Exercise Solutions to "R for Data Science"

Jeffrey B. Arnold

Contents

Welcome

This contains my exercise solutions for Hadley Wickham and Garret Grolemund, R for Data Science. The original website is at r4ds.had.co.nz.

Note that the chapter and section numbers don't exactly match those of R4DS. Refer to the titles, not the numbers.

Chapter 1

Introduction

No exercises.

Part I

Explore

Chapter 2

Introduction

No exercises.

Chapter 3

Visualize

3.1 Introduction

```
library("tidyverse")
```

No exercises.

3.2 First Steps

3.2.1 Exercise 1.

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

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


An empty plot. The background of the plot is created by `ggplot()`, but nothing else is displayed.

3.2.2 Exercise 2.

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

There are 32 rows and 11 columns in the the `mtcars` data frame.

```
nrow(mtcars)
#> [1] 32
ncol(mtcars)
#> [1] 11
```

The number of rows and columns is also displayed by `glimpse()`:

```
glimpse(mtcars)
#> Observations: 32
#> Variables: 11
#> $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19....
#> $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, ...
#> $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 1...
#> $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, ...
#> $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.9...
#> $ wt   <dbl> 2.62, 2.88, 2.32, 3.21, 3.44, 3.46, 3.57, 3.19, 3.15, 3.4...
#> $ qsec <dbl> 16.5, 17.0, 18.6, 19.4, 17.0, 20.2, 15.8, 20.0, 22.9, 18....
#> $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, ...
#> $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
#> $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, ...
#> $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, ...
```

3.2.3 Exercise 3.

What does the `drv` variable describe? Read the help for `?mpg` to find out.

```
?mpg
```

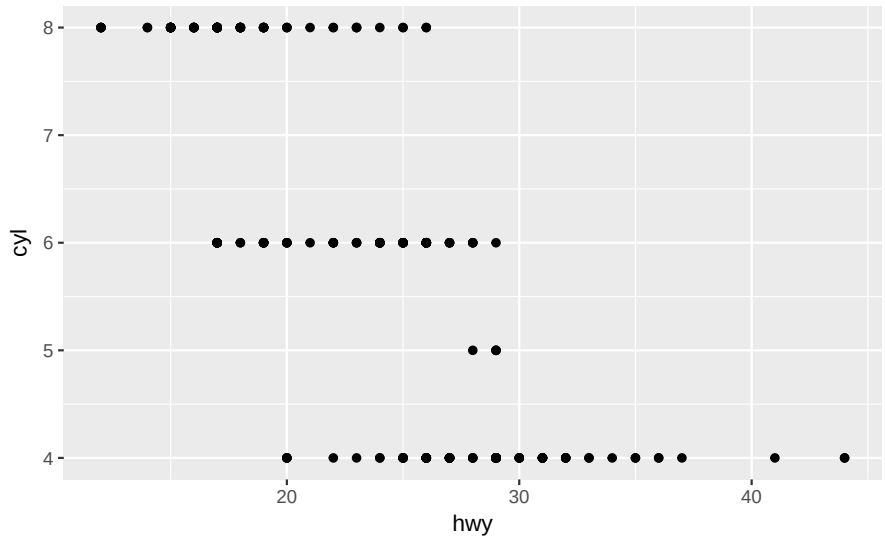
The `drv` variable takes the following values

- `"f"`	front-wheel drive
- `"r"`	rear-wheel drive
- `"4"`	four-wheel drive

3.2.4 Exercise 4.

Make a scatter plot of `hwy` vs `cyl`

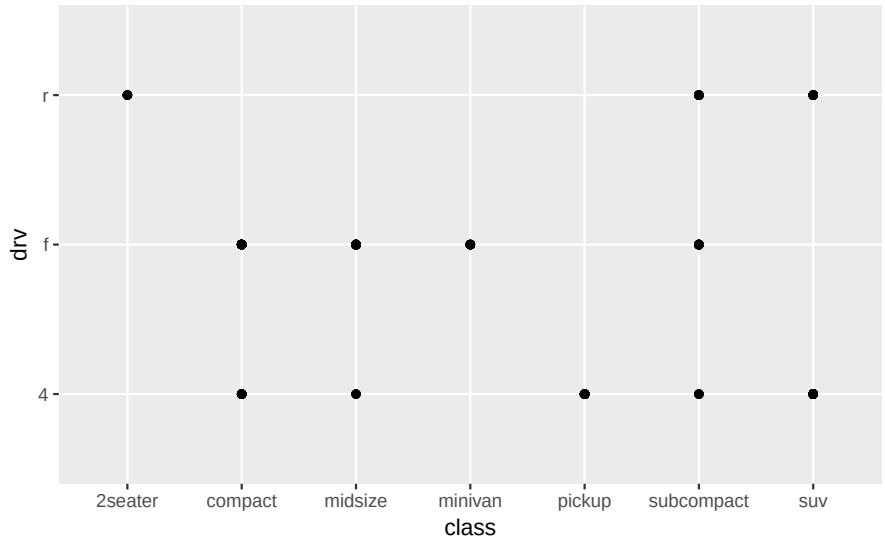
```
ggplot(mpg, aes(x = hwy, y = cyl)) +
  geom_point()
```



3.2.5 Exercise 5.

What happens if you make a scatter plot of `class` vs `drv`. Why is the plot not useful?

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



A scatter plot is not a useful way to plot these variables, since both `drv` and `class` are factor variables taking a limited number of values.

```
count(mpg, drv, class)
#> # A tibble: 12 x 3
#>   drv   class          n
#>   <chr> <chr>      <int>
#> 1 4     compact       12
#> 2 4     midsize        3
#> 3 4     pickup        33
#> 4 4     subcompact     4
#> 5 4     suv           51
```

```
#> 6 f      compact       35
#> # ... with 6 more rows
```

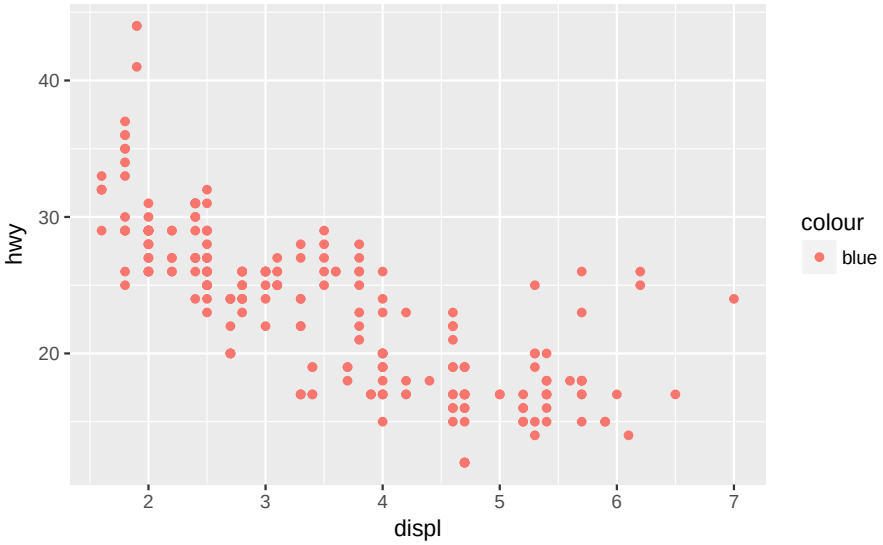
The scatter plot cannot show which are overlapping or not. Later chapters discuss means to deal with this, including alternative plots and jittering the points so they don't overlap.

3.3 Aesthetic mappings

3.3.1 Exercise 1.

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

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



Since `color = "blue"` was included within the `mapping` argument, it was treated as an aesthetic (a mapping between a variable and a value). The expression, `color="blue"`, treats `"blue"` as a variable with only one value: `"blue"`. If this is confusing, consider how `color = 1:234` or `color = 1` would be interpreted by `aes()`.

3.3.2 Exercise 2.

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

```
?mpg
```

When printing the data frame, this information is given at the top of each column within angled brackets. Categorical variables have a class of "character" (`<chr>`).

```
mpg
#> # A tibble: 234 x 11
#>   manufacturer model displ  year   cyl trans drv     cty   hwy fl    class
#>   <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
#> 1 audi         a4     1.80  1999     4 auto~ f        18    29 p     comp~
#> 2 audi         a4     1.80  1999     4 manu~ f        21    29 p     comp~
#> 3 audi         a4     2.00  2008     4 manu~ f        20    31 p     comp~
```

```
#> 4 audi          a4      2.00  2008     4 auto~ f        21    30 p     comp~
#> 5 audi          a4      2.80  1999     6 auto~ f        16    26 p     comp~
#> 6 audi          a4      2.80  1999     6 manu~ f        18    26 p     comp~
#> # ... with 228 more rows
```

Alternatively, the `glimpse` function displays the type of each column:

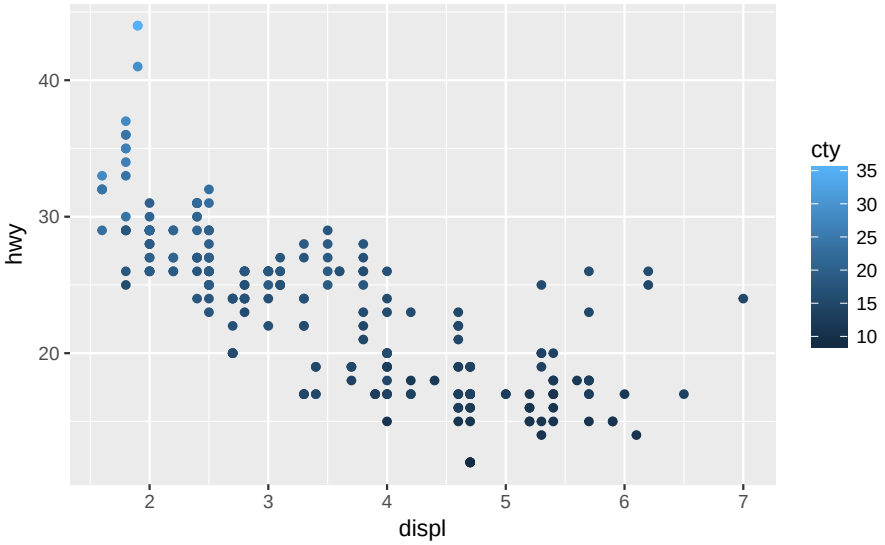
```
glimpse(mpg)
#> Observations: 234
#> Variables: 11
#> $ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "...
#> $ model        <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 qua...
#> $ displ        <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0,...
#> $ year         <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1...
#> $ cyl          <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6...
#> $ trans        <chr> "auto(l5)", "manual(m5)", "manual(m6)", "auto(av)...
#> $ drv          <chr> "f", "f", "f", "f", "f", "f", "f", "4", "4", "4",...
#> $ cty          <int> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 1...
#> $ hwy          <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 2...
#> $ fl           <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p",...
#> $ class        <chr> "compact", "compact", "compact", "compact", "comp...
```

3.3.3 Exercise 3.

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

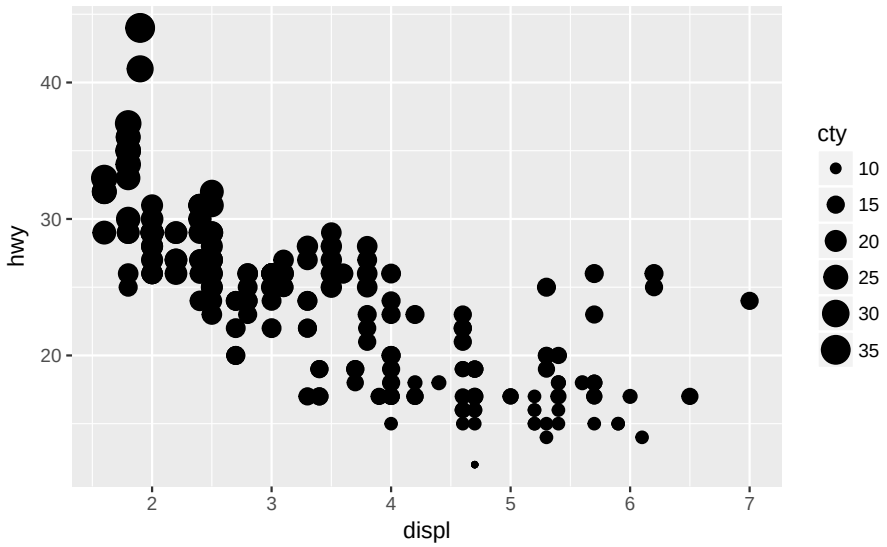
The variable `cty`, city highway miles per gallon, is a continuous variable:

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



Instead of using discrete colors, the continuous variable uses a scale that varies from a light to dark blue color.

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



When mapped to size, the sizes of the points vary continuously with respect to the size (although the legend shows a few representative values)

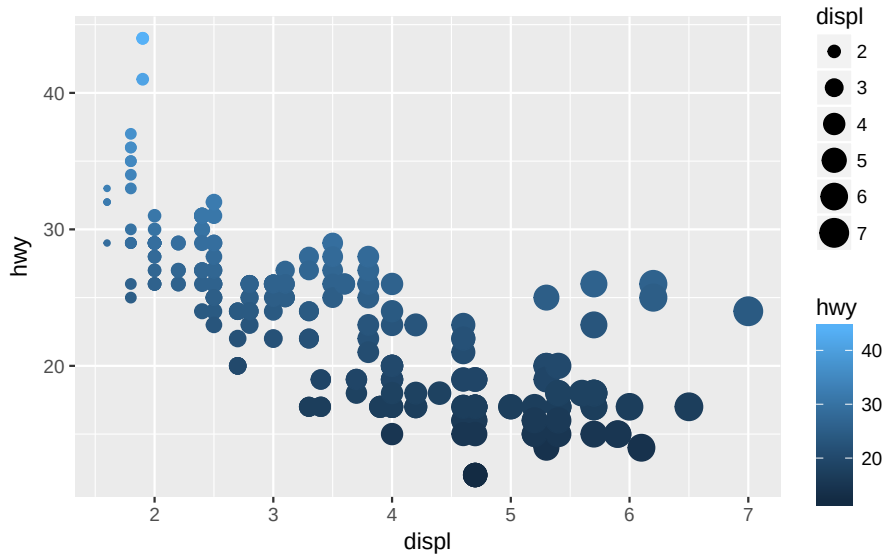
```
ggplot(mpg, aes(x = displ, y = hwy, shape = cty)) +
  geom_point()
#> Error: A continuous variable can not be mapped to shape
```

When a continuous value is mapped to shape, it gives an error. Though we could split a continuous variable into discrete categories and use a shape aesthetic, this would conceptually not make sense. A continuous numeric variable is ordered, but shapes have no natural order. It is clear that smaller points correspond to smaller values, or once the color scale is given, which colors correspond to larger or smaller values. But it is not clear whether a square is greater or less than a circle.

3.3.4 Exercise 4.

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

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



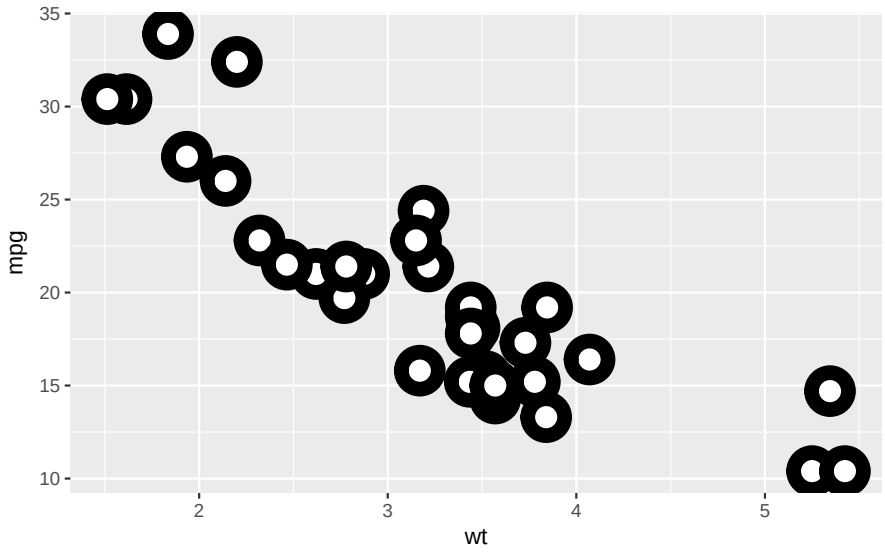
In the above plot, `hwy` is mapped to both location on the y-axis and color, and `displ` is mapped to both location on the x-axis and size. The code works and produces a plot, even if it is a bad one. Mapping a single variable to multiple aesthetics is redundant. Because it is redundant information, in most cases avoid mapping a single variable to multiple aesthetics.

3.3.5 Exercise 5.

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

The following example is given in `?geom_point`:

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(shape = 21, colour = "black", fill = "white", size = 5, stroke = 5)
```

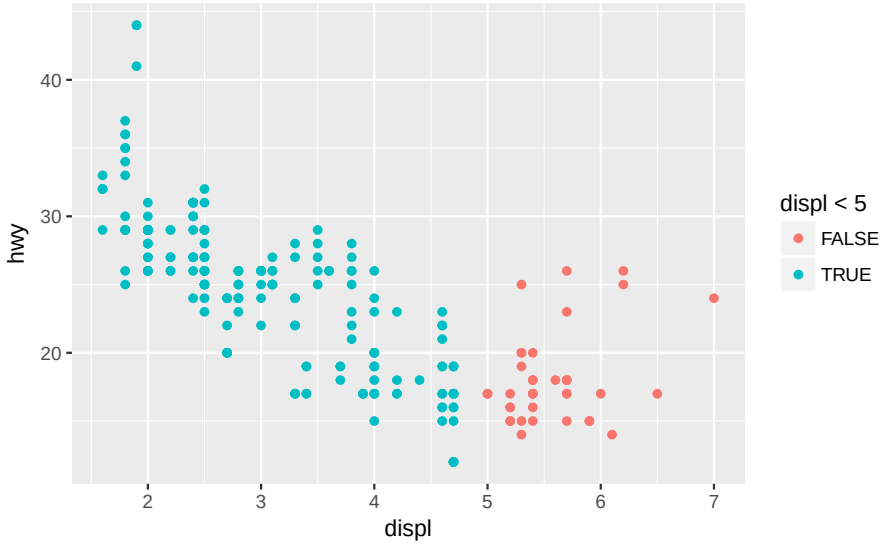


Stroke changes the color of the border for shapes (22-24).

3.3.6 Exercise 6.

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

