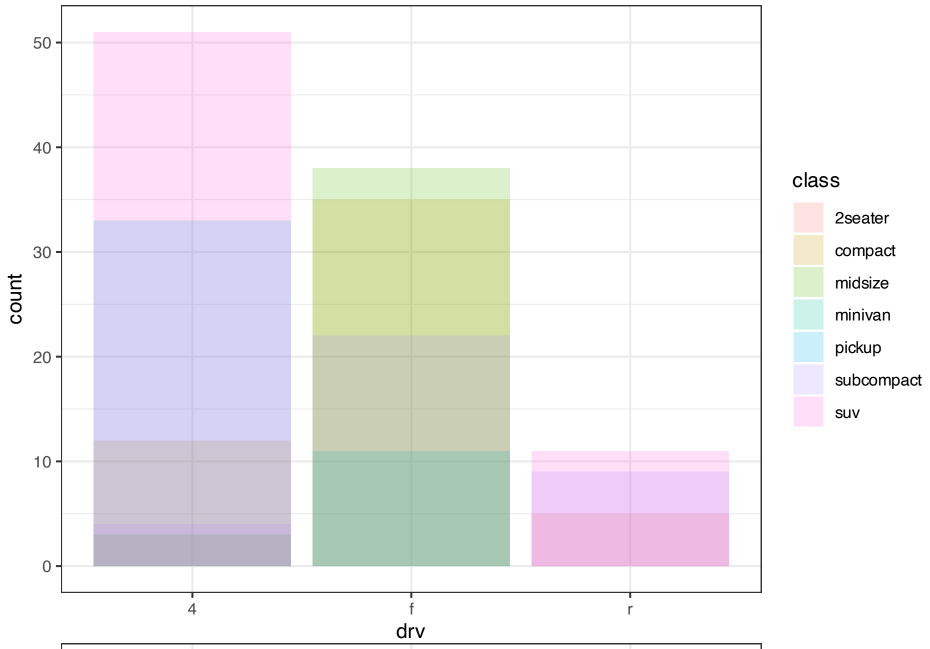


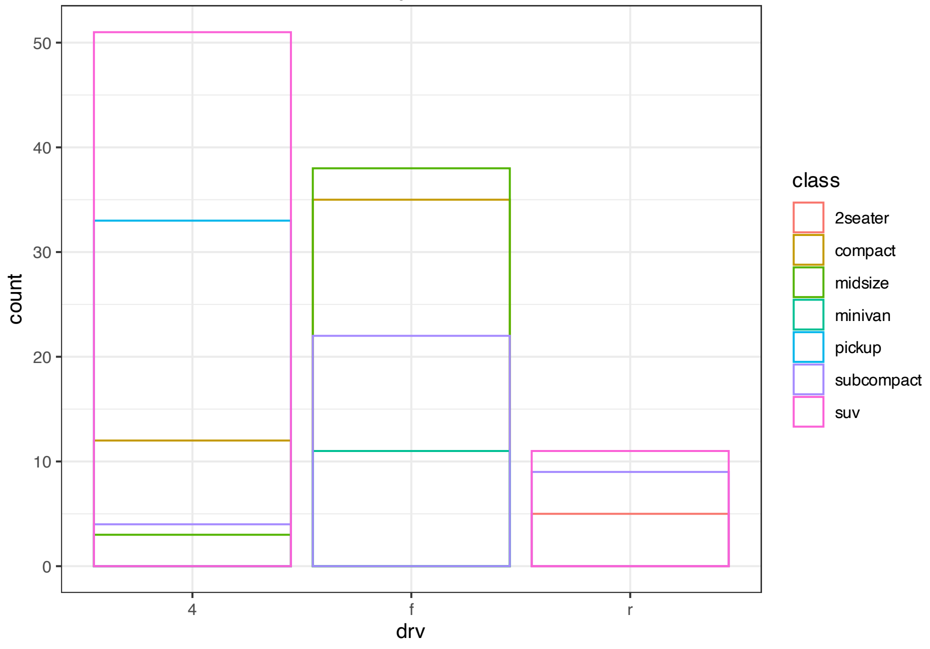
The stacking is performed automatically using the **position adjustment** specified by the `position` argument. If you don't want a stacked bar chart, you can use one of three other options: `"identity"`, `"dodge"` or `"fill"`.