```
ggplot(mpg, aes(x = displ, y = hwy, colour = displ < 5)) +
  geom_point()
```



Aesthetics can also be mapped to expressions (code like `displ < 5`). It will create a temporary variable which takes values from the result of the expression. In this case, it is logical variable which is `TRUE` or `FALSE`. This also explains exercise 1, `color = "blue"` created a categorical variable that only had one category: "blue".

3.4 Common problems

No exercises

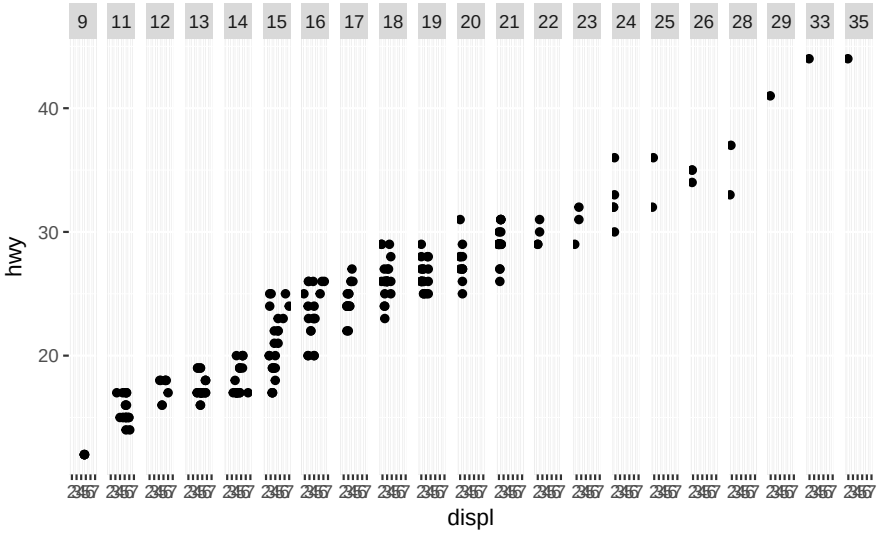
3.5 Facets

3.5.1 Exercise 1.

What happens if you facet on a continuous variable?

Let's see.

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



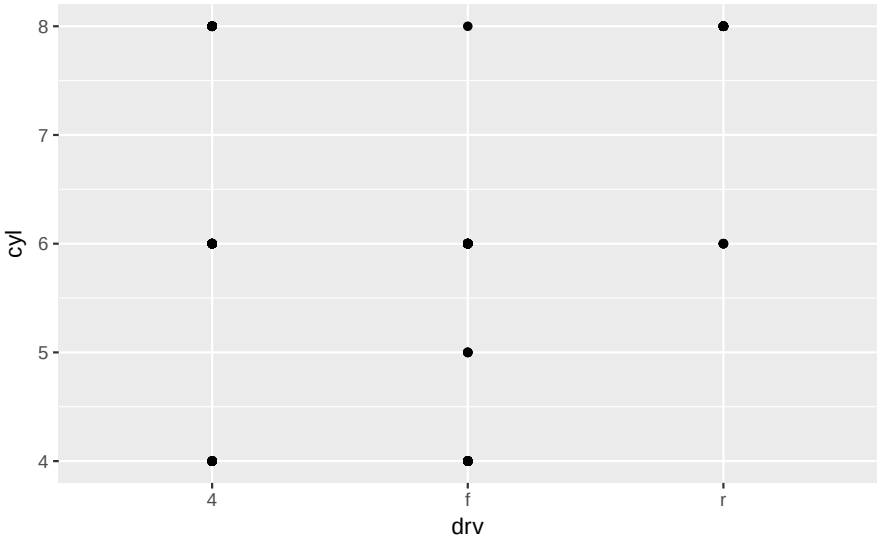
It converts the continuous variable to a factor and creates facets for **all** unique values of it.

3.5.2 Exercise 2.

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

They are cells in which there are no values of the combination of `drv` and `cyl`.

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



The locations in the above plot without points are the same cells in `facet_grid(drv ~ cyl)` that have no points.

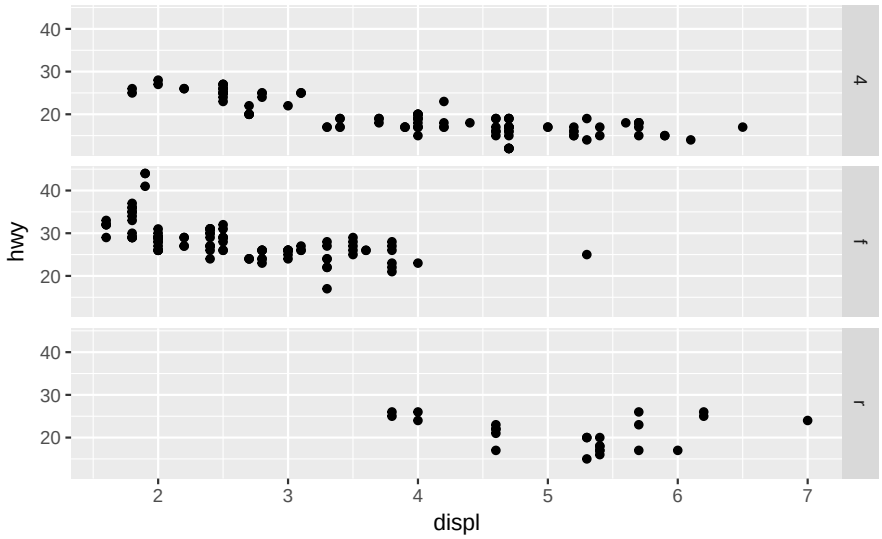
3.5.3 Exercise 3.

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

The symbol `.` ignores that dimension for faceting.

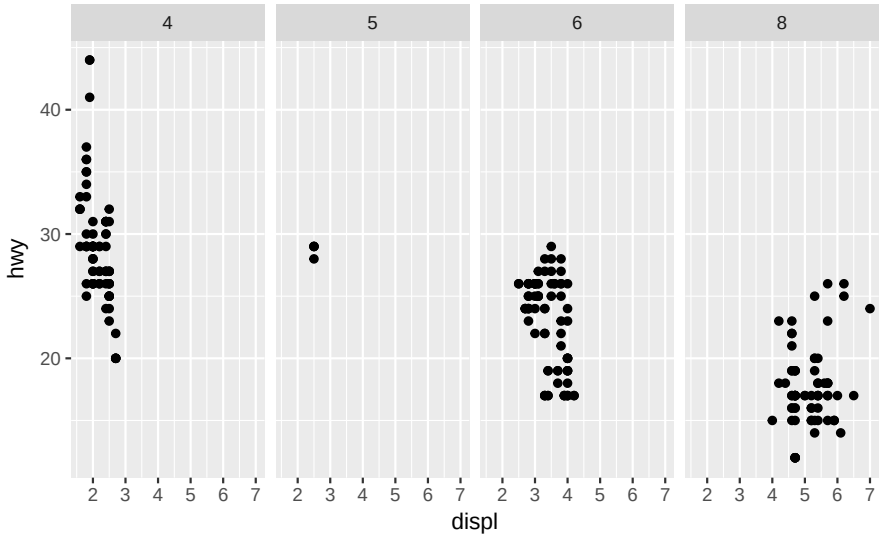
This plot facets by values of `drv` on the y-axis:

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



This plot facets by values of `cyl` on the x-axis:

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



3.5.4 Exercise 5.

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

The arguments `nrow` (`ncol`) determines the number of rows (columns) to use when laying out the facets. It is necessary since `facet_wrap` only facets on one variable. These arguments are unnecessary for `facet_grid` since the number of rows and columns are determined by the number of unique values of the variables

specified.

3.5.5 Exercise 6.

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

You should put the variable with more unique levels in the columns if the plot is laid out landscape. It is easier to compare relative levels of y by scanning horizontally, so it may be easier to visually compare these levels. *I'm actually not sure about the correct answer to this.*

3.6 Geometric Objects

3.6.1 Exercise 1.

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

- line chart: `geom_line`
- boxplot: `geom_boxplot`
- histogram: `geom_hist`

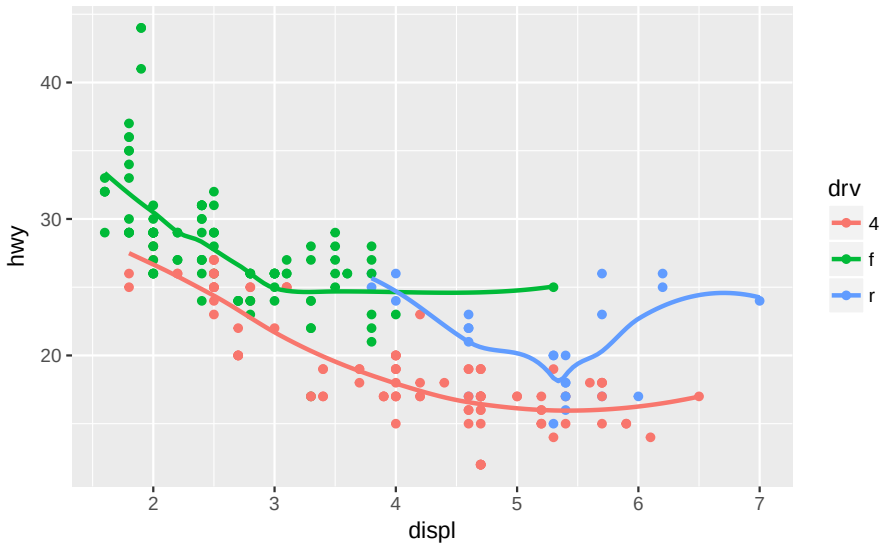
3.6.2 Exercise 2.

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

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

This will produce a scatter plot with `displ` on the x-axis, `hwy` on the y-axis. The points will be colored by `drv`. There will be a smooth line, without standard errors, fit through each `drv` group.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```

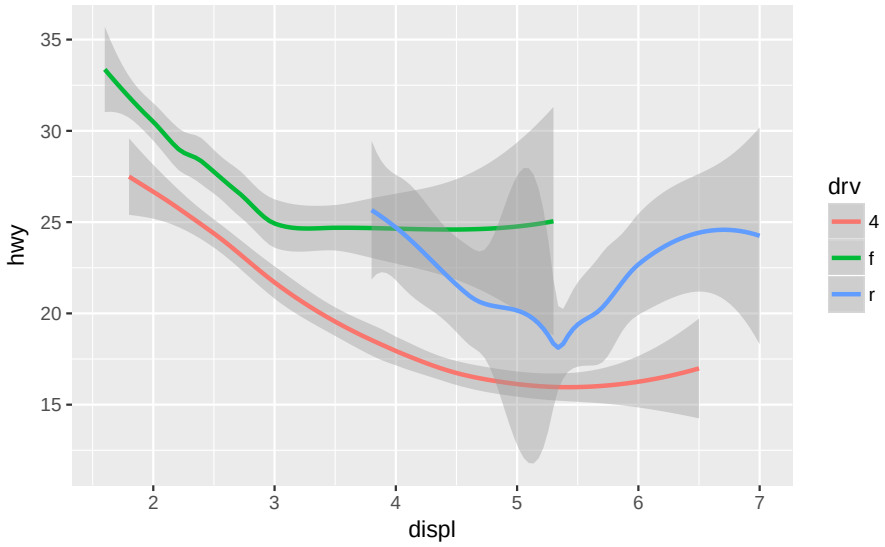


3.6.3 Exercise 3.

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

Show legend hides the legend box. In this code, without show legend, there is a legend.

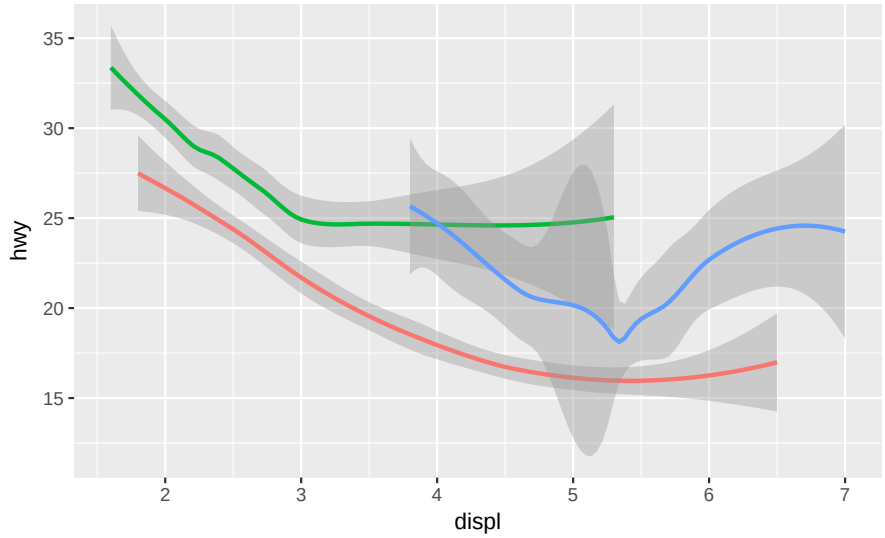
```
ggplot(data = mpg) +
  geom_smooth(
    mapping = aes(x = displ, y = hwy, color = drv),
  )
#> `geom_smooth()` using method = 'loess'
```



But there is no legend in this code:

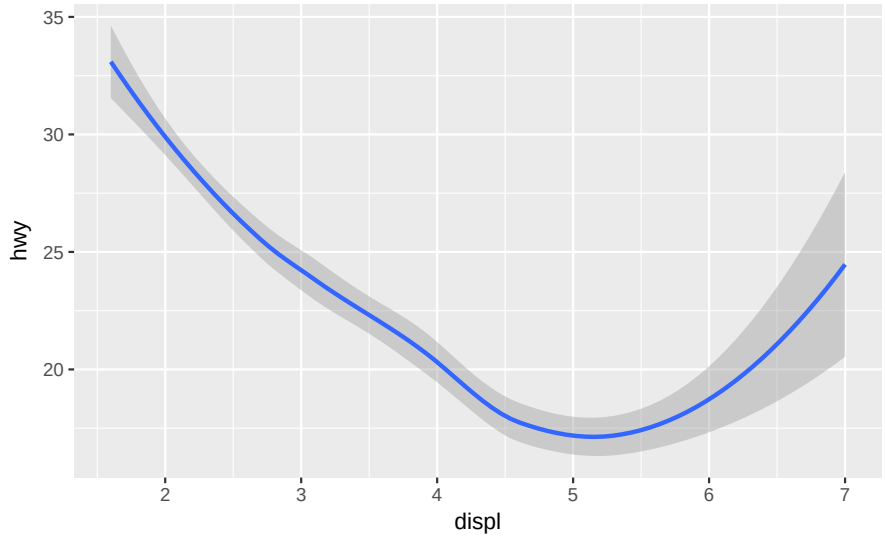
```
ggplot(data = mpg) +
  geom_smooth(
    mapping = aes(x = displ, y = hwy, color = drv),
    show.legend = FALSE
```

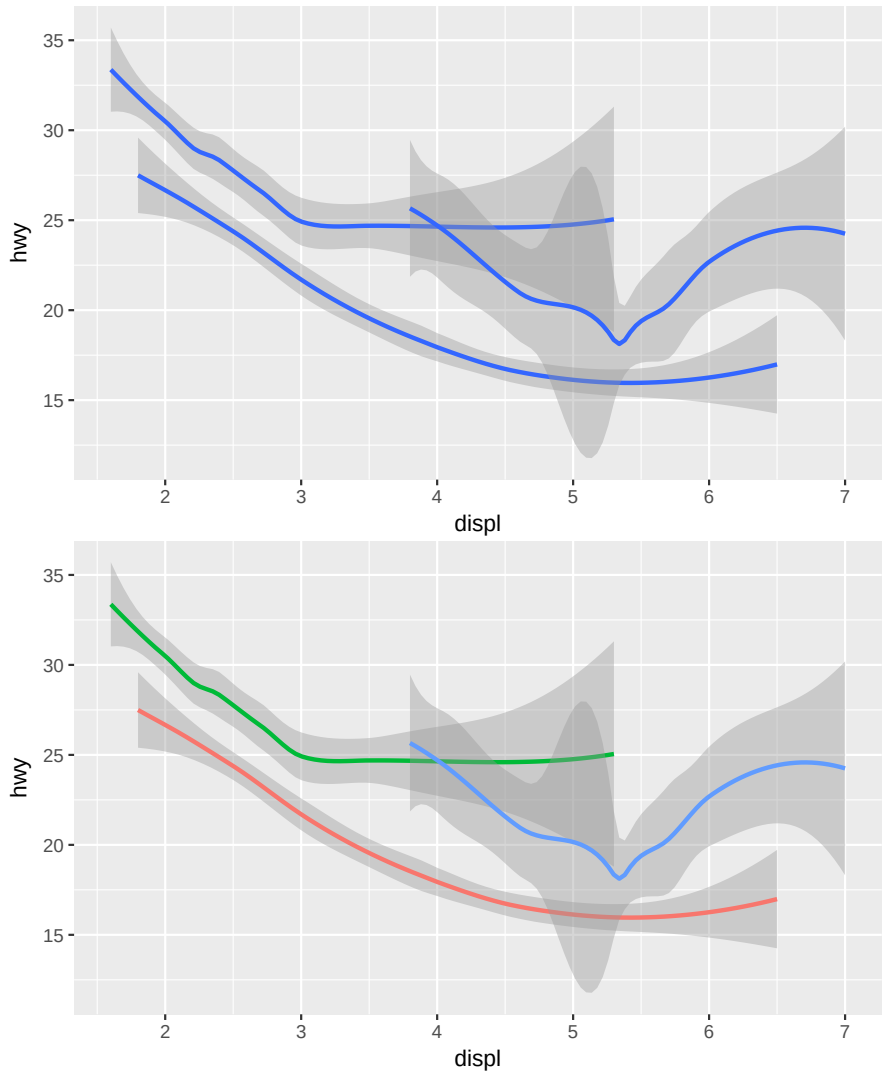
```
  )
#> `geom_smooth()` using method = 'loess'
```



In the example earlier in the chapter,

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
#> `geom_smooth()` using method = 'loess'

ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, group = drv))
#> `geom_smooth()` using method = 'loess'

ggplot(data = mpg) +
  geom_smooth(
    mapping = aes(x = displ, y = hwy, color = drv),
    show.legend = FALSE
  )
#> `geom_smooth()` using method = 'loess'
```





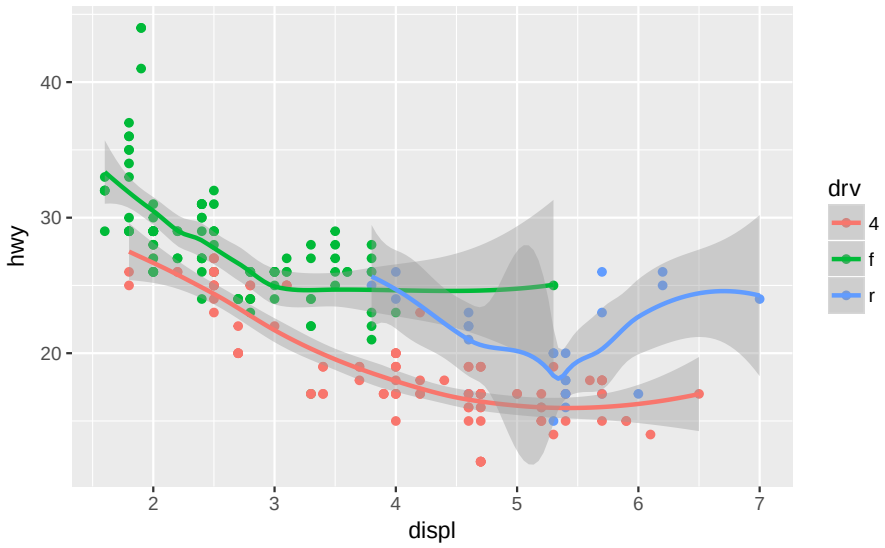
the legend is suppressed because there are three plots, and adding a legend that only appears in the last one would make the presentation asymmetric. Additionally, the purpose of this plot is to illustrate the difference between not grouping, using a `group` aesthetic, and using a `color` aesthetic (with implicit grouping). In that example, the legend isn't necessary since looking up the values associated with each color isn't necessary to make that point.

3.6.4 Exercise 4.

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

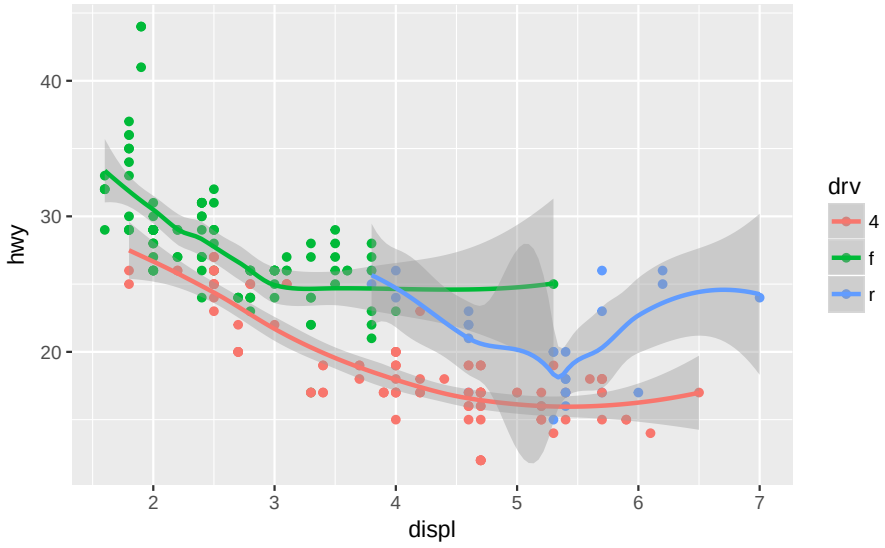
It adds standard error bands to the lines.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  geom_smooth(se = TRUE)
#> `geom_smooth()` using method = 'loess'
```



By default `se = TRUE`:

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'loess'
```

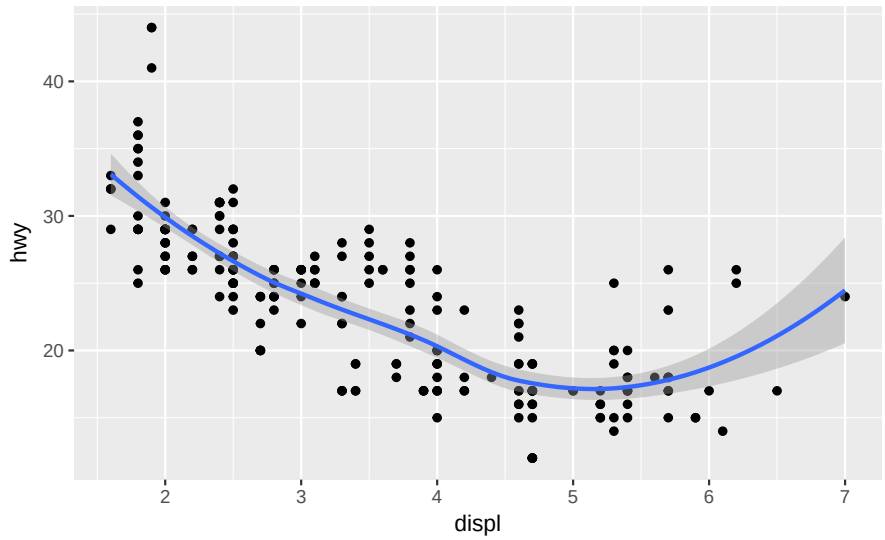


3.6.5 Exercise 5.

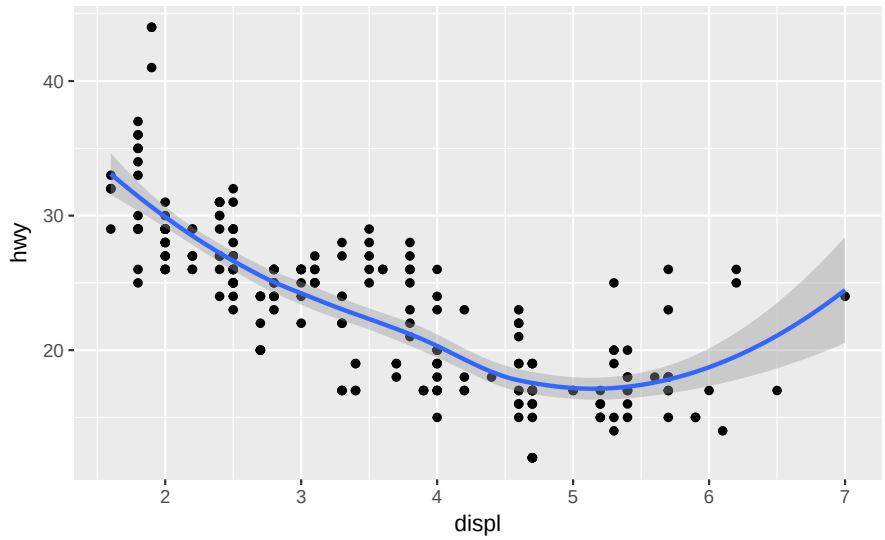
Will these two graphs look different? Why/why not?

No. Because both `geom_point` and `geom_smooth` use the same data and mappings. They will inherit those options from the `ggplot` object, and thus don't need to specified again (or twice).

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'loess'
```



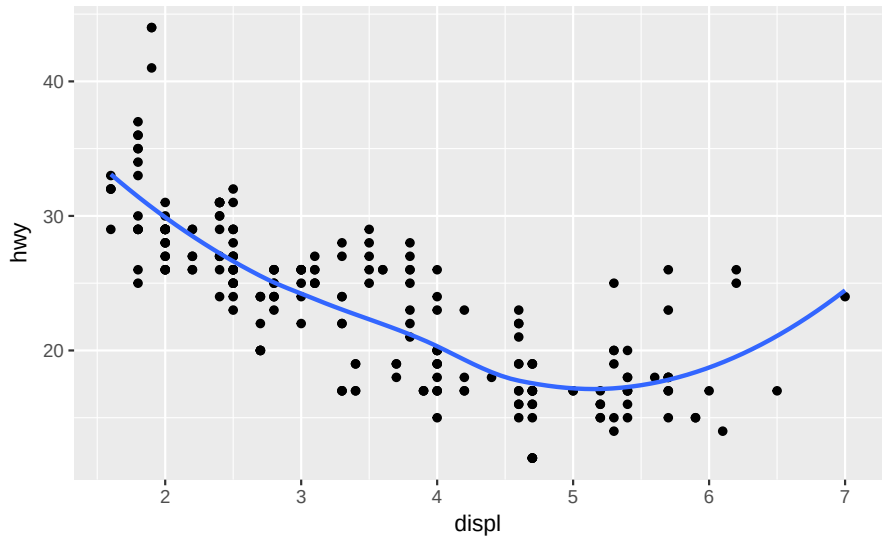

```
ggplot() +
  geom_point(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_smooth(data = mpg, mapping = aes(x = displ, y = hwy))
#> `geom_smooth()` using method = 'loess'
```



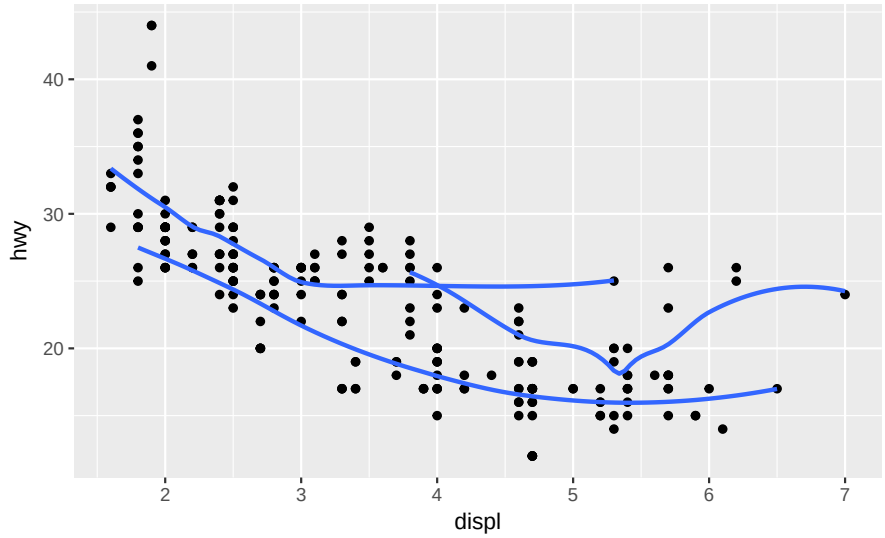
3.6.6 Exercise 6.

Recreate the R code necessary to generate the following graphs.

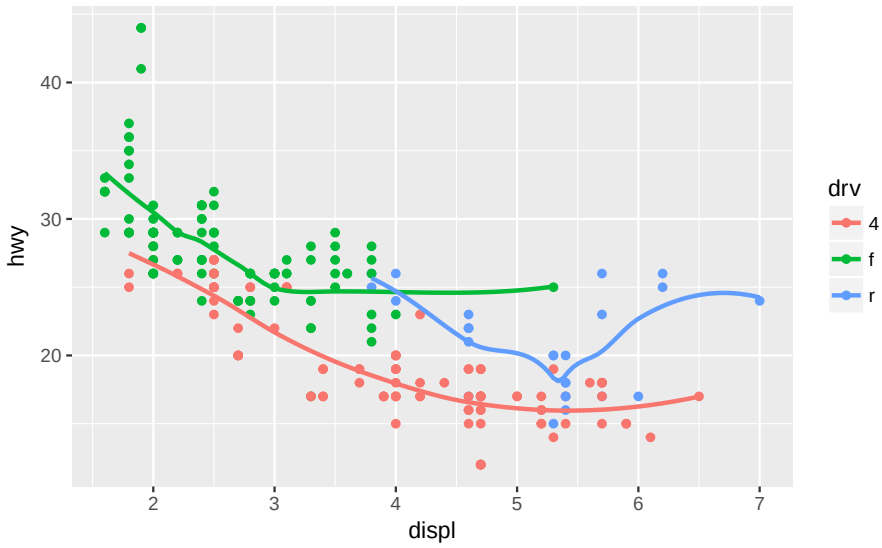
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```



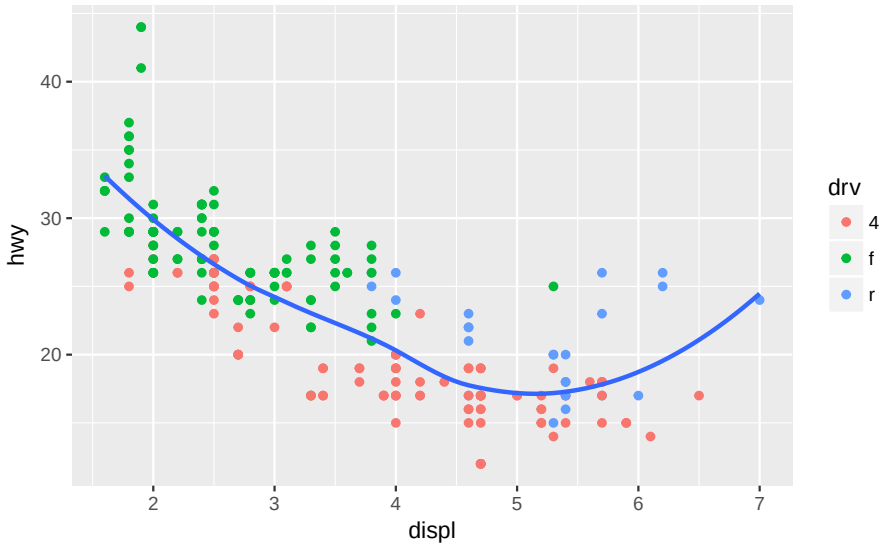

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(mapping = aes(group = drv), se = FALSE)
#> `geom_smooth()` using method = 'loess'
```



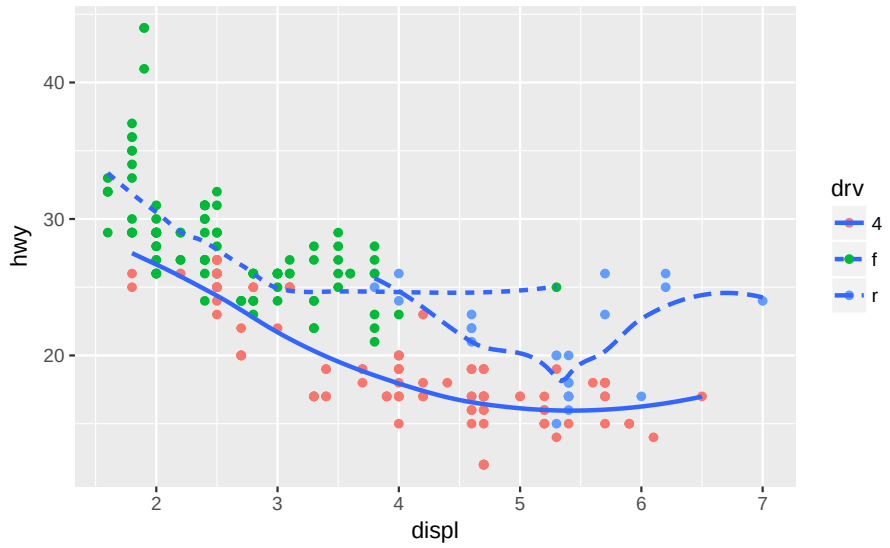

```
ggplot(mpg, aes(x = displ, y = hwy, colour = drv)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```



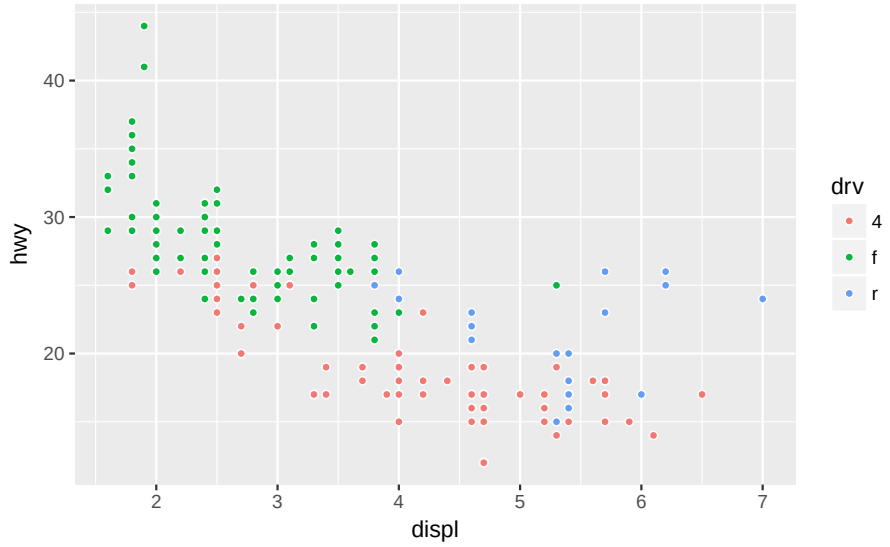

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(colour = drv)) +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```




```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(colour = drv)) +
  geom_smooth(aes(linetype = drv), se = FALSE)
#> `geom_smooth()` using method = 'loess'
```




```
ggplot(mpg, aes(x = displ, y = hwy, fill = drv)) +
  geom_point(color = "white", shape = 21)
```



3.7 Statistical Transformations

3.7.1 Exercise 1.

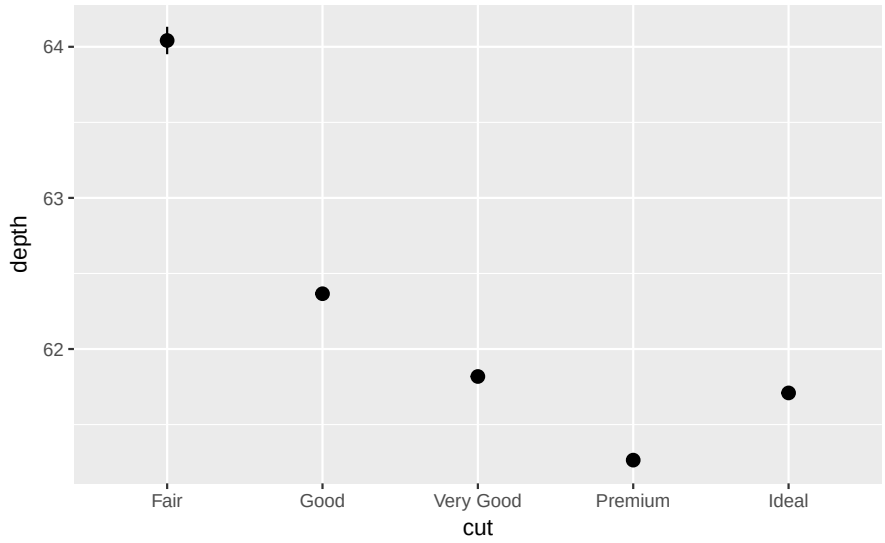
> What is the default geom associated with `stat_summary()`? How could you rewrite the previous plot to use that geom function instead of the stat function?

The default geom for `stat_summary` is `geom_pointrange` (see the `stat`) argument.

But, the default `stat` for `geom_pointrange` is `identity`, so use `geom_pointrange(stat = "summary")`.

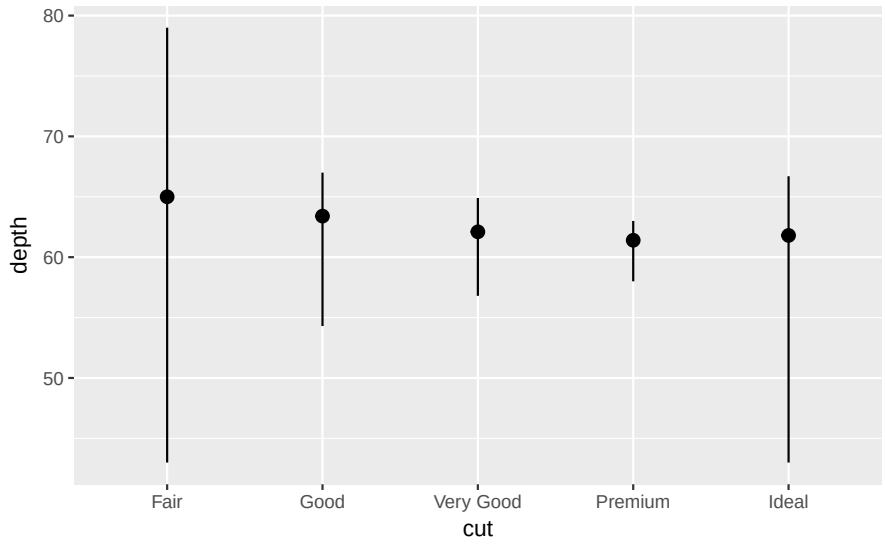
```
ggplot(data = diamonds) +
  geom_pointrange(
    mapping = aes(x = cut, y = depth),
    stat = "summary",
```

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



The default message says that `stat_summary` uses the `mean` and `sd` to calculate the point, and range of the line. So lets use the previous values of `fun.ymin`, `fun.ymax`, and `fun.y`:

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



3.7.2 Exercise 2.

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

`geom_col` differs from `geom_bar` in its default stat. `geom_col` has uses the `identity` stat. So it expects that a variable already exists for the height of the bars. `geom_bar` uses the `count` stat, and so will count observations in groups in order to generate the variable to use for the height of the bars.

3.7.3 Exercise 3.

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

See the ggplot2 documentation

3.7.4 Exercise 4.

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

`stat_smooth` calculates

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

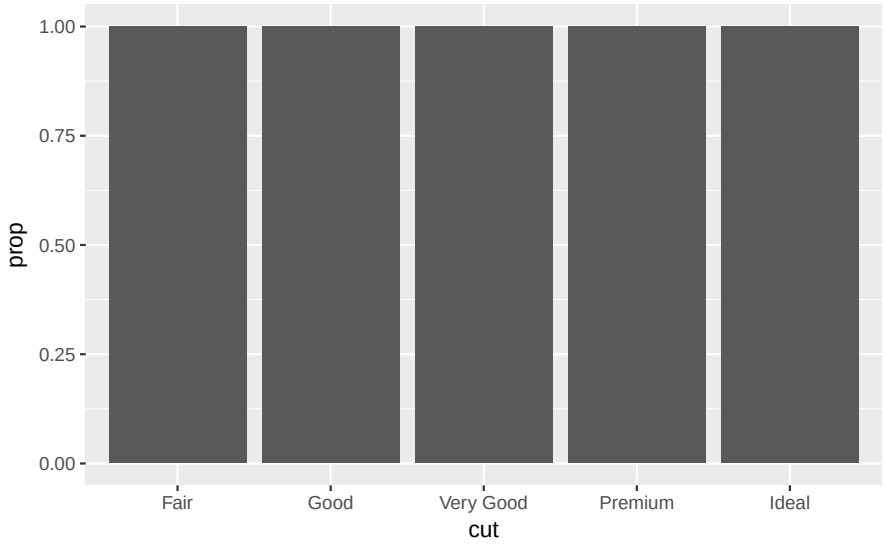
There's parameters such as `method` which determines which method is used to calculate the predictions and confidence interval, and some other arguments that are passed to that.

3.7.5 Exercise 5.

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

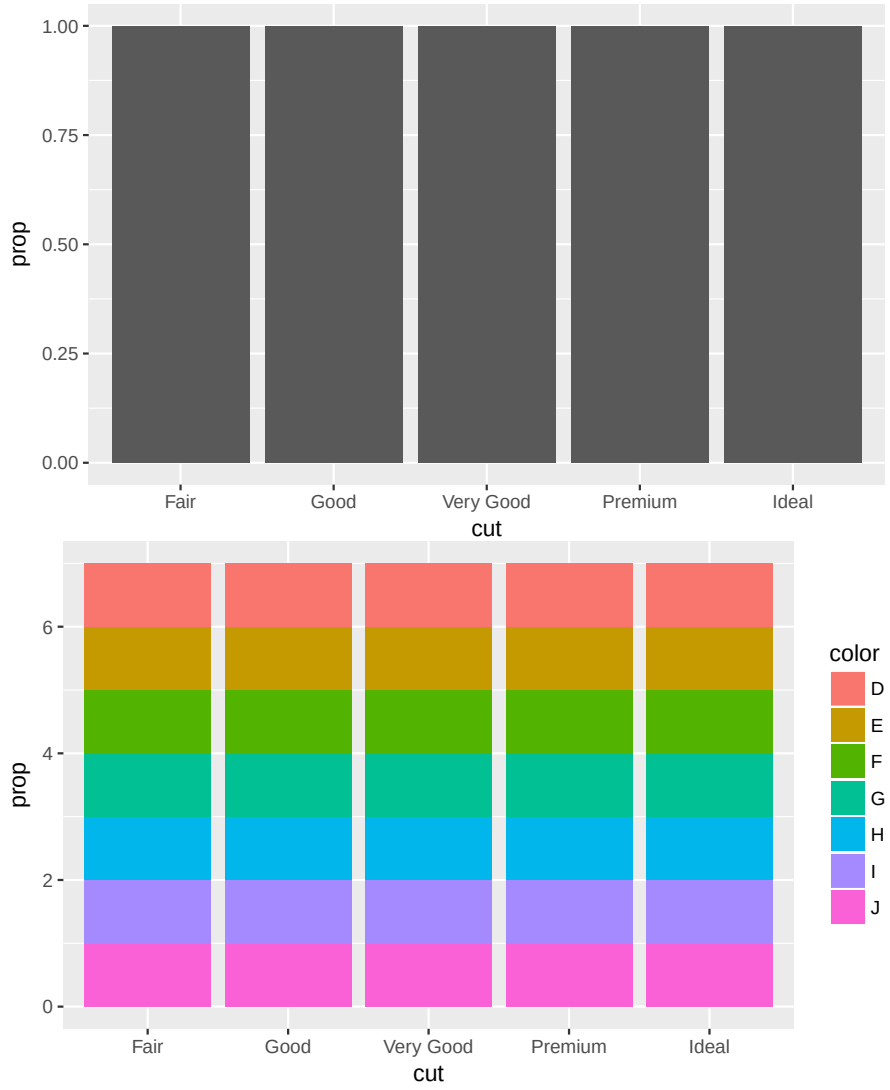
If `group` is not set to 1, then all the bars have `prop == 1`. The function `geom_bar` assumes that the groups are equal to the `x` values, since the stat computes the counts within the group.

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



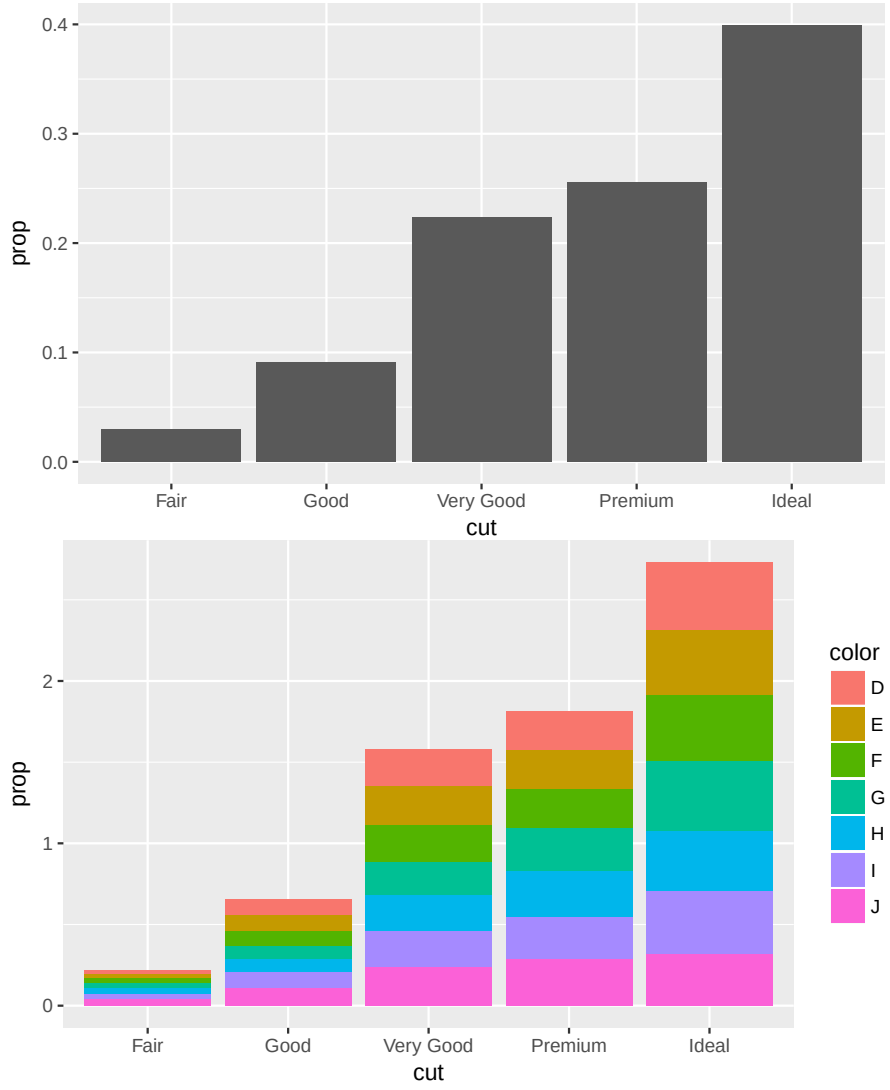
The problem with these two plots is that the proportions are calculated within the groups.

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

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



This is more likely what was intended:

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

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



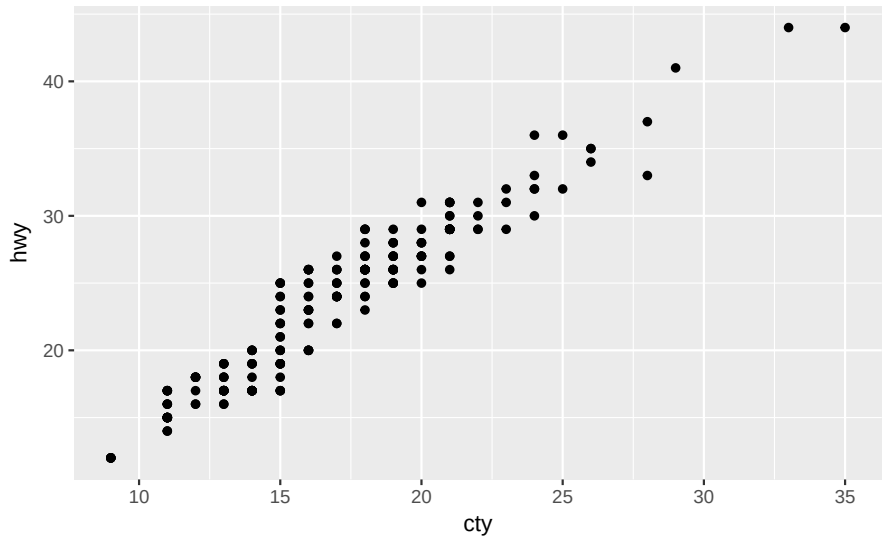
3.8 Position Adjustments

3.8.1 Exercise 1.

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

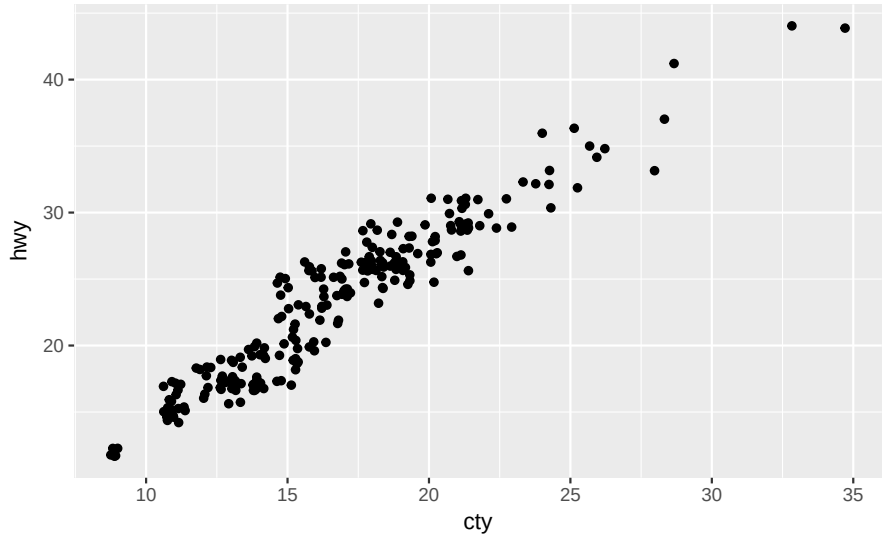
There is overplotting because there are multiple observations for each combination of `cty` and `hwy`.

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



I'd fix it by using a jitter position adjustment.

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



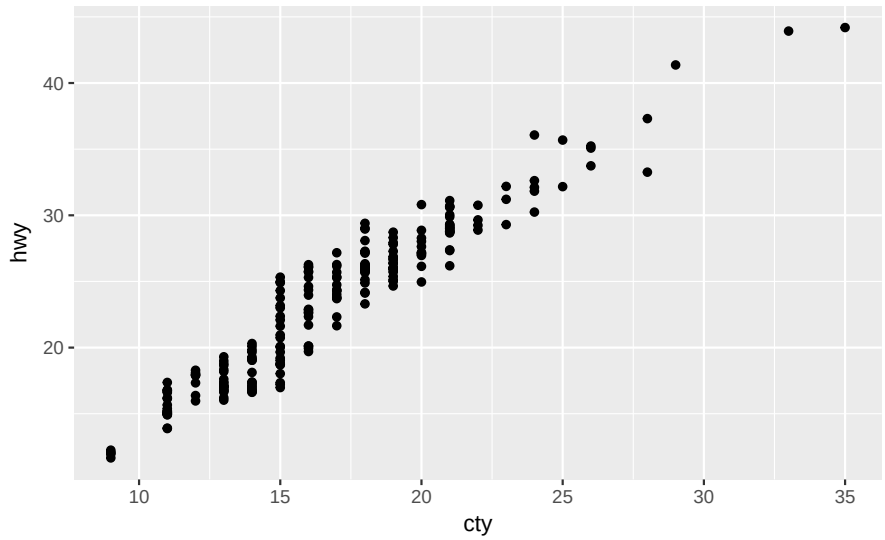
3.8.2 Exercise 2.

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

From the position_jitter documentation, there are two arguments to jitter: `width` and `height`, which control the amount of vertical and horizontal jitter.

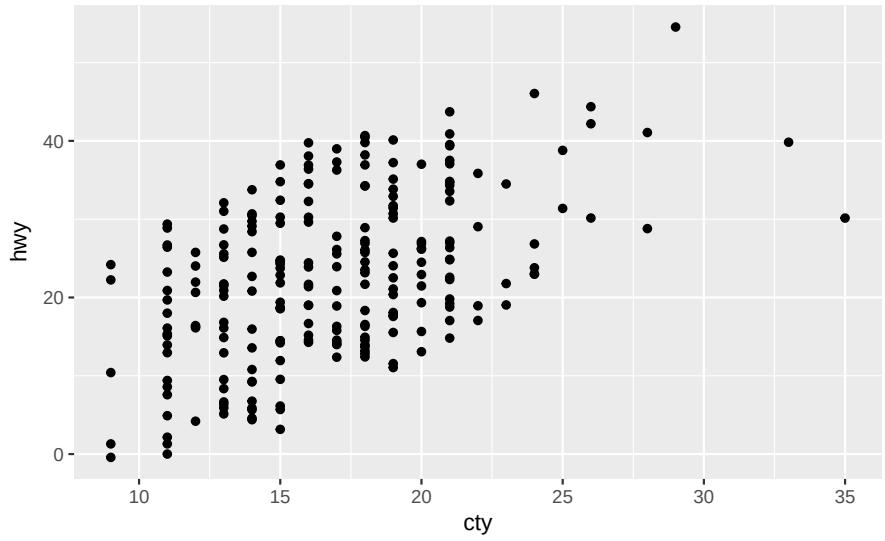
No horizontal jitter

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point(position = position_jitter(width = 0))
```



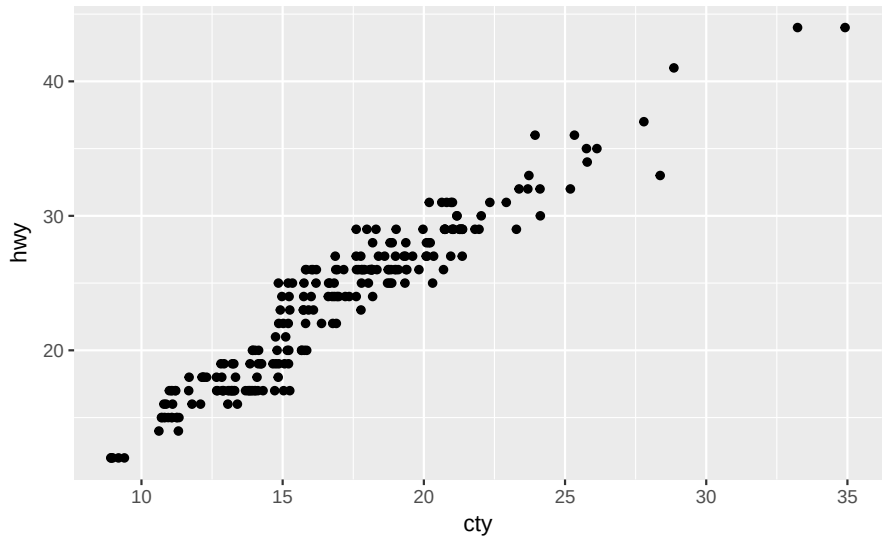
Way too much vertical jitter

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point(position = position_jitter(width = 0, height = 15))
```



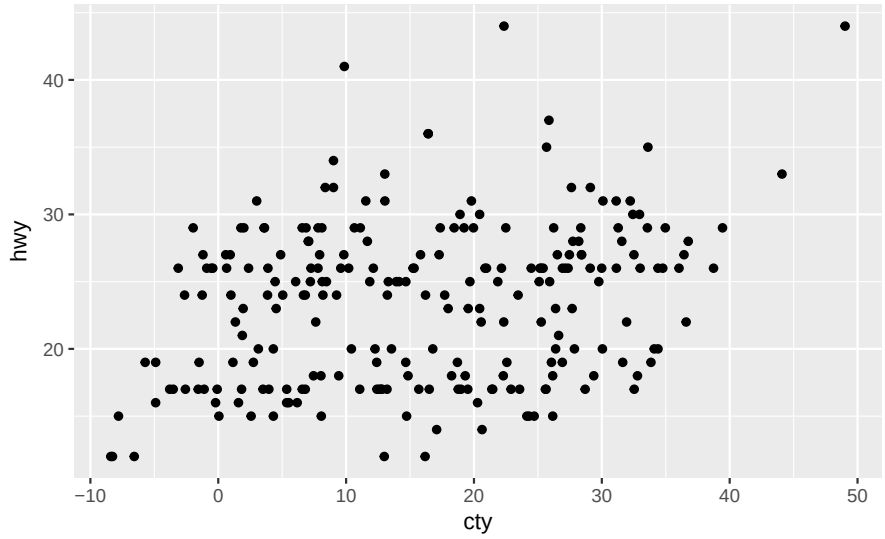
Only horizontal jitter:

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point(position = position_jitter(height = 0))
```



Way too much horizontal jitter:

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point(position = position_jitter(height = 0, width = 20))
```



3.8.3 Exercise 3.

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

TODO

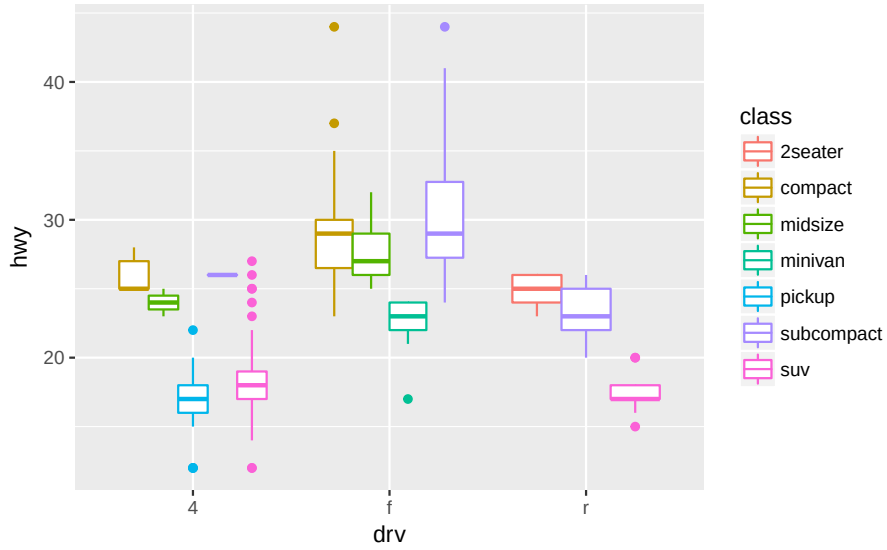
3.8.4 Exercise 4.

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

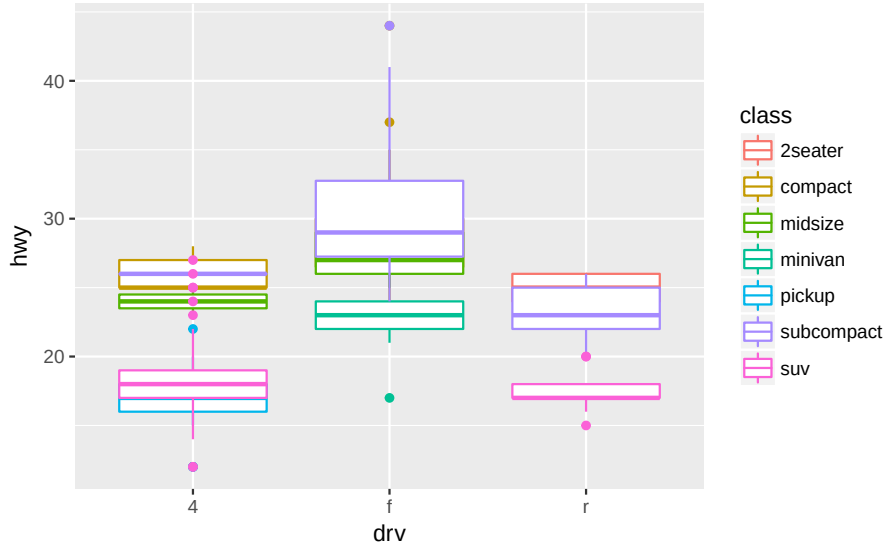
The default position for `geom_boxplot` is `position_dodge` (see its docs).

When we add `color = class` to the box plot, the different classes within `drv` are placed side by side, i.e. dodged. If it was `position_identity`, they would be overlapping.

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




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



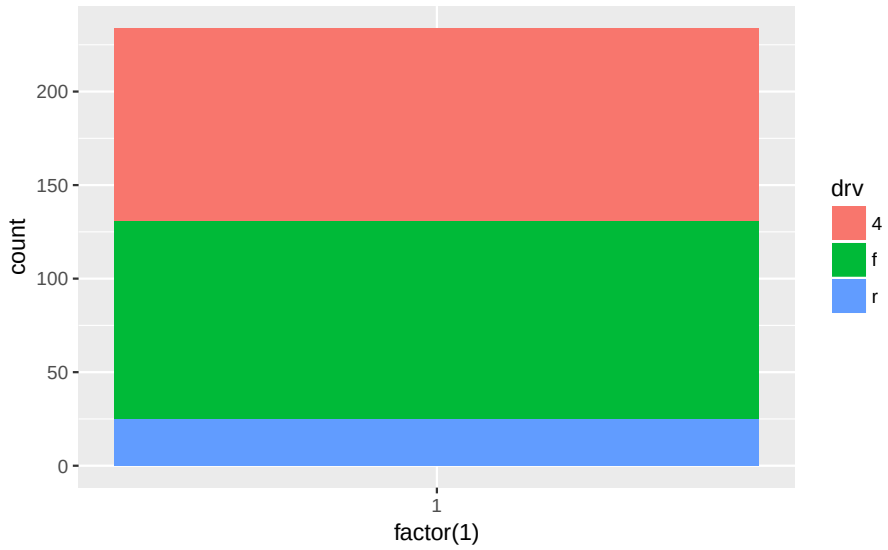
3.9 Coordinate Systems

3.9.1 Exercise 1.

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

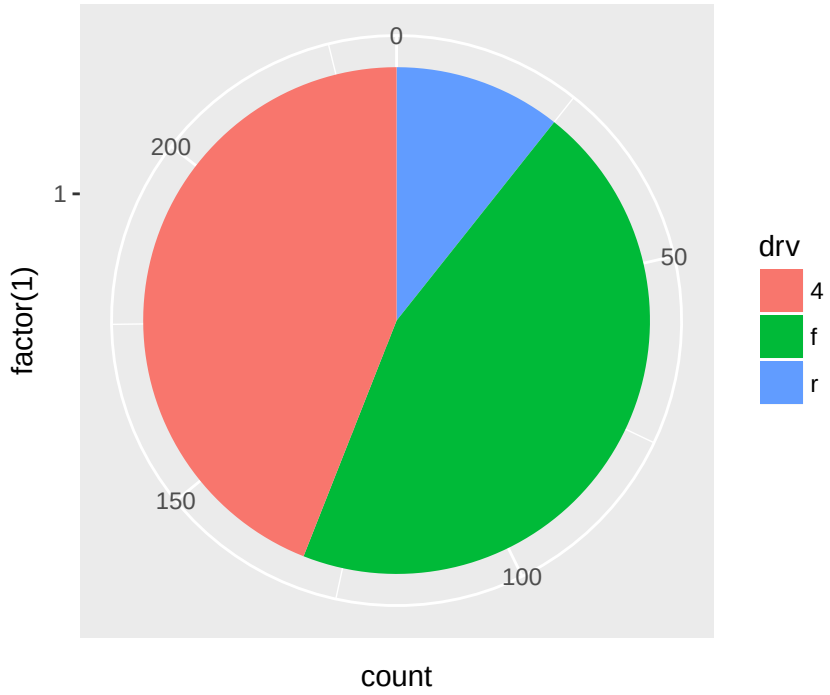
This is a stacked bar chart with a single category

```
ggplot(mpg, aes(x = factor(1), fill = drv)) +
  geom_bar()
```



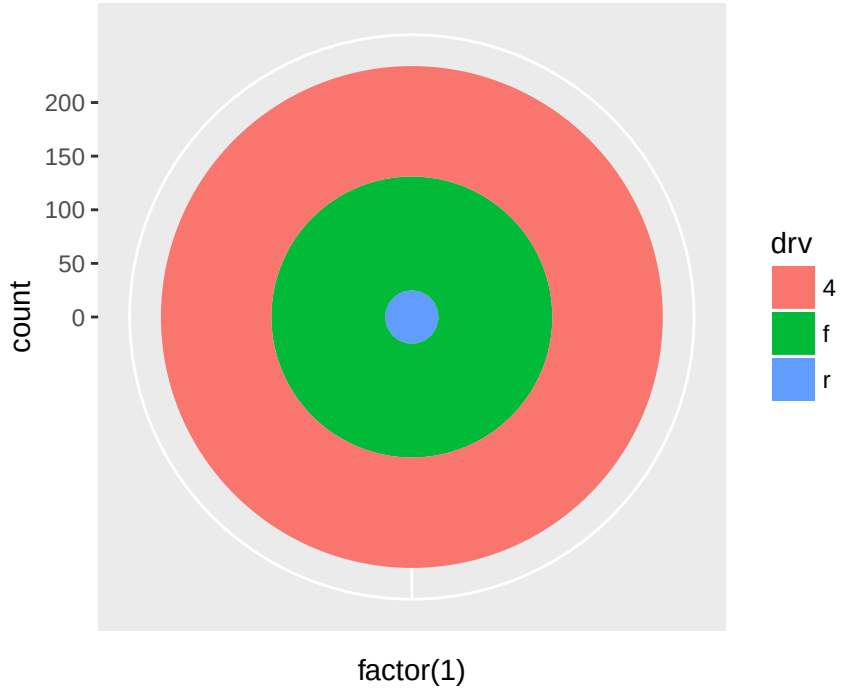
See the documentation for coord_polar for an example of making a pie chart. In particular, `theta = "y"`, meaning that the angle of the chart is the `y` variable has to be specified.

```
ggplot(mpg, aes(x = factor(1), fill = drv)) +
  geom_bar(width = 1) +
  coord_polar(theta = "y")
```



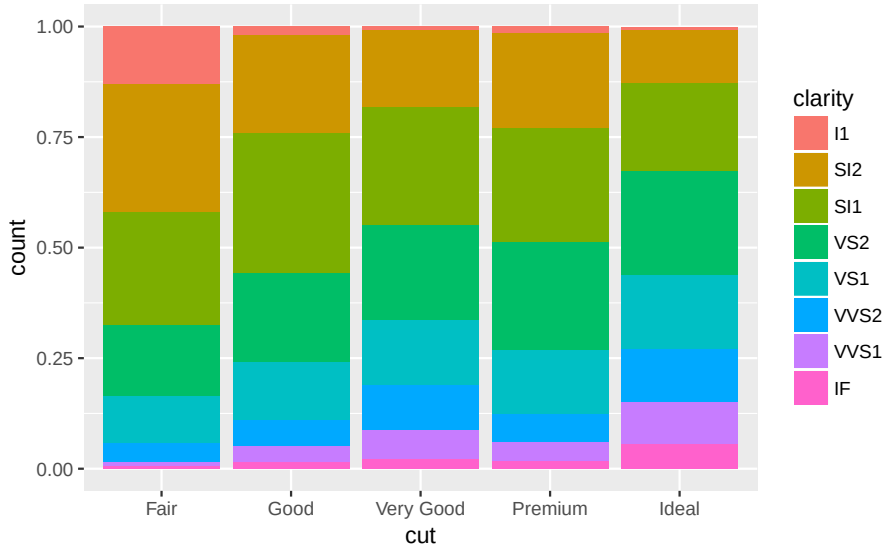
If `theta = "y"` is not specified, then you get a bull's-eye chart

```
ggplot(mpg, aes(x = factor(1), fill = drv)) +
  geom_bar(width = 1) +
  coord_polar()
```



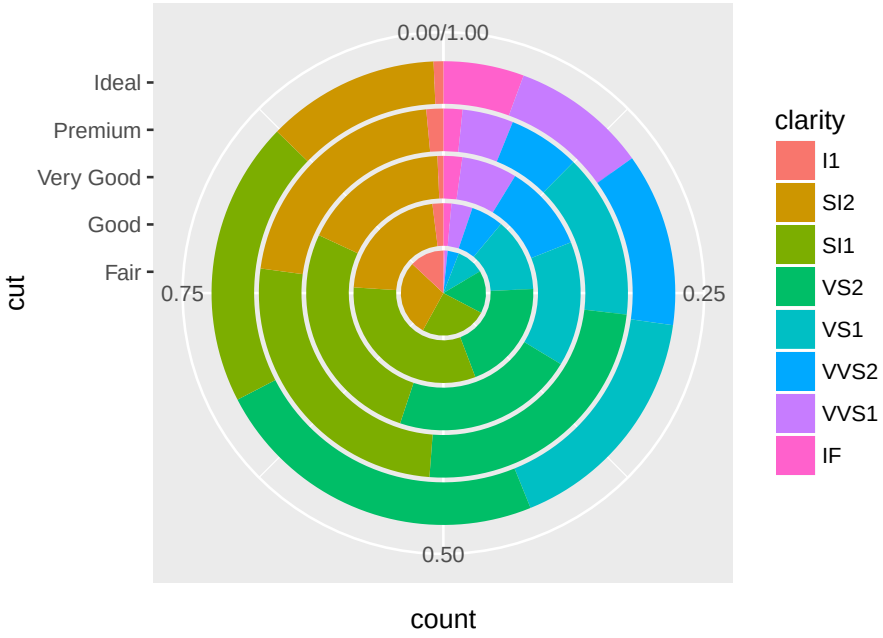
If you had a multiple stacked bar chart, like,

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



you end up with a multi-doughnut chart

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

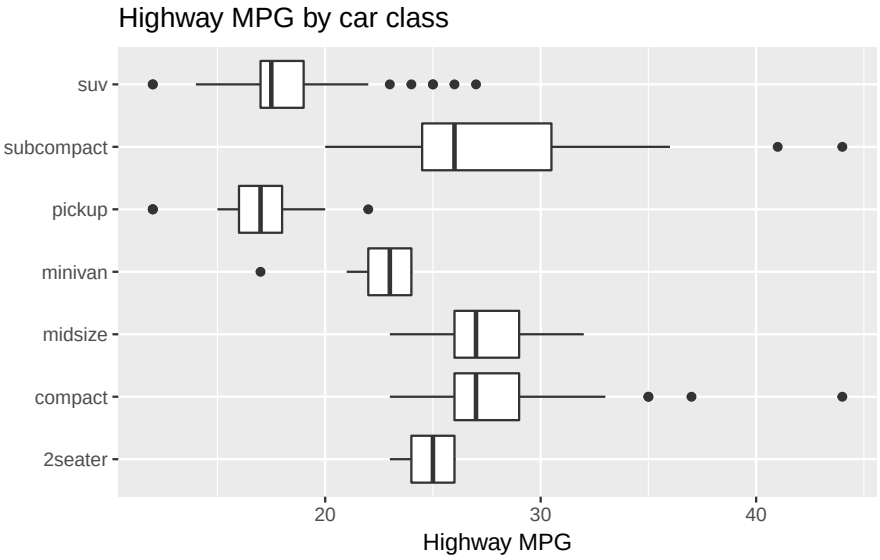


3.9.2 Exercise 2.

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

The `labs` function adds labels for different scales and the title of the plot.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip() +
  labs(y = "Highway MPG", x = "", title = "Highway MPG by car class")
```



3.9.3 Exercise 3.

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

See the docs:

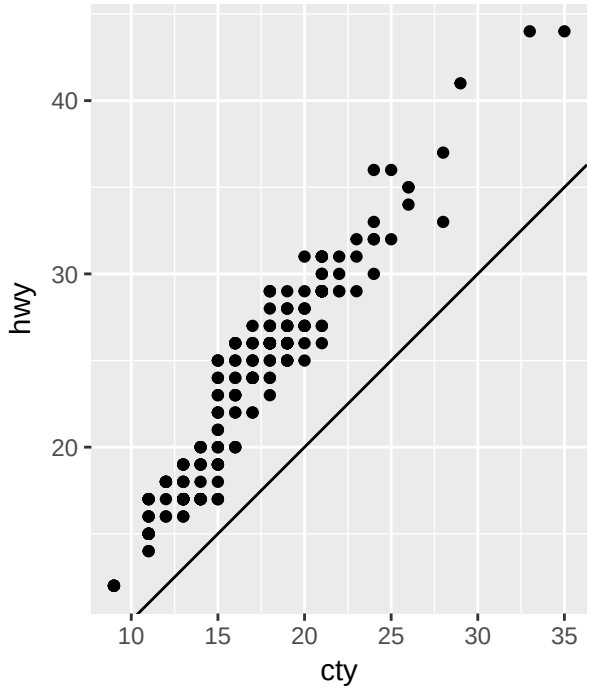
- `coord_map` uses a 2D projection: by default the Mercator project of the sphere to the plot. But this requires transforming all geoms.
- `coord_quickmap` uses a approximate, but faster, map projection using the lat/long ratio as an approximation. This is "quick" because the shapes don't need to be transformed.

3.9.4 Exercise 4.

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

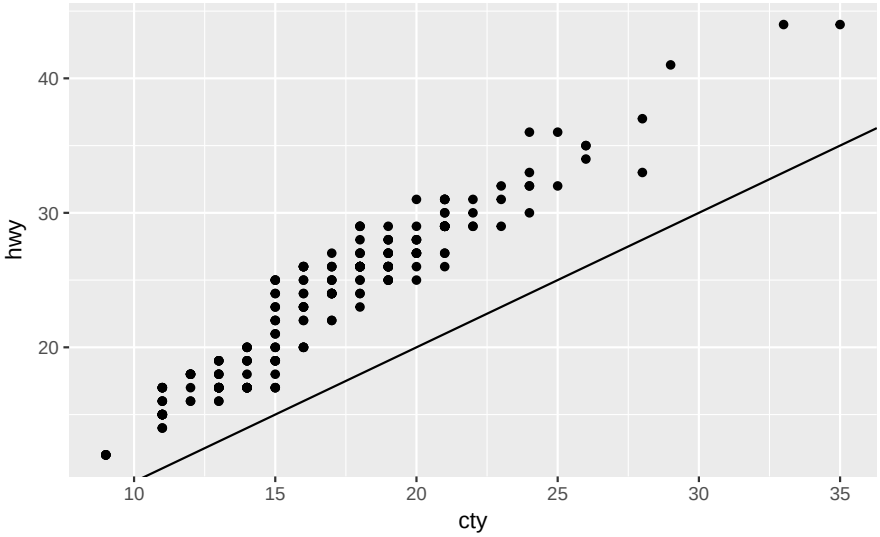
The coordinates `coord_fixed` ensures that the `abline` is at a 45 degree angle, which makes it easy to compare the highway and city mileage against what it would be if they were exactly the same.

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



If we didn't include geom_point, then the line is no longer at 45 degrees:

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



3.10 The Layered Grammar of Graphics

No exercises

Chapter 4

Workflow Basics

4.1 Coding basics

No exercises

4.2 What's in a name?

No exercises

4.3 Calling functions

No exercises

4.4 Practice

4.4.1 Exercise 1

Why does this code not work?

```
my_variable <- 10
my_varıable
#> Error in eval(expr, envir, enclos): object 'my_varıable' not found
```

The variable being printed is `my_varıable`, not `my_variable`: the seventh character is "ı" ("LATIN SMALL LETTER DOTLESS I"), not "i".

While it wouldn't have helped much in this case, the importance of distinguishing characters in code is reasons why fonts which clearly distinguish similar characters are preferred in programming: especially important are distinguishing between zero (0), Latin small letter O (o), and Latin capital letter O (O); and the numeral one (1), Latin small letter I (i), Latin capital letter I (i), and Latin small letter L (l). In these fonts, zero and the Latin letter O are often distinguished by using a glyph for zero that uses either a dot in the interior or a slash through it.

Also note that the error messages of the form "object '…' not found", mean just what they say, the object can't be found by R. This is usually because you either (1) forgot to define the function (or had an error

that prevented it from being defined earlier), (2) didn't load a package with the object, or (3) made a typo in the object's name (either when using it or when you originally defined it).

4.4.2 Exercise 2

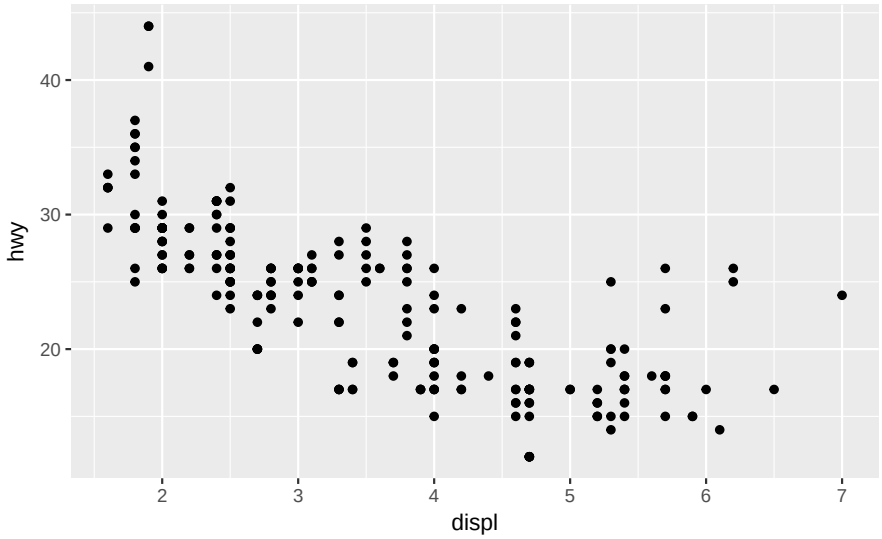
Tweak each of the following R commands so that they run correctly:

```
library(tidyverse)
#> -- Attaching packages --------------------------------------- tidyverse 1.2.1 --
#> ✓ ggplot2 2.2.1          ✓ purrr   0.2.4
#> ✓ tibble  1.4.2          ✓ dplyr   0.7.4.9000
#> ✓ tidyr   0.8.0          ✓ stringr 1.2.0
#> ✓ readr   1.1.1          ✓ forcats 0.2.0
#> -- Conflicts ------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()

ggplot(dota = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
#> Error in structure(list(data = data, layers = list(), scales = scales_list(), : argument "data" is m
```

The error message is `argument "data" is missing, with no default`.

It looks like a typo, `dota` instead of `data`.

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




```
fliter(mpg, cyl = 8)
#> Error in fliter(mpg, cyl = 8): could not find function "fliter"
```

R could not find the function `fliter` because we made a typo: `fliter` instead of `filter`.

```
filter(mpg, cyl = 8)
#> Error: `cyl` (`cyl = 8`) must not be named, do you need `==`?
```

We aren't done yet. But the error message gives a suggestion. Let's follow it.

```
filter(mpg, cyl == 8)
#> # A tibble: 70 x 11
```

```
#>   manufacturer model displ  year   cyl trans drv     cty   hwy fl    class
#>   <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
#> 1 audi         a6 q~  4.20  2008     8 auto~ 4        16    23 p     mids~
#> 2 chevrolet    c150~  5.30  2008     8 auto~ r        14    20 r     suv
#> 3 chevrolet    c150~  5.30  2008     8 auto~ r        11    15 e     suv
#> 4 chevrolet    c150~  5.30  2008     8 auto~ r        14    20 r     suv
#> 5 chevrolet    c150~  5.70  1999     8 auto~ r        13    17 r     suv
#> 6 chevrolet    c150~  6.00  2008     8 auto~ r        12    17 r     suv
#> # ... with 64 more rows
```

```
filter(diamond, carat > 3)
#> Error in filter(diamond, carat > 3): object 'diamond' not found
```

R says it can't find the object `diamond`. This is a typo; the data frame is named `diamonds`.

```
filter(diamonds, carat > 3)
#> # A tibble: 32 x 10
#>   carat cut     color clarity depth table price     x     y     z
#>   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 1  3.01 Premium I     I1       62.7  58.0  8040  9.10  8.97  5.67
#> 2  3.11 Fair    J     I1       65.9  57.0  9823  9.15  9.02  5.98
#> 3  3.01 Premium F     I1       62.2  56.0  9925  9.24  9.13  5.73
#> 4  3.05 Premium E     I1       60.9  58.0 10453  9.26  9.25  5.66
#> 5  3.02 Fair    I     I1       65.2  56.0 10577  9.11  9.02  5.91
#> 6  3.01 Fair    H     I1       56.1  62.0 10761  9.54  9.38  5.31
#> # ... with 26 more rows
```

How did I know? I started typing in `diamond` and RStudio completed it to `diamonds`. Since `diamonds` includes the variable `carat` and the code works, that appears to have been the problem.

4.4.3 Exercise 3

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

This gives a menu with keyboard shortcuts. This can be found in the menu under `Tools -> Keyboard Shortcuts Help`.

Chapter 5

Data Transformation

5.1 Introduction

```
library("nycflights13")
library("tidyverse")
```

5.2 Filter rows with `filter()`

```
glimpse(flights)
#> Observations: 336,776
#> Variables: 19
#> $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013,...
#> $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 55...
#> $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
#> $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2...
#> $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 7...
#> $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 7...
#> $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -...
#> $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV",...
#> $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79...
#> $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN...
#> $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR"...
#> $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL"...
#> $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138...
#> $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 94...
#> $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5,...
#> $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013...
```

5.2.1 Exercise 1

Find all flights that

1. Had an arrival delay of two or more hours
2. Flew to Houston (IAH or HOU)
3. Were operated by United, American, or Delta
4. Departed in summer (July, August, and September)
5. Arrived more than two hours late, but didn't leave late
6. Were delayed by at least an hour, but made up over 30 minutes in flight
7. Departed between midnight and 6am (inclusive)

Had an arrival delay of two or more hours Since delay is in minutes, we are looking for flights where `arr_delay > 120`:

```
flights %>%
  filter(arr_delay > 120)
#> # A tibble: 10,034 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      811            630       101     1047
#> 2  2013     1     1      848           1835       853     1001
#> 3  2013     1     1      957            733       144     1056
#> 4  2013     1     1     1114            900       134     1447
#> 5  2013     1     1     1505           1310       115     1638
#> 6  2013     1     1     1525           1340       105     1831
#> # ... with 1.003e+04 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Flew to Houston (IAH or HOU):

```
flights %>%
  filter(dest %in% c("IAH", "HOU"))
#> # A tibble: 9,313 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      623            627     -4.00      933
#> 4  2013     1     1      728            732     -4.00     1041
#> 5  2013     1     1      739            739      0        1104
#> 6  2013     1     1      908            908      0        1228
#> # ... with 9,307 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

Were operated by United, American, or Delta The variable `carrier` has the airline: but it is in two-digit carrier codes. However, we can look it up in the `airlines` dataset.

```
airlines
#> # A tibble: 16 x 2
#>   carrier name
#>   <chr>   <chr>
#> 1 9E      Endeavor Air Inc.
#> 2 AA      American Airlines Inc.
#> 3 AS      Alaska Airlines Inc.
#> 4 B6      JetBlue Airways
#> 5 DL      Delta Air Lines Inc.
```

```
#> 6 EV      ExpressJet Airlines Inc.
#> # ... with 10 more rows
```

Since there are only 16 rows, its not even worth filtering. Delta is `DL`, American is `AA`, and United is `UA`:

```
filter(flights, carrier %in% c("AA", "DL", "UA"))
#> # A tibble: 139,504 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      542            540      2.00      923
#> 4  2013     1     1      554            600     -6.00      812
#> 5  2013     1     1      554            558     -4.00      740
#> 6  2013     1     1      558            600     -2.00      753
#> # ... with 1.395e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Departed in summer (July, August, and September) The variable `month` has the month, and it is numeric.

```
filter(flights, between(month, 7, 9))
#> # A tibble: 86,326 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     7     1        1           2029    212         236
#> 2  2013     7     1        2           2359      3.00      344
#> 3  2013     7     1       29           2245    104         151
#> 4  2013     7     1       43           2130    193         322
#> 5  2013     7     1       44           2150    174         300
#> 6  2013     7     1       46           2051    235         304
#> # ... with 8.632e+04 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

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

```
filter(flights, !is.na(dep_delay), dep_delay <= 0, arr_delay > 120)
#> # A tibble: 29 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1    27     1419           1420     -1.00     1754
#> 2  2013    10     7     1350           1350      0        1736
#> 3  2013    10     7     1357           1359     -2.00     1858
#> 4  2013    10    16      657            700     -3.00     1258
#> 5  2013    11     1      658            700     -2.00     1329
#> 6  2013     3    18     1844           1847     -3.00       39
#> # ... with 23 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

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

```
filter(flights, !is.na(dep_delay), dep_delay >= 60, dep_delay-arr_delay > 30)
#> # A tibble: 1,844 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1     2205           1720     285         46
#> 2  2013     1     1     2326           2130     116        131
#> 3  2013     1     3     1503           1221     162       1803
#> 4  2013     1     3     1839           1700      99.0     2056
#> 5  2013     1     3     1850           1745      65.0     2148
#> 6  2013     1     3     1941           1759     102       2246
#> # ... with 1,838 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

Departed between midnight and 6am (inclusive).

```
filter(flights, dep_time <=600 | dep_time == 2400)
#> # A tibble: 9,373 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      542            540      2.00      923
#> 4  2013     1     1      544            545     -1.00     1004
#> 5  2013     1     1      554            600     -6.00      812
#> 6  2013     1     1      554            558     -4.00      740
#> # ... with 9,367 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

or using `between` (see next question)

```
filter(flights, between(dep_time, 0, 600))
#> # A tibble: 9,344 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      542            540      2.00      923
#> 4  2013     1     1      544            545     -1.00     1004
#> 5  2013     1     1      554            600     -6.00      812
#> 6  2013     1     1      554            558     -4.00      740
#> # ... with 9,338 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

5.2.2 Exercise 2

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

`between(x, left, right)` is equivalent to `x >= left & x <= right`. I already used it in 1.4.