- `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`.

```
# | fig-alt: | | Segmented bar chart of drive types of
# cars, where each bar is filled with | colors for the
# classes of cars. Heights of the bars correspond to the |
# number of cars in each drive category, and heights of the
# colored | segments are proportional to the number of cars
# with a given class | level within a given drive type
# level. However the segments overlap. In | the first plot
# the bars are filled with transparent colors | and in the
# second plot they are only outlined with color.

# Top
ggplot(mpg, aes(x = drv, fill = class)) + geom_bar(alpha = 1/5,
    position = "identity")
```

```
# Bottom
ggplot(mpg, aes(x = drv, color = class)) + geom_bar(fill = NA,
    position = "identity")
```



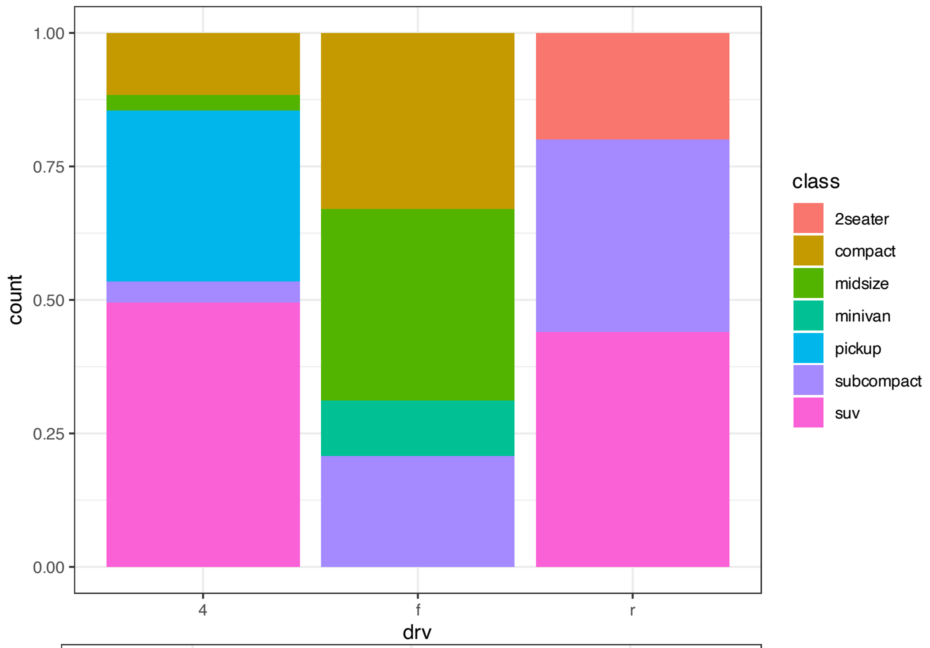


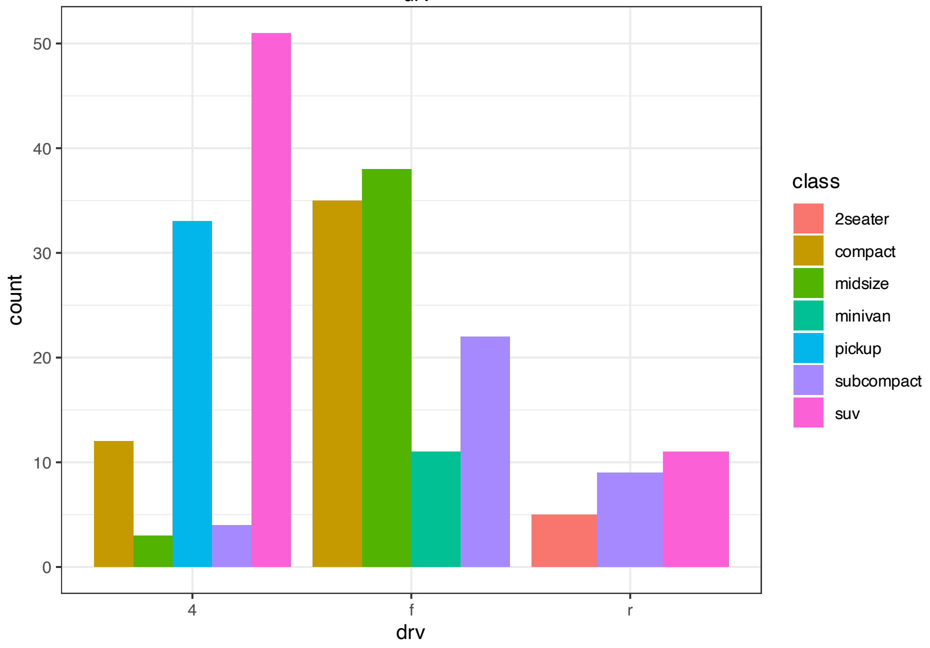
The identity position adjustment is more useful for 2d geoms, like points, where it is the default.