5.2.3 Exercise 3

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

```
filter(flights, is.na(dep_time))
#> # A tibble: 8,255 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1       NA           1630        NA       NA
#> 2  2013     1     1       NA           1935        NA       NA
#> 3  2013     1     1       NA           1500        NA       NA
#> 4  2013     1     1       NA            600        NA       NA
#> 5  2013     1     2       NA           1540        NA       NA
#> 6  2013     1     2       NA           1620        NA       NA
#> # ... with 8,249 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

Since `arr_time` is also missing, these are canceled flights.

5.2.4 Exercise 4

Why is `NA ^ 0` not missing? Why is `NA | TRUE` not missing? Why is `FALSE & NA` not missing? Can you figure out the general rule? (`NA * 0` is a tricky counterexample!)

`NA ^ 0 == 1` since for all numeric values $x^0 = 1$.

```
NA ^ 0
#> [1] 1
```

`NA | TRUE` is `TRUE` because the it doesn't matter whether the missing value is `TRUE` or `FALSE`, `x \lor T = T` for all values of `x`.

```
NA | TRUE
#> [1] TRUE
```

Likewise, anything and `FALSE` is always `FALSE`.

```
NA & FALSE
#> [1] FALSE
```

Because the value of the missing element matters in `NA | FALSE` and `NA & TRUE`, these are missing:

```
NA | FALSE
#> [1] NA
NA & TRUE
#> [1] NA
```

Wut?? Since `x * 0 = 0` for all x (except `Inf`) we might expect `NA * 0 = 0`, but that's not the case.

```
NA * 0
#> [1] NA
```

The reason that `NA * 0` is not equal to `0` is that `x * 0 = NaN` is undefined when `x = Inf` or `x = -Inf`.

```
Inf * 0
#> [1] NaN
```

```
-Inf * 0
#> [1] NaN
```

5.3 Arrange rows with `arrange()`

5.3.1 Exercise 1

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

This sorts by increasing `dep_time`, but with all missing values put first.

```
arrange(flights, desc(is.na(dep_time)), dep_time)
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1       NA           1630        NA       NA
#> 2  2013     1     1       NA           1935        NA       NA
#> 3  2013     1     1       NA           1500        NA       NA
#> 4  2013     1     1       NA            600        NA       NA
#> 5  2013     1     2       NA           1540        NA       NA
#> 6  2013     1     2       NA           1620        NA       NA
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

5.3.2 Exercise 2

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

The most delayed flights are found by sorting by `dep_delay` in descending order.

```
arrange(flights, desc(dep_delay))
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     9      641            900      1301     1242
#> 2  2013     6    15     1432           1935      1137     1607
#> 3  2013     1    10     1121           1635      1126     1239
#> 4  2013     9    20     1139           1845      1014     1457
#> 5  2013     7    22      845           1600      1005     1044
#> 6  2013     4    10     1100           1900       960     1342
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

If we sort `dep_delay` in ascending order, we get those that left earliest. There was a flight that left 43 minutes early.

```
arrange(flights, dep_delay)
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
```

```
#> 1  2013    12     7     2040           2123     -43.0       40
#> 2  2013     2     3     2022           2055     -33.0     2240
#> 3  2013    11    10     1408           1440     -32.0     1549
#> 4  2013     1    11     1900           1930     -30.0     2233
#> 5  2013     1    29     1703           1730     -27.0     1947
#> 6  2013     8     9      729            755     -26.0     1002
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

5.3.3 Exercise 3

Sort flights to find the fastest flights.

I assume that by by "fastest flights" it means the flights with the minimum air time. So I sort by `air_time`. The fastest flights. The fastest flights area couple of flights between EWR and BDL with an air time of 20 minutes.

```
arrange(flights, air_time)
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1    16     1355           1315      40.0     1442
#> 2  2013     4    13      537            527      10.0      622
#> 3  2013    12     6      922            851      31.0     1021
#> 4  2013     2     3     2153           2129      24.0     2247
#> 5  2013     2     5     1303           1315     -12.0     1342
#> 6  2013     2    12     2123           2130     - 7.00    2211
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

5.3.4 Exercise 4

Which flights traveled the longest? Which traveled the shortest?

I'll assume hat traveled the longest or shortest refers to distance, rather than air-time.

The longest flights are the Hawaii Air (HA 51) between JFK and HNL (Honolulu) at 4,983 miles.

```
arrange(flights, desc(distance))
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      857            900     - 3.00    1516
#> 2  2013     1     2      909            900       9.00    1525
#> 3  2013     1     3      914            900      14.0     1504
#> 4  2013     1     4      900            900       0       1516
#> 5  2013     1     5      858            900     - 2.00    1519
#> 6  2013     1     6     1019            900      79.0     1558
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
```

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

Apart from an EWR to LGA flight that was canceled, the shortest flights are the Envoy Air Flights between EWR and PHL at 80 miles.

```
arrange(flights, distance)
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     7    27       NA            106        NA       NA
#> 2  2013     1     3     2127           2129    -  2.00     2222
#> 3  2013     1     4     1240           1200     40.0      1333
#> 4  2013     1     4     1829           1615    134        1937
#> 5  2013     1     4     2128           2129    -  1.00     2218
#> 6  2013     1     5     1155           1200    -  5.00     1241
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

5.4 Select columns with `select()`

5.4.1 Exercise 1

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

A few ways include:

```
select(flights, dep_time, dep_delay, arr_time, arr_delay)
#> # A tibble: 336,776 x 4
#>   dep_time dep_delay arr_time arr_delay
#>      <int>     <dbl>    <int>     <dbl>
#> 1      517      2.00      830      11.0
#> 2      533      4.00      850      20.0
#> 3      542      2.00      923      33.0
#> 4      544     -1.00     1004     -18.0
#> 5      554     -6.00      812     -25.0
#> 6      554     -4.00      740      12.0
#> # ... with 3.368e+05 more rows
select(flights, starts_with("dep_"), starts_with("arr_"))
#> # A tibble: 336,776 x 4
#>   dep_time dep_delay arr_time arr_delay
#>      <int>     <dbl>    <int>     <dbl>
#> 1      517      2.00      830      11.0
#> 2      533      4.00      850      20.0
#> 3      542      2.00      923      33.0
#> 4      544     -1.00     1004     -18.0
#> 5      554     -6.00      812     -25.0
#> 6      554     -4.00      740      12.0
#> # ... with 3.368e+05 more rows
select(flights, matches("^(dep|arr)_(time|delay)$"))
#> # A tibble: 336,776 x 4
```

```
#>   dep_time dep_delay arr_time arr_delay
#>      <int>     <dbl>    <int>     <dbl>
#> 1      517      2.00      830      11.0
#> 2      533      4.00      850      20.0
#> 3      542      2.00      923      33.0
#> 4      544     -1.00     1004     -18.0
#> 5      554     -6.00      812     -25.0
#> 6      554     -4.00      740      12.0
#> # ... with 3.368e+05 more rows
```

using `ends_with()` doesn't work well since it would return both `sched_arr_time` and `sched_dep_time`.

5.4.2 Exercise 2

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

It ignores the duplicates, and that variable is only included once. No error, warning, or message is emitted.

```
select(flights, year, month, day, year, year)
#> # A tibble: 336,776 x 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # ... with 3.368e+05 more rows
```

5.4.3 Exercise 3

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

The `one_of` vector allows you to select variables with a character vector rather than as unquoted variable names. It's useful because then you can easily pass vectors to `select()`.

```
vars <- c("year", "month", "day", "dep_delay", "arr_delay")
select(flights, one_of(vars))
#> # A tibble: 336,776 x 5
#>    year month   day dep_delay arr_delay
#>   <int> <int> <int>     <dbl>     <dbl>
#> 1  2013     1     1      2.00      11.0
#> 2  2013     1     1      4.00      20.0
#> 3  2013     1     1      2.00      33.0
#> 4  2013     1     1     -1.00     -18.0
#> 5  2013     1     1     -6.00     -25.0
#> 6  2013     1     1     -4.00      12.0
#> # ... with 3.368e+05 more rows
```

5.4.4 Exercise 4

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

```
select(flights, contains("TIME"))
#> # A tibble: 336,776 x 6
#>   dep_time sched_dep_time arr_time sched_arr_time air_time
#>      <int>          <int>    <int>          <int>    <dbl>
#> 1      517            515      830            819      227
#> 2      533            529      850            830      227
#> 3      542            540      923            850      160
#> 4      544            545     1004           1022      183
#> 5      554            600      812            837      116
#> 6      554            558      740            728      150
#> # ... with 3.368e+05 more rows, and 1 more variable: time_hour <dttm>
```

The default behavior for contains is to ignore case. Yes, it surprises me. Upon reflection, I realized that this is likely the default behavior because `dplyr` is designed to deal with a variety of data backends, and some database engines don't differentiate case.

To change the behavior add the argument `ignore.case = FALSE`. Now no variables are selected.

```
select(flights, contains("TIME", ignore.case = FALSE))
#> # A tibble: 336,776 x 0
```

5.5 Add new variables with `mutate()`

5.5.1 Exercise 1

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

To get the departure times in the number of minutes, (integer) divide `dep_time` by 100 to get the hours since midnight and multiply by 60 and add the remainder of `dep_time` divided by 100.

```
mutate(flights,
       dep_time_mins = dep_time %/% 100 * 60 + dep_time %% 100,
       sched_dep_time_mins = sched_dep_time %/% 100 * 60 + sched_dep_time %% 100) %>%
  select(dep_time, dep_time_mins, sched_dep_time, sched_dep_time_mins)
#> # A tibble: 336,776 x 4
#>   dep_time dep_time_mins sched_dep_time sched_dep_time_mins
#>      <int>         <dbl>          <int>               <dbl>
#> 1      517           317            515                 315
#> 2      533           333            529                 329
#> 3      542           342            540                 340
#> 4      544           344            545                 345
#> 5      554           354            600                 360
#> 6      554           354            558                 358
#> # ... with 3.368e+05 more rows
```

This would be more cleanly done by first defining a function and reusing that:

```
time2mins <- function(x) {
  x %/% 100 * 60 + x %% 100
}
mutate(flights,
       dep_time_mins = time2mins(dep_time),
       sched_dep_time_mins = time2mins(sched_dep_time)) %>%
```

```
  select(dep_time, dep_time_mins, sched_dep_time, sched_dep_time_mins)
#> # A tibble: 336,776 x 4
#>   dep_time dep_time_mins sched_dep_time sched_dep_time_mins
#>      <int>         <dbl>          <int>               <dbl>
#> 1      517           317            515                 315
#> 2      533           333            529                 329
#> 3      542           342            540                 340
#> 4      544           344            545                 345
#> 5      554           354            600                 360
#> 6      554           354            558                 358
#> # ... with 3.368e+05 more rows
```

5.5.2 Exercise 2

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

Since `arr_time` and `dep_time` may be in different time zones, the `air_time` doesn't equal the difference. We would need to account for time-zones in these calculations.

```
mutate(flights,
       air_time2 = arr_time - dep_time,
       air_time_diff = air_time2 - air_time) %>%
  filter(air_time_diff != 0) %>%
  select(air_time, air_time2, dep_time, arr_time, dest)
#> # A tibble: 326,128 x 5
#>   air_time air_time2 dep_time arr_time dest
#>      <dbl>     <int>    <int>    <int> <chr>
#> 1      227       313      517      830 IAH
#> 2      227       317      533      850 IAH
#> 3      160       381      542      923 MIA
#> 4      183       460      544     1004 BQN
#> 5      116       258      554      812 ATL
#> 6      150       186      554      740 ORD
#> # ... with 3.261e+05 more rows
```

5.5.3 Exercise 3

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

I'd expect `dep_time`, `sched_dep_time`, and `dep_delay` to be related so that `dep_time - sched_dep_time = dep_delay`.

```
mutate(flights,
       dep_delay2 = dep_time - sched_dep_time) %>%
  filter(dep_delay2 != dep_delay) %>%
  select(dep_time, sched_dep_time, dep_delay, dep_delay2)
#> # A tibble: 99,777 x 4
#>   dep_time sched_dep_time dep_delay dep_delay2
#>      <int>          <int>     <dbl>      <int>
#> 1      554            600     -6.00        -46
#> 2      555            600     -5.00        -45
#> 3      557            600     -3.00        -43
```

```
#> 4      557            600     -3.00        -43
#> 5      558            600     -2.00        -42
#> 6      558            600     -2.00        -42
#> # ... with 9.977e+04 more rows
```

Oops, I forgot to convert to minutes. I'll reuse the `time2mins` function I wrote earlier.

```
mutate(flights,
       dep_delay2 = time2mins(dep_time) - time2mins(sched_dep_time)) %>%
  filter(dep_delay2 != dep_delay) %>%
  select(dep_time, sched_dep_time, dep_delay, dep_delay2)
#> # A tibble: 1,207 x 4
#>   dep_time sched_dep_time dep_delay dep_delay2
#>      <int>          <int>     <dbl>      <dbl>
#> 1      848           1835     853        - 587
#> 2       42           2359      43.0      -1397
#> 3      126           2250     156        -1284
#> 4       32           2359      33.0      -1407
#> 5       50           2145     185        -1255
#> 6      235           2359     156        -1284
#> # ... with 1,201 more rows
```

Well, that solved most of the problems, but these two numbers don't match because we aren't accounting for flights where the departure time is the next day from the scheduled departure time.

5.5.4 Exercise 4

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

I'd want to handle ties by taking the minimum of tied values. If three flights are have the same value and are the most delayed, we would say they are tied for first, not tied for third or second.

```
mutate(flights,
       dep_delay_rank = min_rank(-dep_delay)) %>%
  arrange(dep_delay_rank) %>%
  filter(dep_delay_rank <= 10)
#> # A tibble: 10 x 20
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     9      641            900      1301     1242
#> 2  2013     6    15     1432           1935      1137     1607
#> 3  2013     1    10     1121           1635      1126     1239
#> 4  2013     9    20     1139           1845      1014     1457
#> 5  2013     7    22      845           1600      1005     1044
#> 6  2013     4    10     1100           1900       960     1342
#> # ... with 4 more rows, and 13 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, dep_delay_rank <int>
```

5.5.5 Exercise 5

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

It returns `c(1 + 1, 2 + 2, 3 + 3, 1 + 4, 2 + 5, 3 + 6, 1 + 7, 2 + 8, 3 + 9, 1 + 10)`. When adding two vectors recycles the shorter vector's values to get vectors of the same length. We get a warning vector since the shorter vector is not a multiple of the longer one (this often, but not necessarily, means we made an error somewhere).

```
1:3 + 1:10
#> Warning in 1:3 + 1:10: longer object length is not a multiple of shorter
#> object length
#>  [1]  2  4  6  5  7  9  8 10 12 11
```

5.5.6 Exercise 6

What trigonometric functions does R provide?

These are all described in the same help page,

```
help("Trig")
```

Cosine (`cos`), sine (`sin`), tangent (`tan`) are provided:

```
tibble(
  x = seq(-3, 7, by = 1 / 2),
  cosx = cos(pi * x),
  sinx = cos(pi * x),
  tanx = tan(pi * x)
)
#> # A tibble: 21 x 4
#>       x                   cosx                   sinx                tanx
#>   <dbl>                  <dbl>                  <dbl>               <dbl>
#> 1 -3.00  -1.00                    -1.00                             3.67e-16
#> 2 -2.50   0.000000000000000306    0.000000000000000306            -3.27e+15
#> 3 -2.00   1.00                     1.00                             2.45e-16
#> 4 -1.50  -0.000000000000000184   -0.000000000000000184            -5.44e+15
#> 5 -1.00  -1.00                    -1.00                             1.22e-16
#> 6 -0.500  0.000000000000000612    0.000000000000000612            -1.63e+16
#> # ... with 15 more rows
```

The convenience function `cospi(x)` is equivalent to `cos(pi * x)`, with `sinpi` and `tanpi` similarly defined,

```
tibble(
  x = seq(-3, 7, by = 1 / 2),
  cosx = cospi(x),
  sinx = cos(x),
  tanx = tan(x)
)
#> # A tibble: 21 x 4
#>       x  cosx    sinx    tanx
#>   <dbl> <dbl>   <dbl>   <dbl>
#> 1 -3.00  -1.00 -0.990    0.143
#> 2 -2.50   0    -0.801    0.747
#> 3 -2.00   1.00 -0.416    2.19
#> 4 -1.50   0     0.0707 -14.1
#> 5 -1.00  -1.00  0.540   - 1.56
#> 6 -0.500  0     0.878   - 0.546
#> # ... with 15 more rows
```

The inverse function arc-cosine (`acos`), arc-sine (`asin`), and arc-tangent (`atan`) are provided,

```
tibble(
  x = seq(-1, 1, by = 1 / 4),
  acosx = acos(x),
  asinx = asin(x),
  atanx = atan(x)
)
#> # A tibble: 9 x 4
#>        x acosx  asinx  atanx
#>    <dbl> <dbl>  <dbl>  <dbl>
#> 1 -1.00   3.14 -1.57  -0.785
#> 2 -0.750  2.42 -0.848 -0.644
#> 3 -0.500  2.09 -0.524 -0.464
#> 4 -0.250  1.82 -0.253 -0.245
#> 5  0      1.57  0      0
#> 6  0.250  1.32  0.253  0.245
#> # ... with 3 more rows
```

The function `atan2` is the angle between the x-axis and the the vector (0,0) to (`x`, `y`).

```
atan2(c(1, 0, -1, 0), c(0, 1, 0, -1))
#> [1]  1.57  0.00 -1.57  3.14
```

5.6 Grouped summaries with `summarise()`

5.6.1 Exercise 1

> Brainstorm at least 5 different ways to assess the typical delay characteristics of a group of flights. Consider the following scenarios: - A flight is 15 minutes early 50% of the time, and 15 minutes late 50% of the time. - A flight is always 10 minutes late. - A flight is 30 minutes early 50% of the time, and 30 minutes late 50% of the time. - 99% of the time a flight is on time. 1% of the time it's 2 hours late.
>
> Which is more important: arrival delay or departure delay?

Arrival delay is more important. Arriving early is nice, but equally as good as arriving late is bad. Variation is worse than consistency; if I know the plane will always arrive 10 minutes late, then I can plan for it arriving as if the actual arrival time was 10 minutes later than the scheduled arrival time.

So I'd try something that calculates the expected time of the flight, and then aggregates over any delays from that time. I would ignore any early arrival times. A better ranking would also consider cancellations, and need a way to convert them to a delay time (perhaps using the arrival time of the next flight to the same destination).

5.6.2 Exercise 2

> Come up with another approach that will give you the same output as `not_canceled %>% count(dest)` and `not_canceled %>% count(tailnum, wt = distance)` (without using `count()`).

5.6.3 Exercise 3

> Our definition of canceled flights `(is.na(dep_delay) | is.na(arr_delay))` is slightly suboptimal. Why? Which is the most important column?

If a flight doesn't depart, then it won't arrive. A flight can also depart and not arrive if it crashes; I'm not sure how this data would handle flights that are redirected and land at other airports for whatever reason.

The more important column is `arr_delay` so we could just use that.

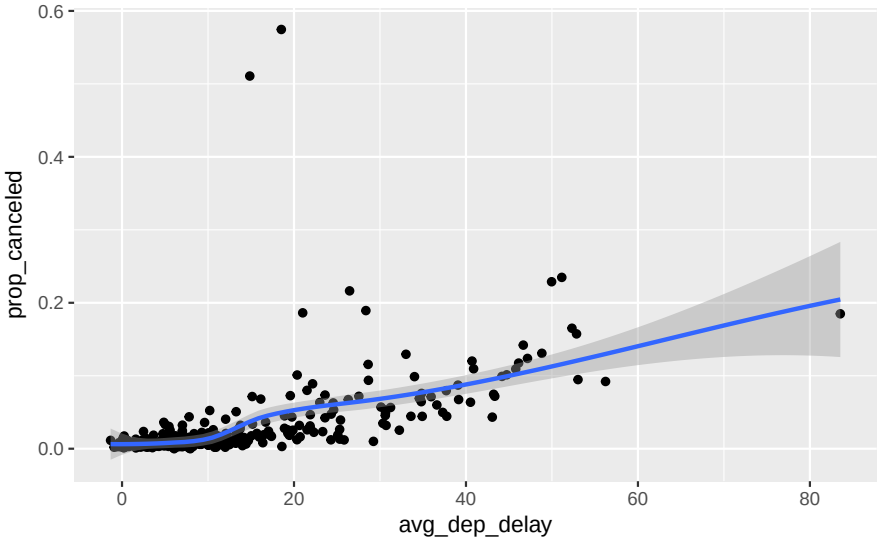
```
filter(flights, !is.na(dep_delay), is.na(arr_delay)) %>%
  select(dep_time, arr_time, sched_arr_time, dep_delay, arr_delay)
#> # A tibble: 1,175 x 5
#>   dep_time arr_time sched_arr_time dep_delay arr_delay
#>      <int>    <int>          <int>     <dbl>     <dbl>
#> 1     1525     1934           1805    - 5.00        NA
#> 2     1528     2002           1647     29.0         NA
#> 3     1740     2158           2020    - 5.00        NA
#> 4     1807     2251           2103     29.0         NA
#> 5     1939       29           2151     59.0         NA
#> 6     1952     2358           2207     22.0         NA
#> # ... with 1,169 more rows
```

Okay, I'm not sure what's going on in this data. `dep_time` can be non-missing and `arr_delay` missing but `arr_time` not missing. They may be combining different flights?

5.6.4 Exercise 4

> Look at the number of canceled flights per day. Is there a pattern? Is the proportion of canceled flights related to the average delay?

```
canceled_delayed <-
  flights %>%
  mutate(canceled = (is.na(arr_delay) | is.na(dep_delay))) %>%
  group_by(year, month, day) %>%
  summarise(prop_canceled = mean(canceled),
            avg_dep_delay = mean(dep_delay, na.rm = TRUE))

ggplot(canceled_delayed, aes(x = avg_dep_delay, prop_canceled)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'loess'
```



5.6.5 Exercise 5

> Which carrier has the worst delays? Challenge: can you disentangle the effects of bad airports vs. bad carriers? Why/why not? (Hint: think about `flights %>% group_by(carrier, dest) %>% summarise(n())`)

```
flights %>%
  group_by(carrier) %>%
  summarise(arr_delay = mean(arr_delay, na.rm = TRUE)) %>%
  arrange(desc(arr_delay))
#> # A tibble: 16 x 2
#>   carrier arr_delay
#>   <chr>       <dbl>
#> 1 F9           21.9
#> 2 FL           20.1
#> 3 EV           15.8
#> 4 YV           15.6
#> 5 OO           11.9
#> 6 MQ           10.8
#> # ... with 10 more rows
```

```
filter(airlines, carrier == "F9")
#> # A tibble: 1 x 2
#>   carrier name
#>   <chr>   <chr>
#> 1 F9      Frontier Airlines Inc.
```

Frontier Airlines (FL) has the worst delays.

You can get part of the way to disentangling the effects of airports vs. carriers by comparing each flight's delay to the average delay of destination airport. However, you'd really want to compare it to the average delay of the destination airport, *after* removing other flights from the same airline.

FiveThirtyEight conducted a similar analysis.

5.6.6 Exercise 6

> For each plane, count the number of flights before the first delay of greater than 1 hour.

I think this requires grouped mutate (but I may be wrong):

```
flights %>%
  arrange(tailnum, year, month, day) %>%
  group_by(tailnum) %>%
  mutate(delay_gt1hr = dep_delay > 60) %>%
  mutate(before_delay = cumsum(delay_gt1hr)) %>%
  filter(before_delay < 1) %>%
  count(sort = TRUE)
#> # A tibble: 3,755 x 2
#> # Groups:   tailnum [3,755]
#>   tailnum     n
#>   <chr>   <int>
#> 1 N954UW    206
#> 2 N952UW    163
#> 3 N957UW    142
#> 4 N5FAAA    117
#> 5 N38727     99
```

```
#> 6 N3742C     98
#> # ... with 3,749 more rows
```

5.6.7 Exercise 7

> What does the sort argument to `count()` do. When might you use it?

The sort argument to `count` sorts the results in order of `n`. You could use this anytime you would do `count` followed by `arrange`.

5.7 Grouped mutates (and filters)

5.7.1 Exercise 1

> Refer back to the table of useful mutate and filtering functions. Describe how each operation changes when you combine it with grouping.

They operate within each group rather than over the entire data frame. E.g. `mean` will calculate the mean within each group.

5.7.2 Exercise 2

> Which plane (`tailnum`) has the worst on-time record?

```
flights %>%
  group_by(tailnum) %>%
  summarise(arr_delay = mean(arr_delay, na.rm = TRUE)) %>%
  ungroup() %>%
  filter(rank(desc(arr_delay)) <= 1)
#> # A tibble: 1 x 2
#>   tailnum arr_delay
#>   <chr>       <dbl>
#> 1 N844MH        320
```

5.7.3 Exercise 3

> What time of day should you fly if you want to avoid delays as much as possible?

Let's group by hour. The earlier the better to fly. This is intuitive as delays early in the morning are likely to propagate throughout the day.

```
flights %>%
  group_by(hour) %>%
  summarise(arr_delay = mean(arr_delay, na.rm = TRUE)) %>%
  ungroup() %>%
  arrange(arr_delay)
#> # A tibble: 20 x 2
#>    hour arr_delay
#>   <dbl>     <dbl>
#> 1  7.00    -5.30
#> 2  5.00    -4.80
#> 3  6.00    -3.38
```

```
#> 4  9.00    -1.45
#> 5  8.00    -1.11
#> 6 10.0      0.954
#> # ... with 14 more rows
```

5.7.4 Exercise 4

For each destination, compute the total minutes of delay. For each, flight, compute the proportion of the total delay for its destination.

```
flights %>%
  filter(!is.na(arr_delay), arr_delay > 0) %>%
  group_by(dest) %>%
  mutate(total_delay = sum(arr_delay),
         prop_delay = arr_delay / sum(arr_delay))
#> # A tibble: 133,004 x 21
#> # Groups:   dest [103]
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      542            540      2.00      923
#> 4  2013     1     1      554            558     -4.00      740
#> 5  2013     1     1      555            600     -5.00      913
#> 6  2013     1     1      558            600     -2.00      753
#> # ... with 1.33e+05 more rows, and 14 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   total_delay <dbl>, prop_delay <dbl>
```

Alternatively, consider the delay as relative to the *minimum* delay for any flight to that destination. Now all non-canceled flights have a proportion.

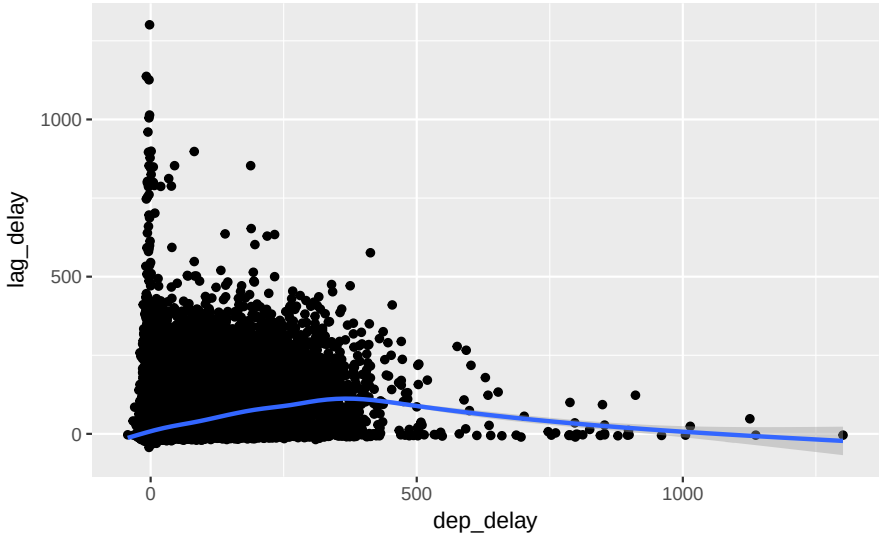
```
flights %>%
  filter(!is.na(arr_delay), arr_delay > 0) %>%
  group_by(dest) %>%
  mutate(total_delay = sum(arr_delay - min(arr_delay)),
         prop_delay = arr_delay / sum(arr_delay))
#> # A tibble: 133,004 x 21
#> # Groups:   dest [103]
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      542            540      2.00      923
#> 4  2013     1     1      554            558     -4.00      740
#> 5  2013     1     1      555            600     -5.00      913
#> 6  2013     1     1      558            600     -2.00      753
#> # ... with 1.33e+05 more rows, and 14 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   total_delay <dbl>, prop_delay <dbl>
```

5.7.5 Exercise 5

> Delays are typically temporally correlated: even once the problem that caused the initial delay has been resolved, later flights are delayed to allow earlier flights to leave. Using `lag()` explore how the delay of a flight is related to the delay of the immediately preceding flight.

We want to group by day to avoid taking the lag from the previous day. Also, I want to use departure delay, since this mechanism is relevant for departures. Also, I remove missing values both before and after calculating the lag delay. However, it would be interesting to ask the probability or average delay after a cancellation.

```
flights %>%
  group_by(year, month, day) %>%
  filter(!is.na(dep_delay)) %>%
  mutate(lag_delay = lag(dep_delay)) %>%
  filter(!is.na(lag_delay)) %>%
  ggplot(aes(x = dep_delay, y = lag_delay)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'gam'
```



5.7.6 Exercise 6

> Look at each destination. Can you find flights that are suspiciously fast? (i.e. flights that represent a potential data entry error). Compute the air time a flight relative to the shortest flight to that destination. Which flights were most delayed in the air?

The shorter BOS and PHL flights that are 20 minutes for 30+ minutes flights seem plausible - though maybe entries of +/- a few minutes can easily create large changes. I assume that departure time has a standardized definition, but I'm not sure; if there is some discretion, that could create errors that are small in absolute time, but large in relative time for small flights. The ATL, GSP, and BNA flights look suspicious as they are almost half the time of longer flights.

```
flights %>%
  filter(!is.na(air_time)) %>%
  group_by(dest) %>%
  mutate(med_time = median(air_time),
         fast = (air_time - med_time) / med_time) %>%
```

```
  arrange(fast) %>%
  select(air_time, med_time, fast, dep_time, sched_dep_time, arr_time, sched_arr_time) %>%
  head(15)
#> Adding missing grouping variables: `dest`
#> # A tibble: 15 x 8
#> # Groups:   dest [9]
#>    dest  air_time med_time   fast dep_time sched_dep_time arr_time
#>    <chr>    <dbl>    <dbl>  <dbl>    <int>          <int>    <int>
#> 1 BOS       21.0     38.0 -0.447     1450           1500     1547
#> 2 ATL       65.0    112   -0.420     1709           1700     1923
#> 3 GSP       55.0     92.0 -0.402     2040           2025     2225
#> 4 BOS       23.0     38.0 -0.395     1954           2000     2131
#> 5 BNA       70.0    113   -0.381     1914           1910     2045
#> 6 MSP       93.0    149   -0.376     1558           1513     1745
#> # ... with 9 more rows, and 1 more variable: sched_arr_time <int>
```

I could also try a z-score. Though the standard deviation and mean will be affected by large delays.

```
flights %>%
  filter(!is.na(air_time)) %>%
  group_by(dest) %>%
  mutate(air_time_mean = mean(air_time),
         air_time_sd = sd(air_time),
         z_score = (air_time - air_time_mean) / air_time_sd) %>%
  arrange(z_score) %>%
  select(z_score, air_time_mean, air_time_sd, air_time, dep_time, sched_dep_time, arr_time, sched_arr_t
#> Adding missing grouping variables: `dest`
#> # A tibble: 327,346 x 9
#> # Groups:   dest [104]
#>    dest  z_score air_time_mean air_time_sd air_time dep_time sched_dep_time
#>    <chr>   <dbl>         <dbl>       <dbl>    <dbl>    <int>          <int>
#> 1 MSP     -4.90         151         11.8      93.0     1558           1513
#> 2 ATL     -4.88         113          9.81     65.0     1709           1700
#> 3 GSP     -4.72          93.4        8.13     55.0     2040           2025
#> 4 BNA     -4.05         114         11.0      70.0     1914           1910
#> 5 CVG     -3.98          96.0        8.52     62.0     1359           1343
#> 6 BOS     -3.63          39.0        4.95     21.0     1450           1500
#> # ... with 3.273e+05 more rows, and 2 more variables: arr_time <int>,
#> #   sched_arr_time <int>
```

```
flights %>%
  filter(!is.na(air_time)) %>%
  group_by(dest) %>%
  mutate(air_time_diff = air_time - min(air_time)) %>%
  arrange(desc(air_time_diff)) %>%
  select(dest, year, month, day, carrier, flight, air_time_diff, air_time, dep_time, arr_time) %>%
  head()
#> # A tibble: 6 x 10
#> # Groups:   dest [5]
#>    dest   year month   day carrier flight air_time_diff air_time dep_time
#>    <chr> <int> <int> <int> <chr>    <int>         <dbl>    <dbl>    <int>
#> 1 SFO    2013     7    28 DL         841           195      490     1727
#> 2 LAX    2013    11    22 DL         426           165      440     1812
#> 3 EGE    2013     1    28 AA         575           163      382     1806
```

```
#> 4 DEN    2013     9    10 UA         745          149      331     1513
#> 5 LAX    2013     7    10 DL          17          147      422     1814
#> 6 LAS    2013    11    22 UA         587          143      399     2142
#> # ... with 1 more variable: arr_time <int>
```

5.7.7 Exercise 7

> Find all destinations that are flown by at least two carriers. Use that information to rank the carriers.

The carrier that flies to the most locations is ExpressJet Airlines (EV). ExpressJet is a regional airline and partner for major airlines, so its one of those that flies small planes to close airports

```
flights %>%
  group_by(dest, carrier) %>%
  count(carrier) %>%
  group_by(carrier) %>%
  count(sort = TRUE)
#> # A tibble: 16 x 2
#> # Groups:   carrier [16]
#>   carrier    nn
#>   <chr>   <int>
#> 1 EV         61
#> 2 9E         49
#> 3 UA         47
#> 4 B6         42
#> 5 DL         40
#> 6 MQ         20
#> # ... with 10 more rows
```

```
filter(airlines, carrier == "EV")
#> # A tibble: 1 x 2
#>   carrier name
#>   <chr>   <chr>
#> 1 EV      ExpressJet Airlines Inc.
```

Chapter 6

Workflow: scripts

6.1 Running code

No exercises

6.2 RStudio diagnostics

No exercises

6.3 Practice

6.3.1 Exercise 1

> Go to the RStudio Tips twitter account, https://twitter.com/rstudiotips and find one tip that looks interesting. Practice using it!

This is by its very nature left to the reader.

6.3.2 Exercise 2

> What other common mistakes will RStudio diagnostics report? Read https://support.rstudio.com/hc/en-us/articles/205753617-Code-Diagnostics to find out.

The user should read that help page. However, to preview its contents, some other diagnostics for R code include:

1. Check for Missing, unmatched, partially matched, and too many arguments to functions
2. Warn if a variable is not defined
3. Warn if a variable is defined but not used
4. Style diagnostics to ensure the code conforms to the tidyverse style guide.

Chapter 7

Exploratory Data Analysis

7.1 Introduction

```
library("tidyverse")
library("viridis")
library("forcats")
```

This will also use data from **nycflights13**,

```
library("nycflights13")
```

7.2 Questions

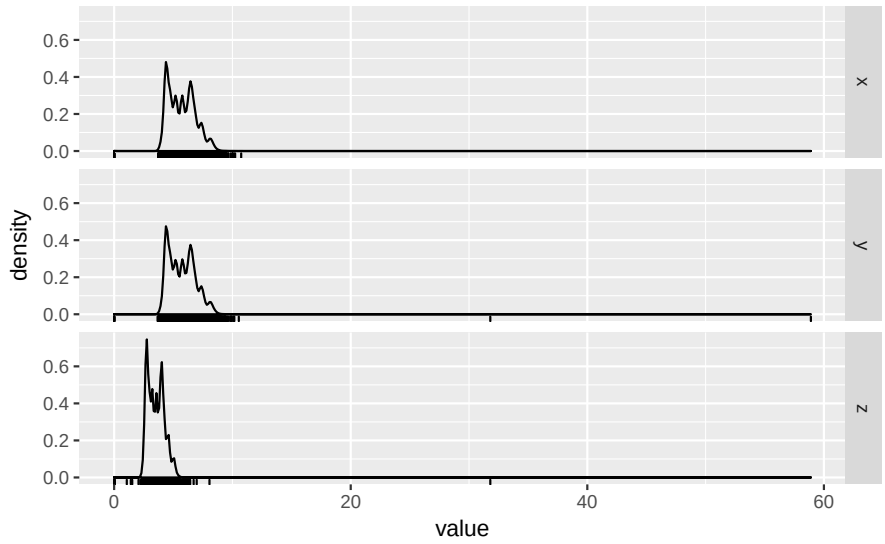
7.3 Variation

7.3.1 Exercise 1

> Explore the distribution of each of the x, y, and z variables in diamonds. What do you learn? Think about a diamond and how you might decide which dimension is the length, width, and depth.

In order to make it easier to plot them, I'll reshape the dataset so that I can use the variables as facets.

```
diamonds %>%
  mutate(id = row_number()) %>%
  select(x, y, z, id) %>%
  gather(variable, value, -id)  %>%
  ggplot(aes(x = value)) +
  geom_density() +
  geom_rug() +
  facet_grid(variable ~ .)
```



There several noticeable features of the distributions

1. They are right skewed, with most diamonds small, but a few very large ones.
2. There is an outlier in `y`, and `z` (see the rug)
3. All three distributions have a bimodality (perhaps due to some sort of threshold)

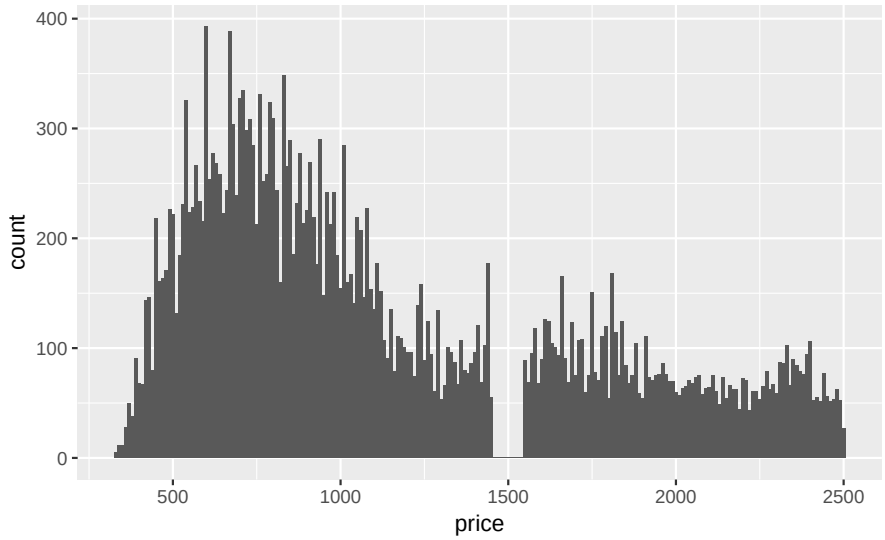
According to the documentation for `diamonds`: `x` is length, `y` is width, and `z` is depth. I don't know if I would have figured that out before; maybe if there was data on the type of cuts.

7.3.2 Exercise 2.

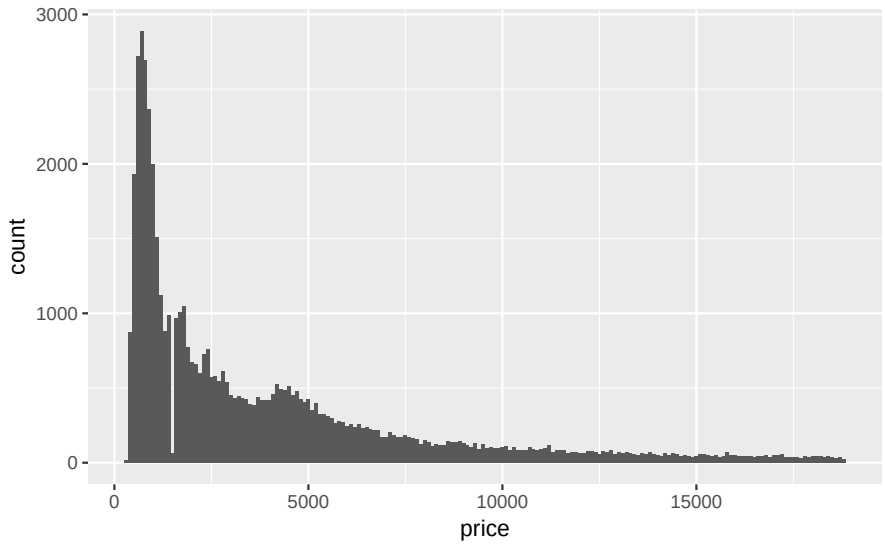
> Explore the distribution of price. Do you discover anything unusual or surprising? (Hint: Carefully think about the `binwidth` and make sure you try a wide range of values.)

- The price data has many spikes, but I can't tell what each spike corresponds to. The following plots don't show much difference in the distributions in the last one or two digits.
- There are no diamonds with a price of $1,500
- There's a bulge in the distribution around $7,500.

```
ggplot(filter(diamonds, price < 2500), aes(x = price)) +
  geom_histogram(binwidth = 10, center = 0)
```

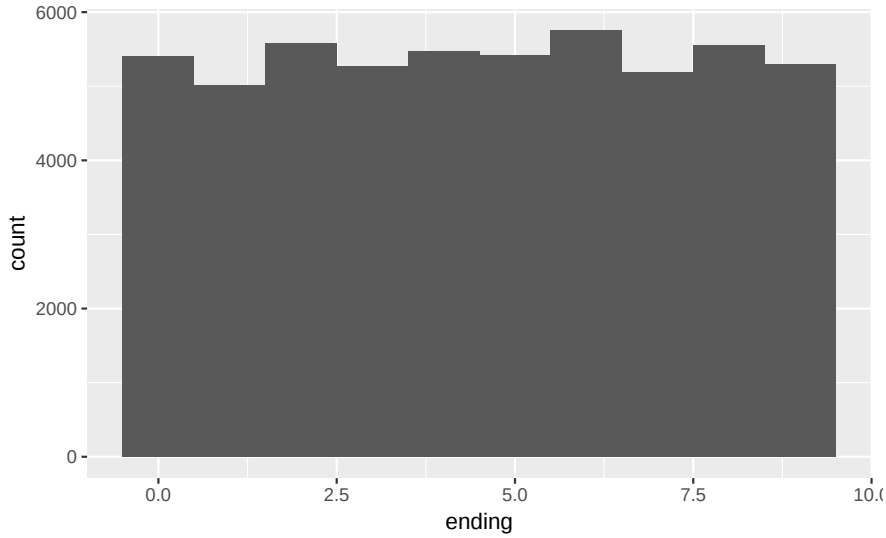


```
ggplot(filter(diamonds), aes(x = price)) +
  geom_histogram(binwidth = 100, center = 0)
```

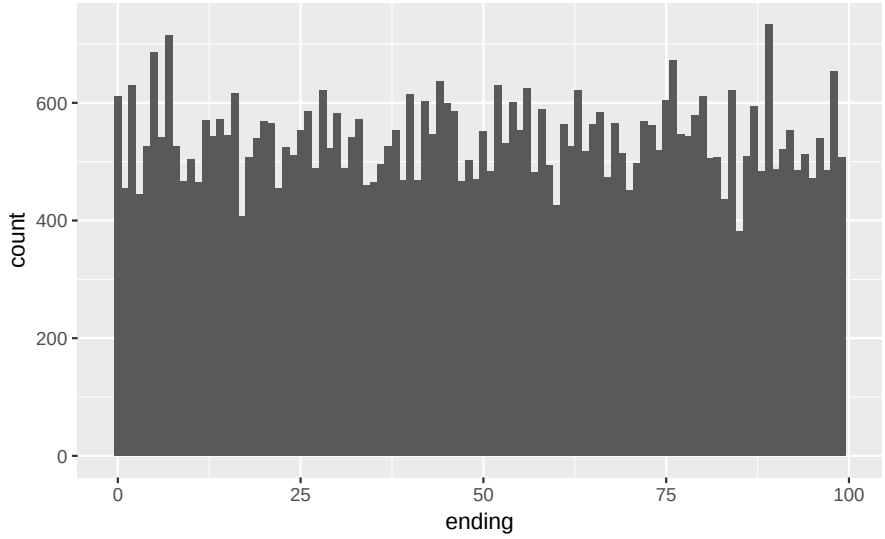


Distribution of last digit

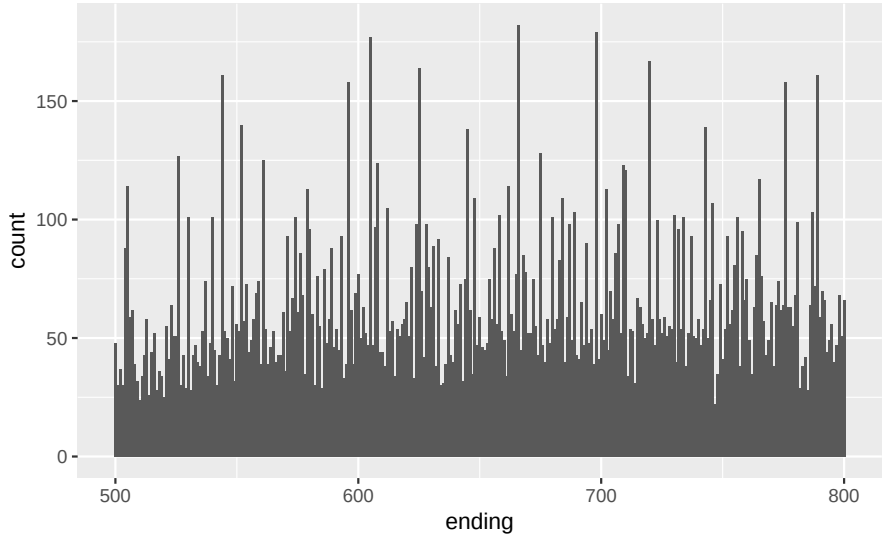
```
diamonds %>%
  mutate(ending = price %% 10) %>%
  ggplot(aes(x = ending)) +
  geom_histogram(binwidth = 1, center = 0) +
  geom_bar()
```




```
diamonds %>%
  mutate(ending = price %% 100) %>%
  ggplot(aes(x = ending)) +
  geom_histogram(binwidth = 1) +
  geom_bar()
```




```
diamonds %>%
  mutate(ending = price %% 1000) %>%
  filter(ending >= 500, ending <= 800)  %>%
  ggplot(aes(x = ending)) +
  geom_histogram(binwidth = 1) +
  geom_bar()
```



7.3.3 Exercise 3.

How many diamonds are 0.99 carat? How many are 1 carat? What do you think is the cause of the difference?

There are more than 70 times as many 1 carat diamonds as 0.99 carat diamond.

```
diamonds %>%
  filter(carat >= 0.99, carat <= 1) %>%
  count(carat)
#> # A tibble: 2 x 2
#>   carat     n
```

```
#>   <dbl> <int>
#> 1 0.990    23
#> 2 1.00   1558
```

I don't know exactly the process behind how carats are measured, but some way or another some diamonds carat values are being "rounded up", because presumably there is a premium for a 1 carat diamond vs. a 0.99 carat diamond beyond the expected increase in price due to a 0.01 carat increase.

To check this intuition, we'd want to look at the number of diamonds in each carat range to seem if there is an abnormally low number at 0.99 carats, and an abnormally high number at 1 carat.

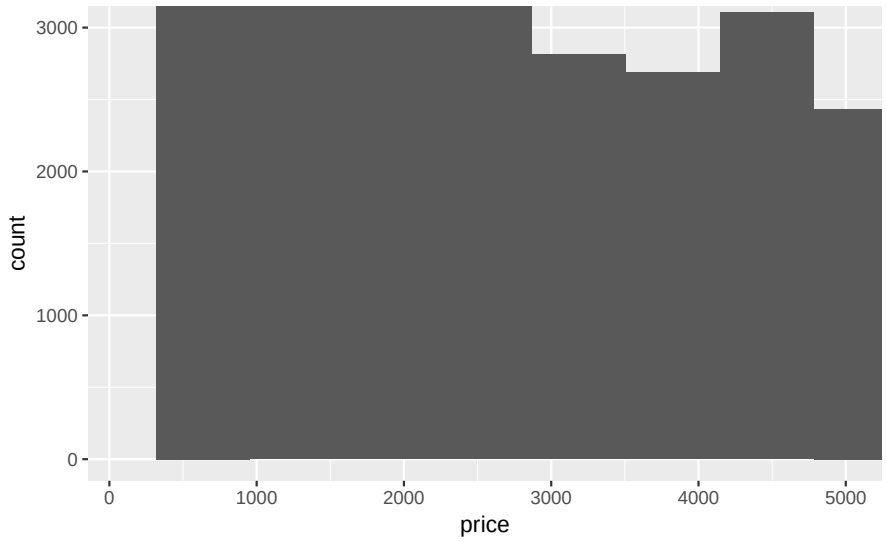
```
diamonds %>%
  filter(carat >= 0.9, carat <= 1.1) %>%
  count(carat) %>%
  print(n = 30)
#> # A tibble: 21 x 2
#>    carat     n
#>    <dbl> <int>
#>  1 0.900  1485
#>  2 0.910   570
#>  3 0.920   226
#>  4 0.930   142
#>  5 0.940    59
#>  6 0.950    65
#>  7 0.960   103
#>  8 0.970    59
#>  9 0.980    31
#> 10 0.990    23
#> 11 1.00   1558
#> 12 1.01   2242
#> 13 1.02    883
#> 14 1.03    523
#> 15 1.04    475
#> 16 1.05    361
#> 17 1.06    373
#> 18 1.07    342
#> 19 1.08    246
#> 20 1.09    287
#> 21 1.10    278
```

7.3.4 Exercise 4

> Compare and contrast `coord_cartesian()` vs `xlim()` or `ylim()` when zooming in on a histogram. What happens if you leave `binwidth` unset? What happens if you try and zoom so only half a bar shows?

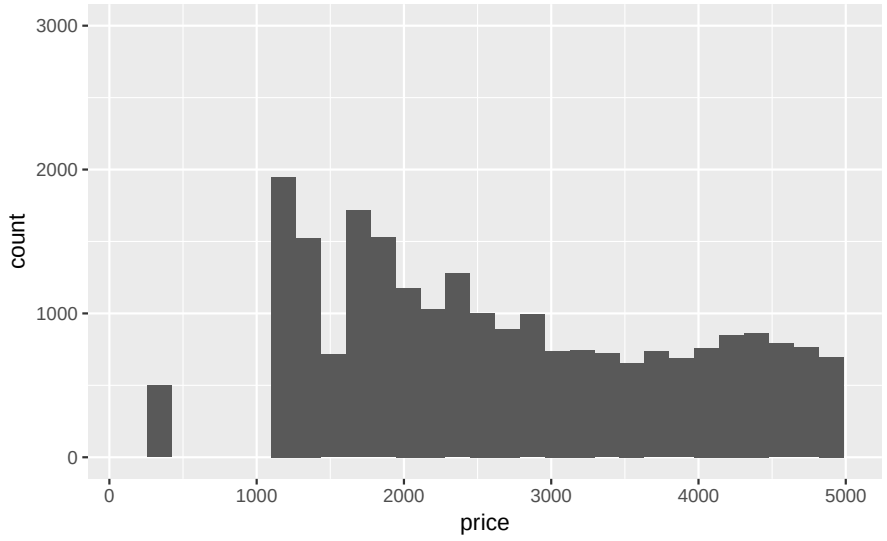
`coord_cartesian` simply zooms in on the area specified by the limits. The calculation of the histogram is unaffected.

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = price)) +
  coord_cartesian(xlim = c(100, 5000), ylim = c(0, 3000))
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



However, the `xlim` and `ylim` functions first drop any values outside the limits (the `ylim` doesn't matter in this case), then calculates the histogram, and draws the graph with the given limits.