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

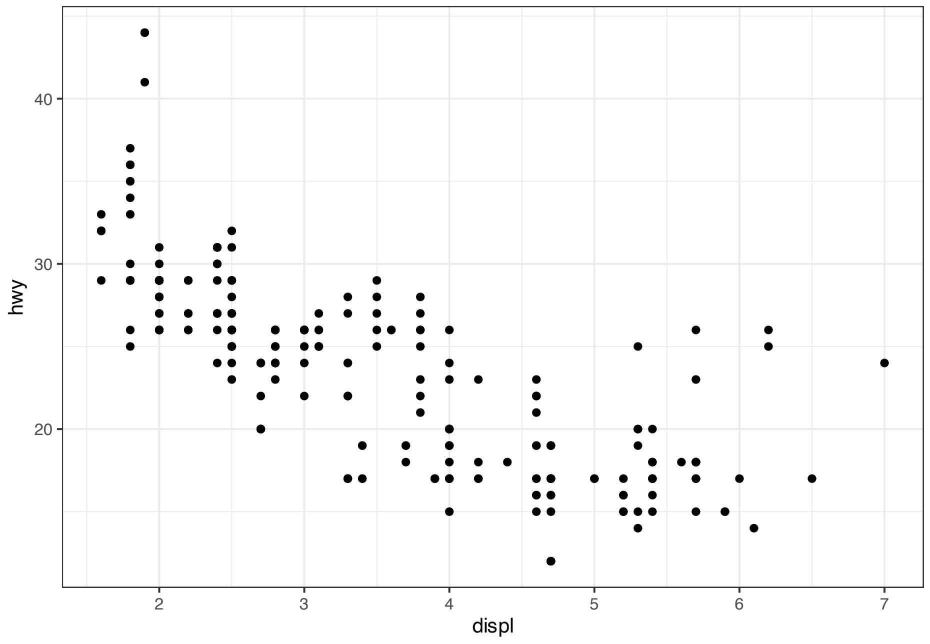
- `position = "dodge"` places overlapping objects directly *beside* one another. This makes it easier to compare individual values.

```
# | layout-ncol: 2 | fig-width: 4 | fig-alt: | | On the
# left, segmented bar chart of drive types of cars, where
# each bar is | filled with colors for the levels of class.
# Height of each bar is 1 and | heights of the colored
# segments represent the proportions of cars | with a given
# class level within a given drive type. | On the right,
# dodged bar chart of drive types of cars. Dodged bars are
# | grouped by levels of drive type. Within each group bars
# represent each | level of class. Some classes are
# represented within some drive types and | not represented
# in others, resulting in unequal number of bars within
# each | group. Heights of these bars represent the number
# of cars with a given | level of drive type and class.

# Left
ggplot(mpg, aes(x = drv, fill = class)) + geom_bar(position = "fill")

# Right
ggplot(mpg, aes(x = drv, fill = class)) + geom_bar(position = "dodge")
```