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = price)) +
  xlim(100, 5000) +
  ylim(0, 3000)
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
#> Warning: Removed 14714 rows containing non-finite values (stat_bin).
#> Warning: Removed 5 rows containing missing values (geom_bar).
```



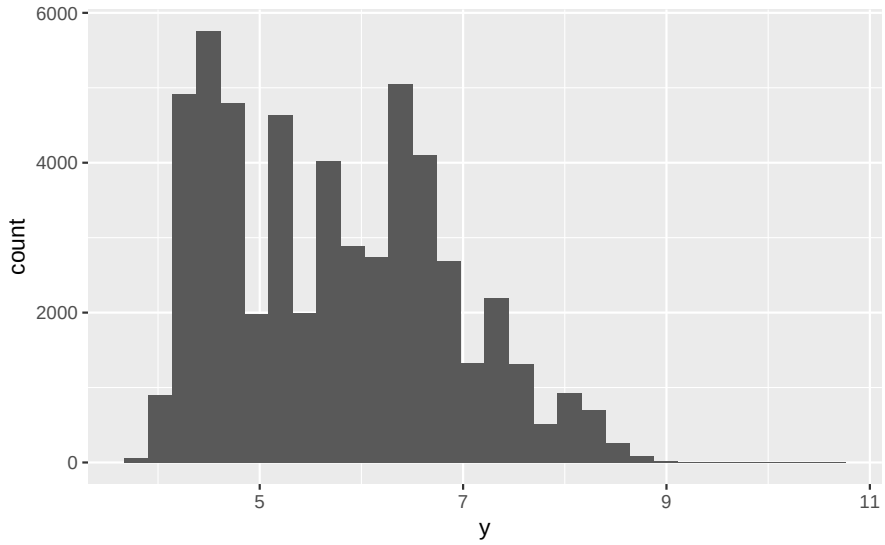
7.4 Missing Values

7.4.1 Exercise 1

What happens to missing values in a histogram? What happens to missing values in a bar chart?
> Why is there a difference?

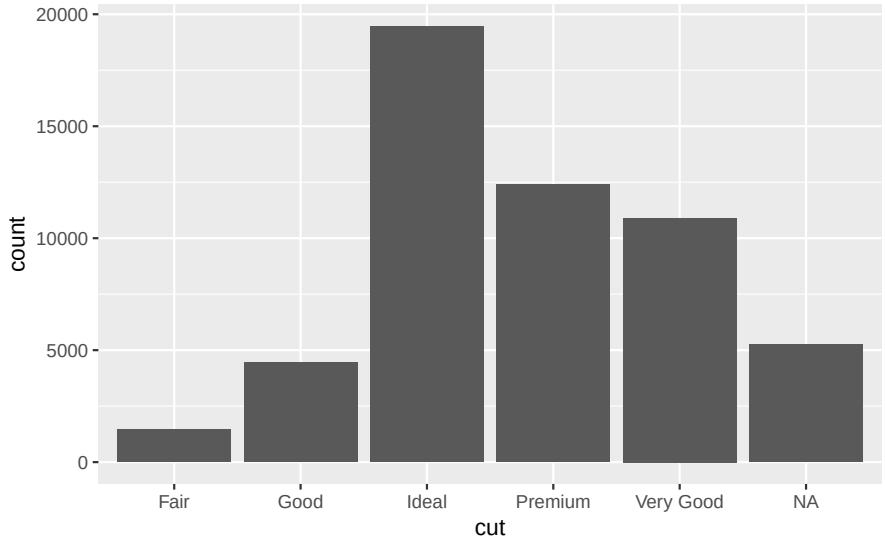
Missing values are removed when the number of observations in each bin are calculated. See the warning message: `Removed 9 rows containing non-finite values (stat_bin)`

```
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))

ggplot(diamonds2, aes(x = y)) +
  geom_histogram()
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
#> Warning: Removed 9 rows containing non-finite values (stat_bin).
```



In `geom_bar`, `NA` is treated as another category. This is because the `x` aesthetic in `geom_bar` should be a discrete (categorical) variable, and missing values are just another category.

```
diamonds %>%
  mutate(cut = if_else(runif(n()) < 0.1, NA_character_, as.character(cut))) %>%
  ggplot() +
  geom_bar(mapping = aes(x = cut))
```



In a histogram, the `x` aesthetic variable needs to be numeric, and `stat_bin` groups the observations by ranges into bins. Since the numeric value of the `NA` observations is unknown, they cannot be placed in a particular

bin, and are dropped.

7.4.2 Exercise 2

> What does `na.rm = TRUE` do in `mean()` and `sum()`?

This option removes `NA` values from the vector prior to calculating the mean and sum.

```
mean(c(0, 1, 2, NA), na.rm = TRUE)
#> [1] 1
sum(c(0, 1, 2, NA), na.rm = TRUE)
#> [1] 3
```

7.5 Covariation

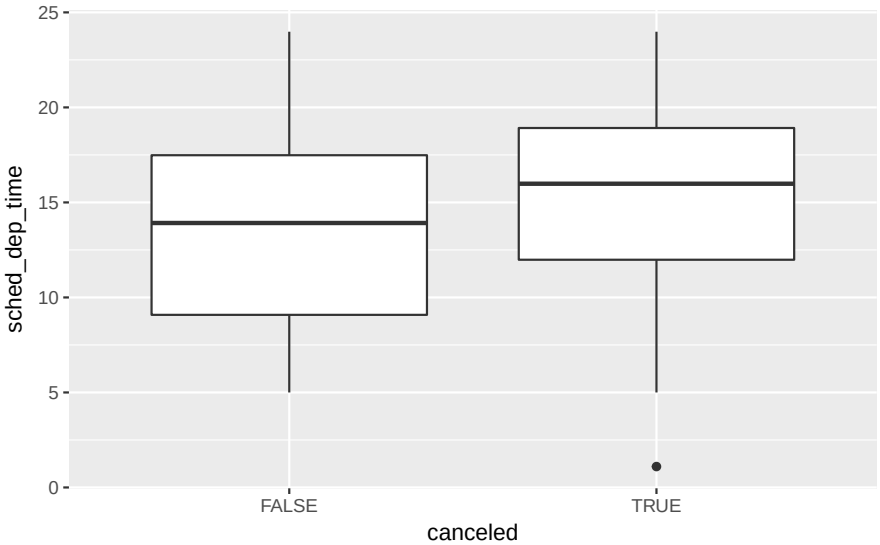
7.5.1 A categorical and continuous variable

7.5.1.1 Exercise 1

> Use what you've learned to improve the visualization of the departure times of canceled vs. non-canceled flights.

Instead of a `freqplot` use a box-plot

```
nycflights13::flights %>%
  mutate(
    canceled = is.na(dep_time),
    sched_hour = sched_dep_time %/% 100,
    sched_min = sched_dep_time %% 100,
    sched_dep_time = sched_hour + sched_min / 60
  ) %>%
  ggplot() +
    geom_boxplot(mapping = aes(y = sched_dep_time, x = canceled))
```



7.5.1.2 Exercise 2

> What variable in the diamonds dataset is most important for predicting the price of a diamond? > How is that variable correlated with Why does the combination of those two relationships lead to lower quality diamonds being more expensive?

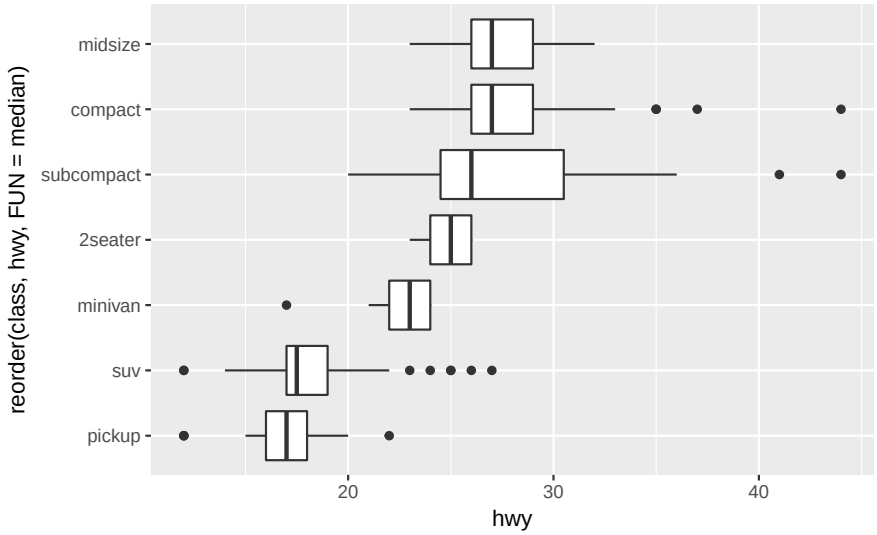
TODO I'm unsure what this question is asking conditional on using only the tools introduced in the book thus far.

7.5.1.3 Exercise 3

> Install the **ggstance** package, and create a horizontal box plot. How does this compare to using `coord_flip()`?

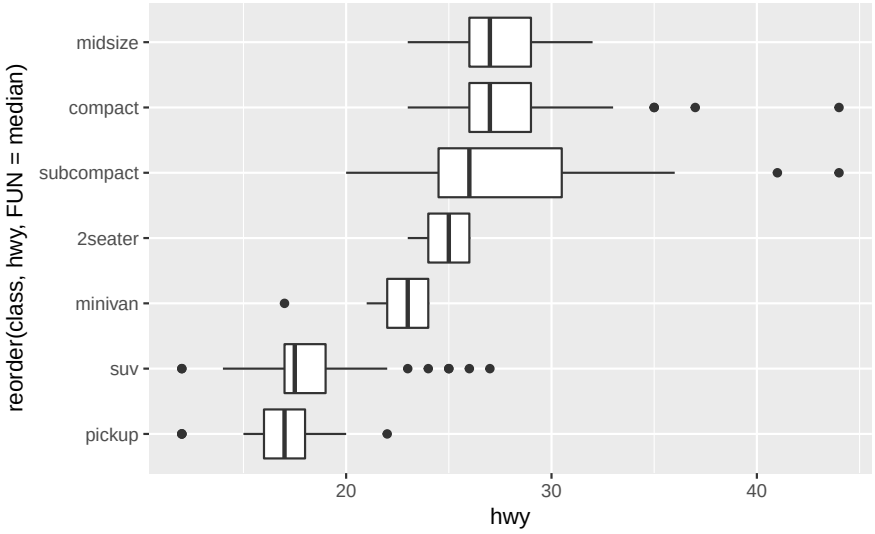
Earlier we created a horizontal box plot of the distribution `hwy` by `class`, using `geom_boxplot` and `coord_flip`:

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



In this case the output looks the same, but in the aesthetics the `x` and `y` are flipped from the previous case.

```
library("ggstance")
#>
#> Attaching package: 'ggstance'
#> The following objects are masked from 'package:ggplot2':
#>
#>     geom_errorbarh, GeomErrorbarh

ggplot(data = mpg) +
  geom_boxploth(mapping = aes(y = reorder(class, hwy, FUN = median), x = hwy))
```



7.5.1.4 Exercise 4

> One problem with box plots is that they were developed in an era of much smaller datasets and tend to display a prohibitively large number of ‘`outlying values''. One approach to remedy this problem is the letter value plot. Install the **lvplot** package, and try using`geom_lv()‘ to display the distribution of price vs cut. What do you learn? How do you interpret the plots?

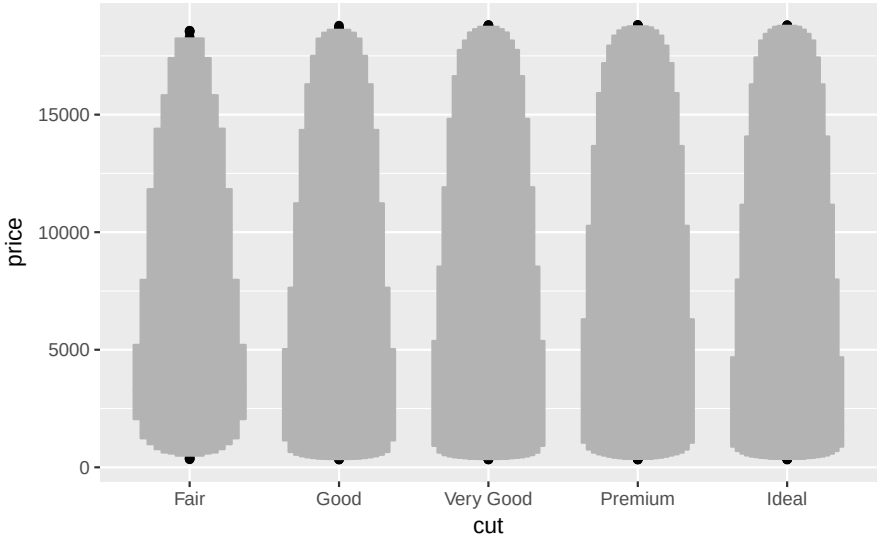
The boxes of the letter-value plot correspond to many more quantiles. They are useful for larger datasets because

1. larger datasets can give precise estimates of quantiles beyond the quartiles, and
2. in expectation, larger datasets should have more outliers (in absolute numbers).

The letter-value plot is described in:

> Heike Hofmann, Karen Kafadar, and Hadley Wickham. 2011. “Letter-value plots: Boxplots for large data” http://vita.had.co.nz/papers/letter-value-plot.pdf

```
library("lvplot")
ggplot(diamonds, aes(x = cut, y = price)) +
  geom_lv()
```

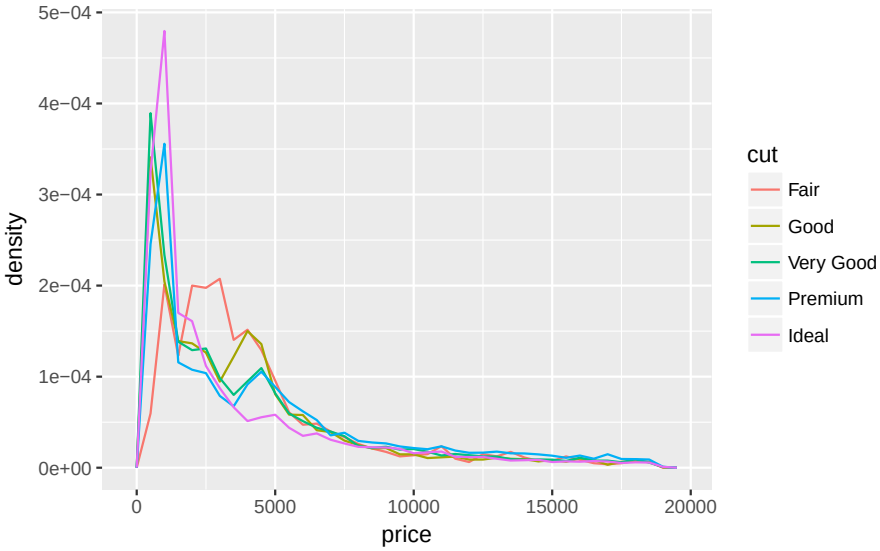


7.5.1.5 Exercise 5

> Compare and contrast `geom_violin()` with a faceted `geom_histogram()`, or a colored `geom_freqpoly()`. What are the pros and cons of each method?

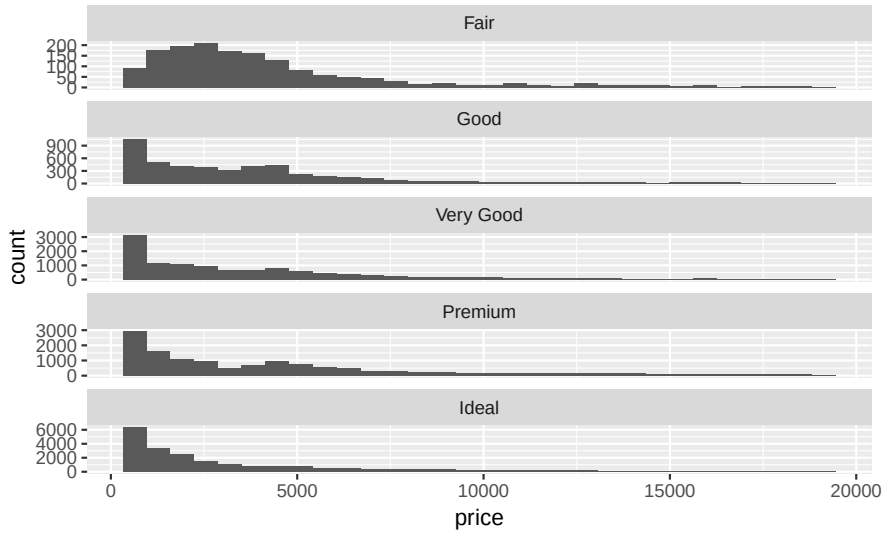
I produce plots for these three methods below. The `geom_freqpoly` is better for look-up: meaning that given a price, it is easy to tell which `cut` has the highest density. However, the overlapping lines makes it difficult to distinguish how the overall distributions relate to each other. The `geom_violin` and faceted `geom_histogram` have similar strengths and weaknesses. It is easy to visually distinguish differences in the overall shape of the distributions (skewness, central values, variance, etc). However, since we can't easily compare the vertical values of the distribution, its difficult to look up which category has the highest density for a given price. All of these methods depend on tuning parameters to determine the level of smoothness of the distribution.

```
ggplot(data = diamonds, mapping = aes(x = price, y = ..density..)) +
  geom_freqpoly(mapping = aes(colour = cut), binwidth = 500)
```

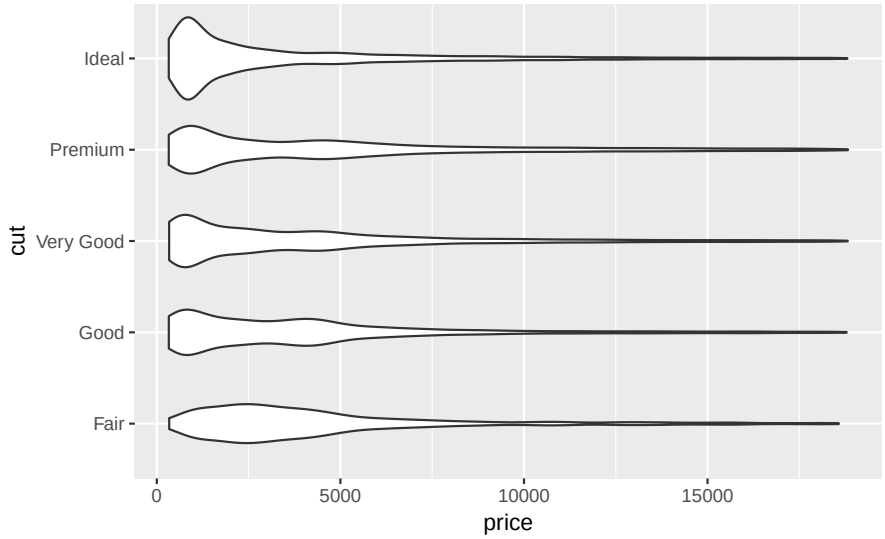



```
ggplot(data = diamonds, mapping = aes(x = price)) +
  geom_histogram() +
```

```
  facet_wrap(~ cut, ncol = 1, scales = "free_y")
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +
  geom_violin() +
  coord_flip()
```



The violin plot was first described in

> Hintze JL, Nelson RD (1998). "Violin Plots: A Box Plot-Density Trace Synergism." The American Statistician, 52(2), 181–184

7.5.1.6 Exercise 6

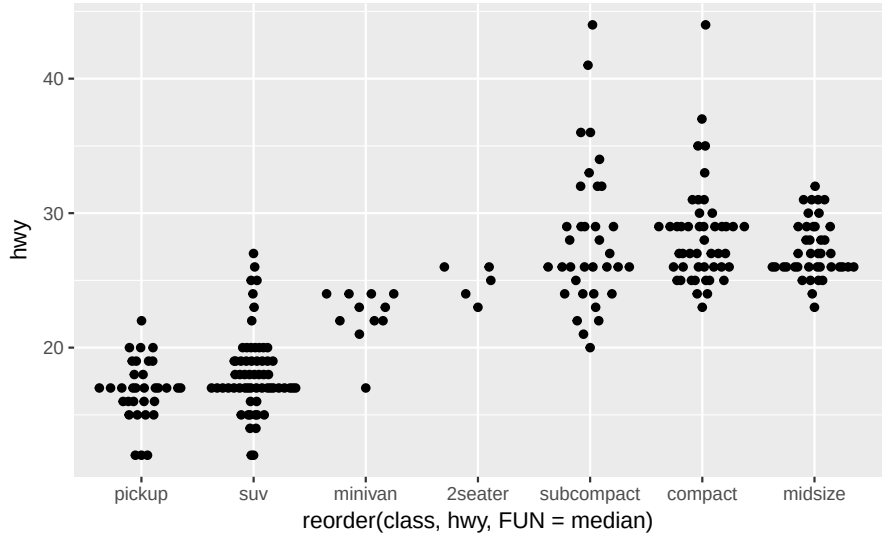
> If you have a small dataset, it's sometimes useful to use `geom_jitter()` to see the relationship between a continuous and categorical variable. The **ggbeeswarm** package provides a number of methods similar to `geom_jitter()`. List them and briefly describe what each one does.

There are two methods:

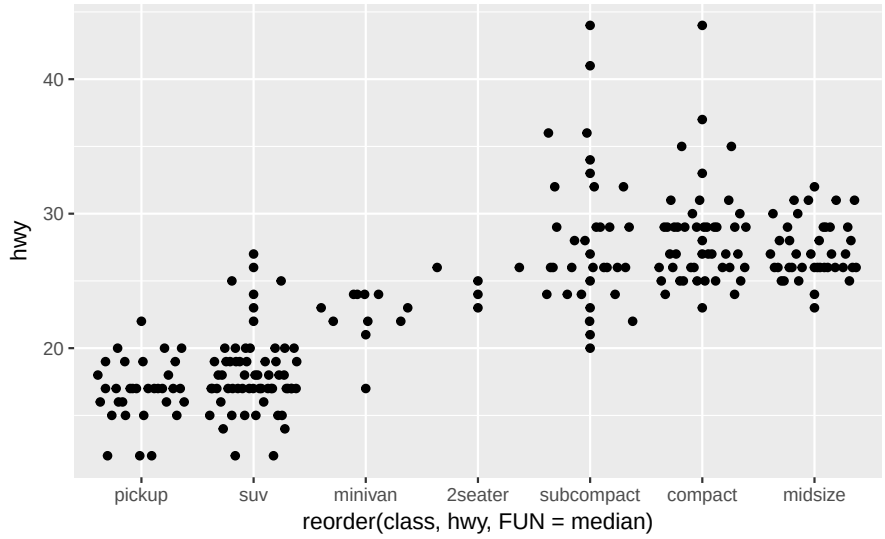
- `geom_quasirandom` that produces plots that resemble something between jitter and violin. There are several different methods that determine exactly how the random location of the points is generated.
- `geom_beeswarm` creates a shape similar to a violin plot, but by offsetting the points.

I'll use the `mpg` box plot example since these methods display individual points, they are better suited for smaller datasets.

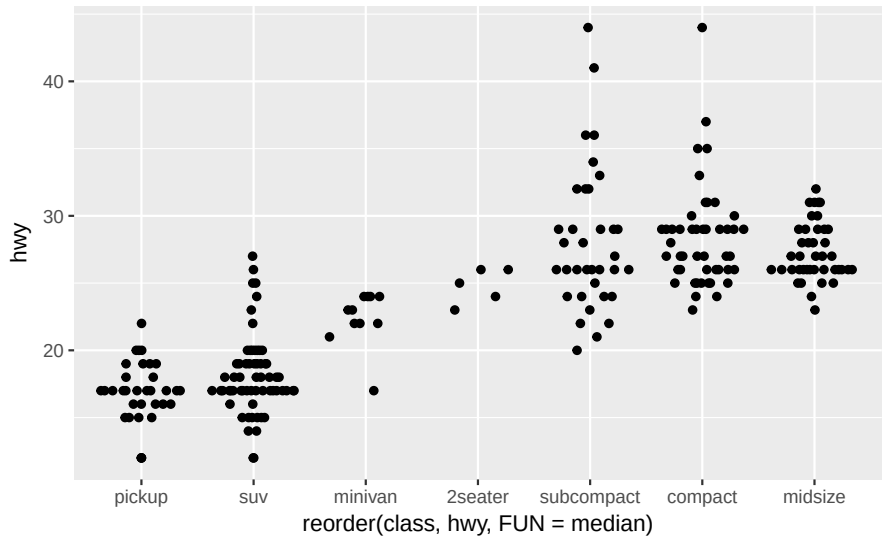
```
library("ggbeeswarm")
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy))
```



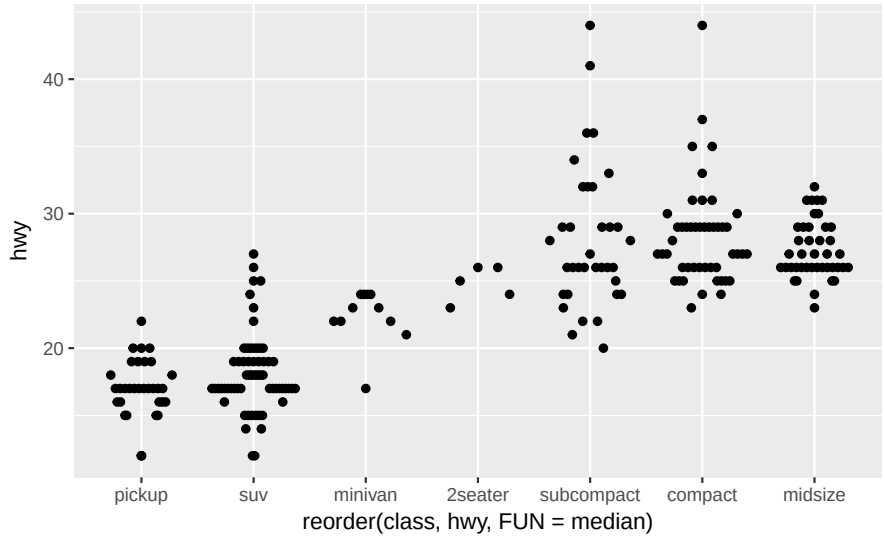