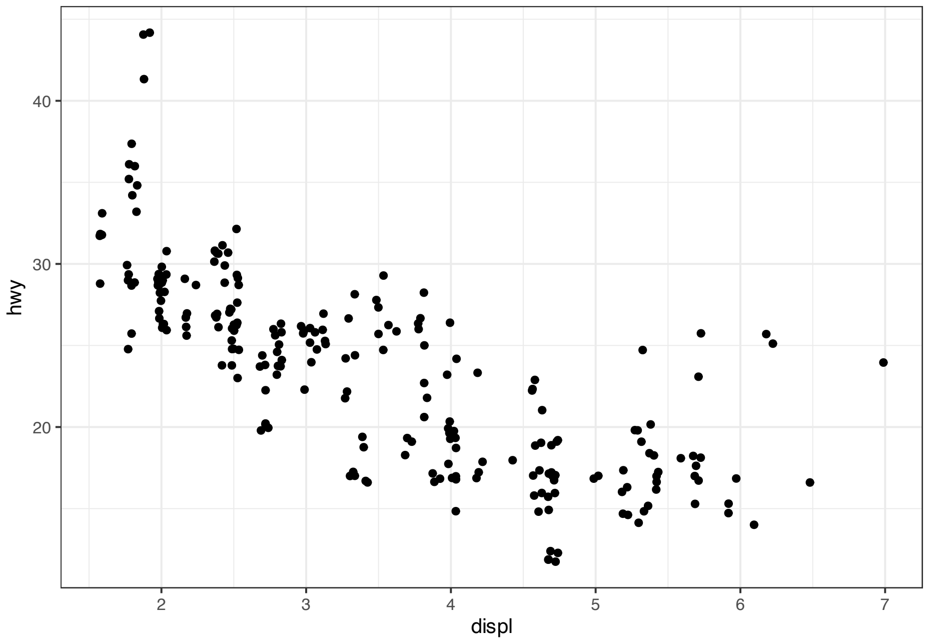
There's one other type of adjustment that's not useful for bar charts, but can be very useful for scatterplots. Recall our first scatterplot. Did you notice that the plot displays only 126 points, even though there are 234 observations in the dataset?



The underlying 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 difficult to see the distribution of the data. 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.

```r
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(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`.

### 2.6.1 Exercises

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

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

**Ans-2.6.1.1:**

2. What, if anything, is the difference between these two plots? Why?

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

ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(position = "identity")
```

**Ans-2.6.1.2:**

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

**Ans-2.6.1.3:**

4. Compare and contrast `geom_jitter()` with `geom_count()` using the following ggplot.

```
g <- ggplot(mpg, aes(x = displ, y = hwy))
```

**Ans-2.6.1.4:**

5. What's the default position adjustment for `geom_boxplot()`? Create a visualization of the `mpg` dataset that demonstrates it by adding the default position using the following ggplot:

```
g <- ggplot(data = mpg, mapping = aes(x = manufacturer, y = hwy,
    color = manufacturer))
```

**Ans-2.6.1.5:**

## 2.7 Grammar of graphics

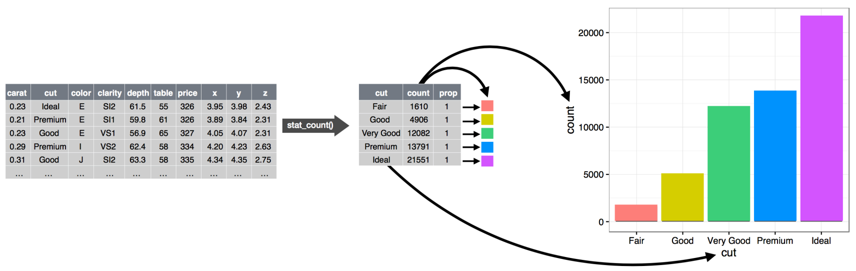
We can expand on the graphing template you've learned by adding position adjustments, stats, coordinate systems, and faceting:

```
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, a faceting scheme, and a theme.

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). 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. These steps are illustrated in 2.4. You'd then select a coordinate system to place the geoms into, using the location of the objects (which is itself an aesthetic property) to display the values of the x and y variables.



| 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 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

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

Figure 2.4: Steps for going from raw data to a table of frequencies to a bar plot where the heights of the bar represent the frequencies.

At this 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.

If you'd like to learn more about the theoretical underpinnings of ggplot2, you might enjoy reading "The Layered Grammar of Graphics", the scientific paper that describes the theory of ggplot2 in detail.

## 2.8 Summary

In this chapter you learned about the layered grammar of graphics starting with aesthetics and geometries to build a simple plot, facets for splitting the plot into subsets, statistics for understanding how geoms are calculated, position adjustments for controlling the fine details of position when geoms might otherwise overlap, and coordinate systems which allow you to fundamentally change what `x` and `y` mean.

Two very useful resources for getting an overview of the complete ggplot2 functionality are the ggplot2 cheatsheet (which you can find at https://posit.co/resources/cheatsheets) and the ggplot2 package website (https://ggplot2.tidyverse.org).

An important lesson you should take from this chapter is that when you feel the need for a geom that is not provided by ggplot2, it's always a good idea to look into whether someone else has already solved your problem by creating a ggplot2 extension package that offers that geom.