```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy),
                   method = "tukey")
```



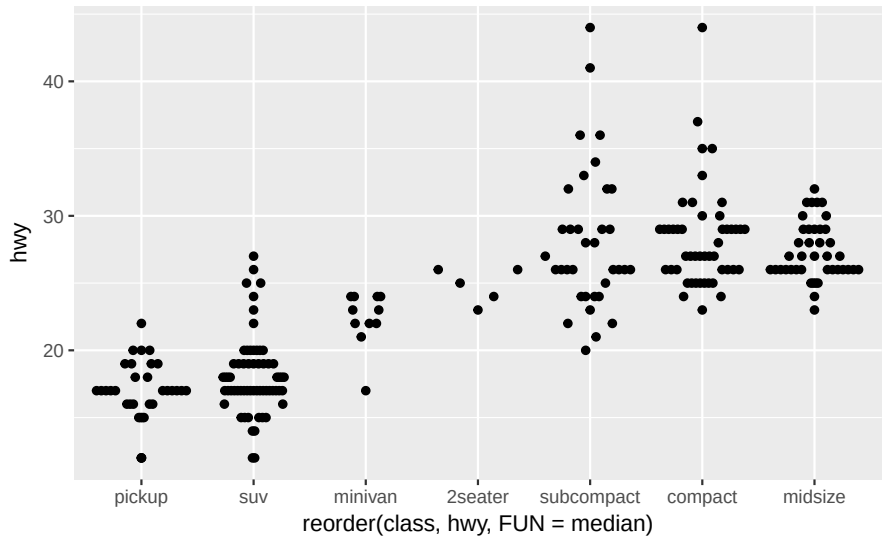

```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy),
                   method = "tukeyDense")
```



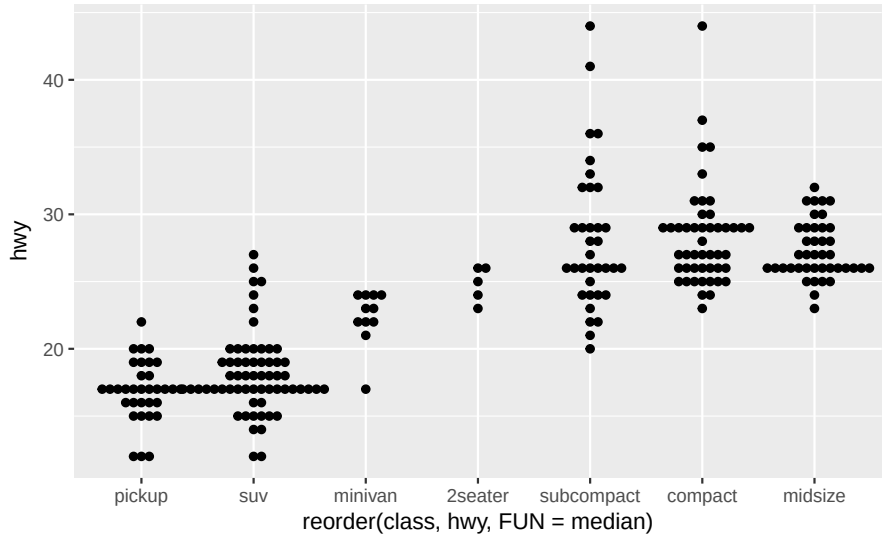

```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy),
                   method = "frowney")
```




```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy),
                   method = "smiley")
```




```
ggplot(data = mpg) +
  geom_beeswarm(mapping = aes(x = reorder(class, hwy, FUN = median),
                              y = hwy))
```



7.5.2 Two categorical variables

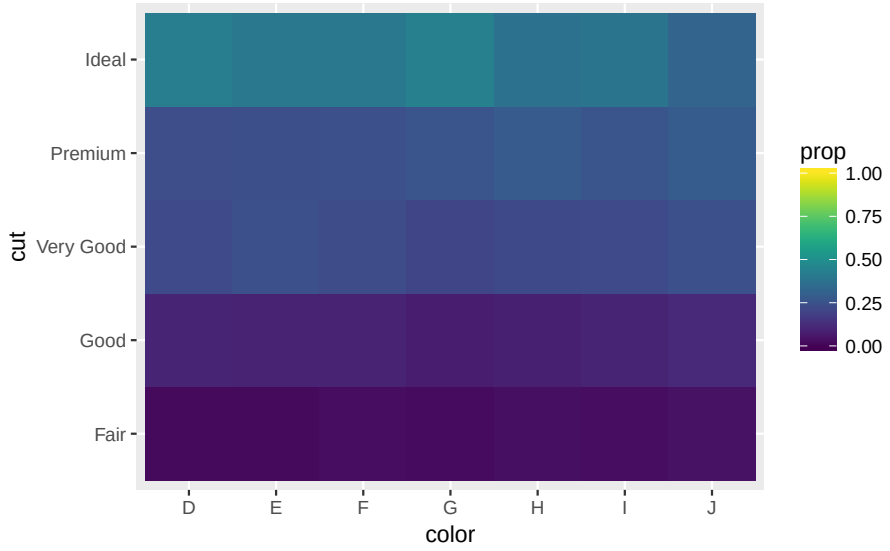
7.5.2.1 Exercise 1

> How could you rescale the count dataset above to more clearly show the distribution of cut within color, or color within cut?

To clearly show the distribution of `cut` within `color`, calculate a new variable `prop` which is the proportion of each cut within a `color`. This is done using a grouped mutate.

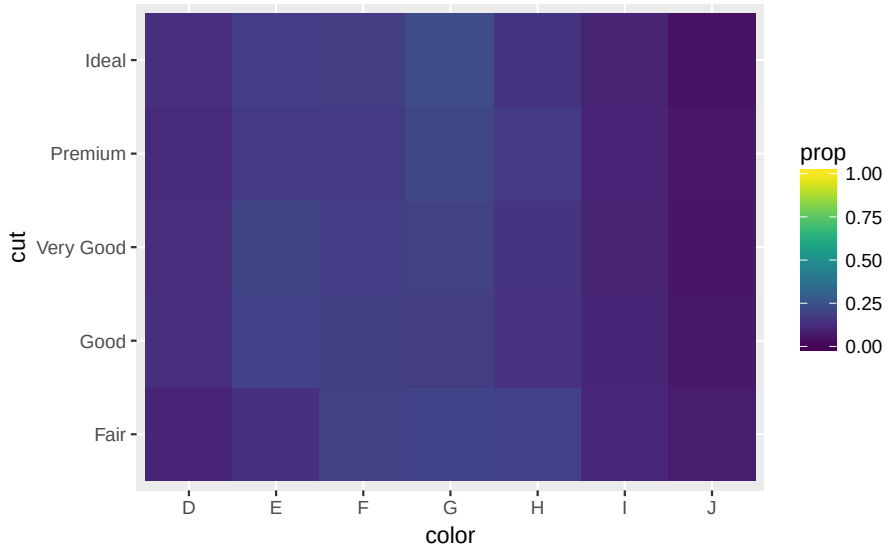
```
diamonds %>%
  count(color, cut) %>%
  group_by(color) %>%
  mutate(prop = n / sum(n)) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
```

```
  geom_tile(mapping = aes(fill = prop)) +
  scale_fill_viridis(limits = c(0, 1))
```



Similarly, to scale by the distribution of `color` within `cut`,

```
diamonds %>%
  count(color, cut) %>%
  group_by(cut) %>%
  mutate(prop = n / sum(n)) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
  geom_tile(mapping = aes(fill = prop)) +
  scale_fill_viridis(limits = c(0, 1))
```

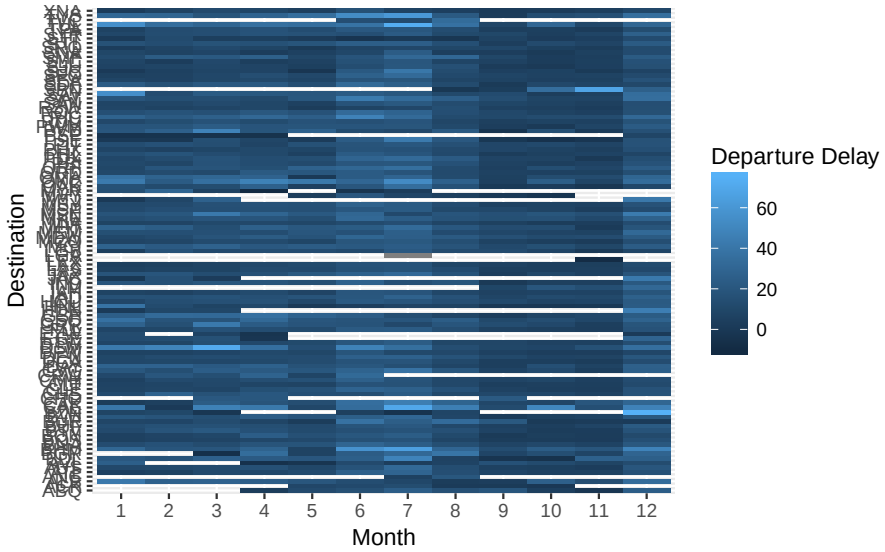


I add `limit = c(0, 1)` to put the color scale between (0, 1). These are the logical boundaries of proportions. This makes it possible to compare each cell to its actual value, and would improve comparisons across multiple plots. However, it ends up limiting the colors and makes it harder to compare within the dataset. However, using the default limits of the minimum and maximum values makes it easier to compare within the dataset the emphasizing relative differences, but harder to compare across datasets.

7.5.2.2 Exercise 2

Use `geom_tile()` together with **dplyr** to explore how average flight delays vary by destination and month of year. What makes the plot difficult to read? How could you improve it?

```
flights %>%
  group_by(month, dest) %>%
  summarise(dep_delay = mean(dep_delay, na.rm = TRUE)) %>%
  ggplot(aes(x = factor(month), y = dest, fill = dep_delay)) +
  geom_tile() +
  labs(x = "Month", y = "Destination", fill = "Departure Delay")
```

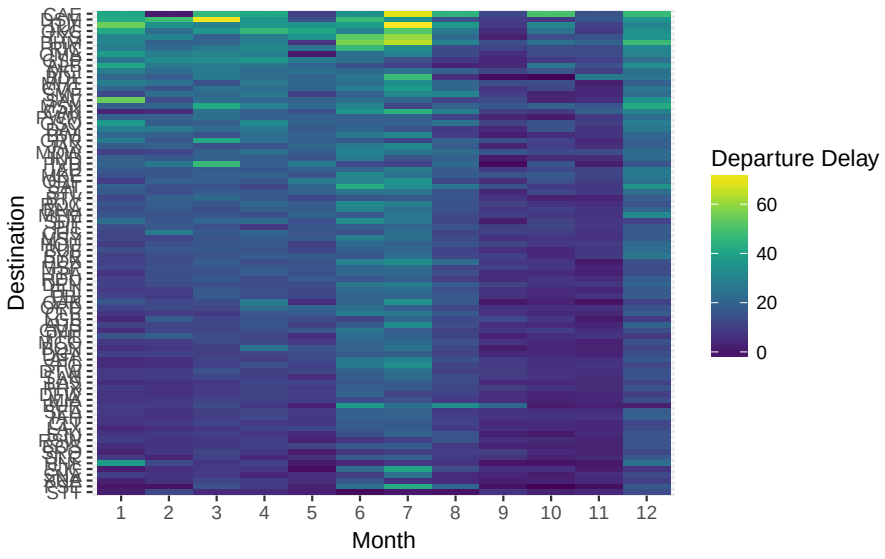


There are several things that could be done to improve it,

- sort destinations by a meaningful quantity (distance, number of flights, average delay)
- remove missing values
- better color scheme (viridis)

How to treat missing values is difficult. In this case, missing values correspond to airports which don't have regular flights (at least one flight each month) from NYC. These are likely smaller airports (with higher variance in their average due to fewer observations).

```
library("viridis")
flights %>%
  group_by(month, dest) %>%
  summarise(dep_delay = mean(dep_delay, na.rm = TRUE)) %>%
  group_by(dest) %>%
  filter(n() == 12) %>%
  ungroup() %>%
  mutate(dest = fct_reorder(dest, dep_delay)) %>%
  ggplot(aes(x = factor(month), y = dest, fill = dep_delay)) +
  geom_tile() +
  scale_fill_viridis() +
  labs(x = "Month", y = "Destination", fill = "Departure Delay")
```



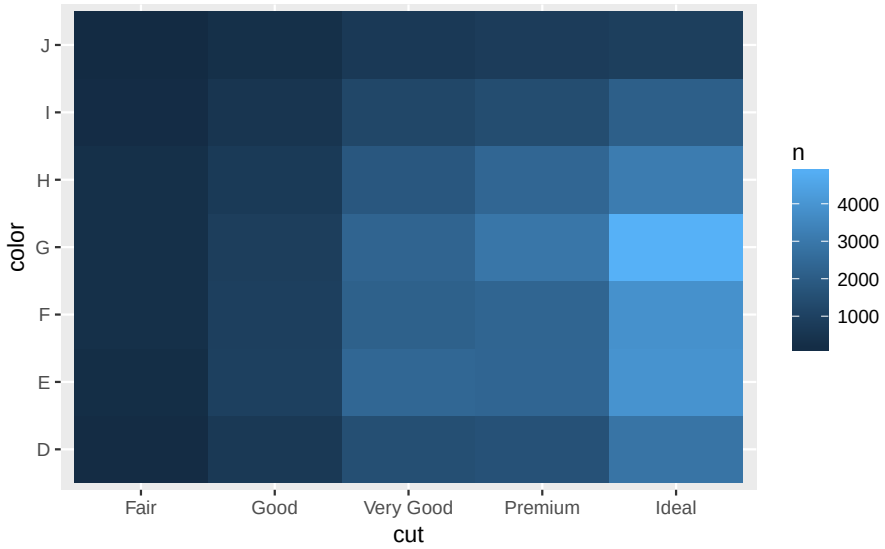
7.5.2.3 Exercise 3

> Why is it slightly better to use `aes(x = color, y = cut)` rather than `aes(x = cut, y = color)` in the example above?

It's usually better to use the categorical variable with a larger number of categories or the longer labels on the y axis. If at all possible, labels should be horizontal because that is easier to read.

However, switching the order doesn't result in overlapping labels.

```
diamonds %>%
  count(color, cut) %>%
  ggplot(mapping = aes(y = color, x = cut)) +
    geom_tile(mapping = aes(fill = n))
```



Another justification, for switching the order is that the larger numbers are at the top when `x = color` and `y = cut`, and that lowers the cognitive burden of interpreting the plot.

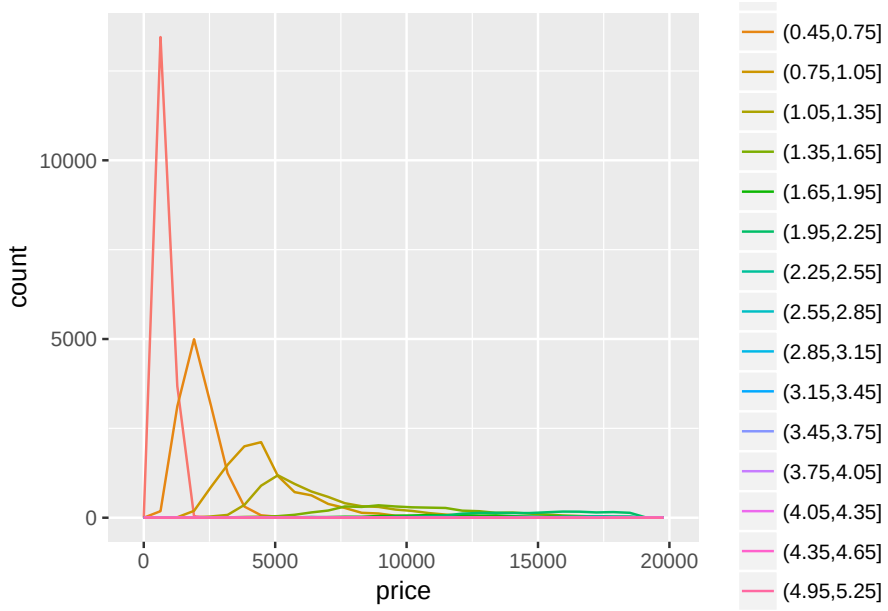
7.5.3 Two continuous variables

7.5.3.1 Exercise 1

> Instead of summarizing the conditional distribution with a box plot, you could use a frequency polygon. What do you need to consider when using `cut_width()` vs `cut_number()`? How does that impact a visualization of the 2d distribution of `carat` and `price`?

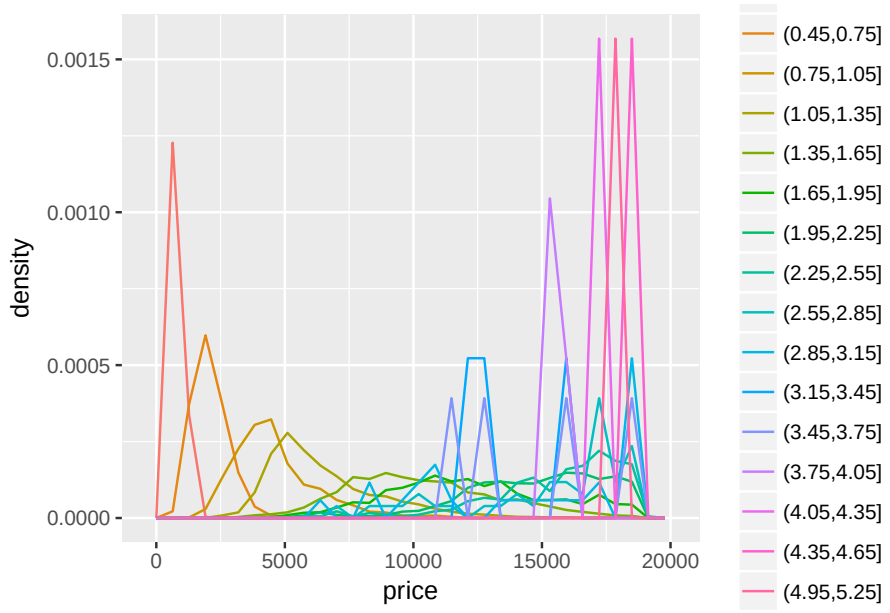
When using `cut_width` the number in each bin may be unequal. The distribution of `carat` is right skewed so there are few diamonds in those bins.

```
ggplot(data = diamonds,
       mapping = aes(x = price,
                     colour = cut_width(carat, 0.3))) +
  geom_freqpoly()
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



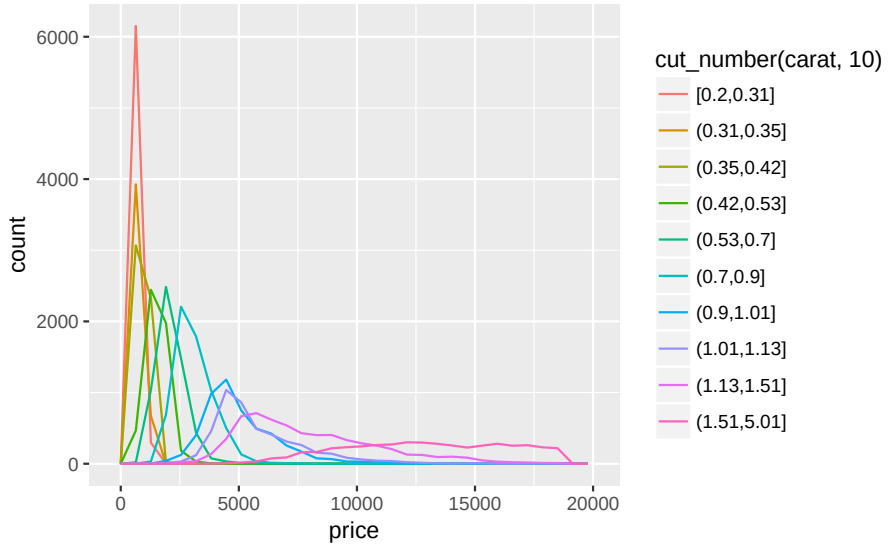
Plotting the density instead of counts will make the distributions comparable, although the bins with few observations will still be hard to interpret.

```
ggplot(data = diamonds,
       mapping = aes(x = price,
                     y = ..density..,
                     colour = cut_width(carat, 0.3))) +
  geom_freqpoly()
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



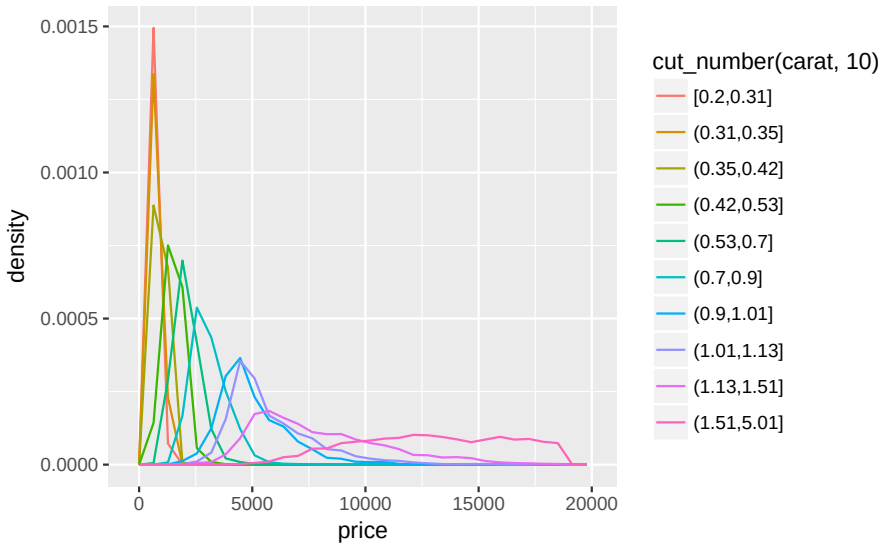
Plotting the density instead of counts will make the distributions comparable, although the bins with few observations will still be hard to interpret.

```
ggplot(data = diamonds,
       mapping = aes(x = price,
                     colour = cut_number(carat, 10))) +
  geom_freqpoly()
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Since there are equal numbers in each bin, the plot looks the same if density is used for the y aesthetic (although the values are on a different scale).

```
ggplot(data = diamonds,
       mapping = aes(x = price,
                     y = ..density..,
                     colour = cut_number(carat, 10))) +
  geom_freqpoly()
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

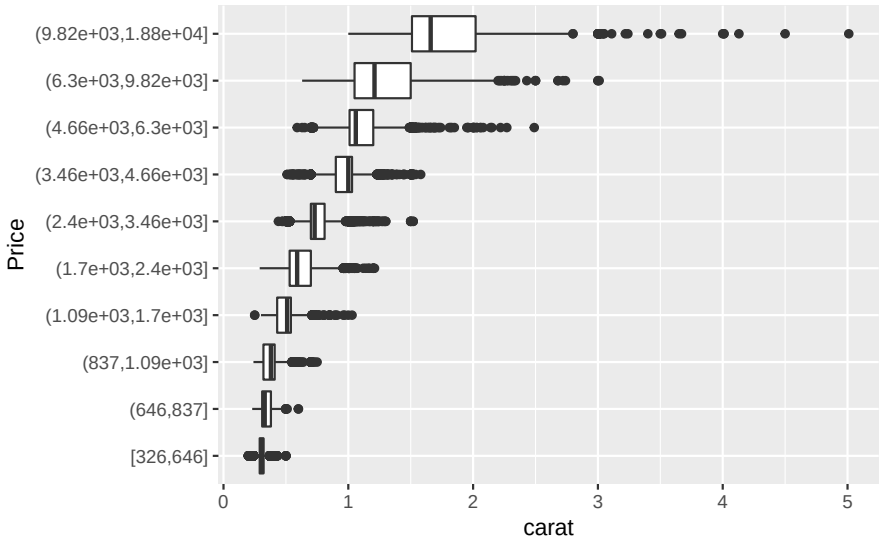


7.5.3.2 Exercise 2

Visualize the distribution of `carat`, partitioned by `price`.

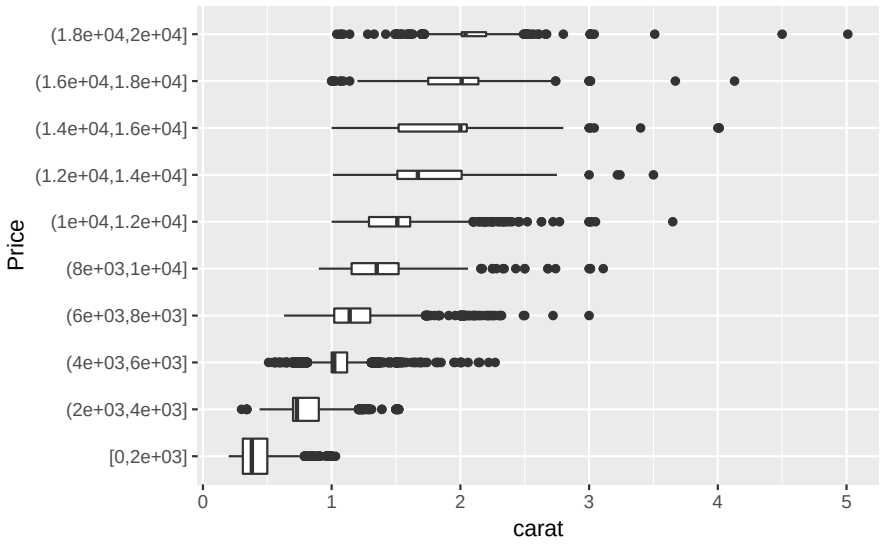
With a box plot, partitioning into an 10 bins with the same number of observations:

```
ggplot(diamonds, aes(x = cut_number(price, 10), y = carat)) +
  geom_boxplot() +
  coord_flip() +
  xlab("Price")
```



With a box plot, partitioning into an bins of $2,000 with the width of the box determined by the number of observations. I use `boundary = 0` to ensure the first bin goes from $0–$2,000.

```
ggplot(diamonds, aes(x = cut_width(price, 2000, boundary = 0), y = carat)) +
  geom_boxplot(varwidth = TRUE) +
  coord_flip() +
  xlab("Price")
```



7.5.3.3 Exercise 3

> How does the price distribution of very large diamonds compare to small diamonds. Is it as you expect, or does it surprise you?

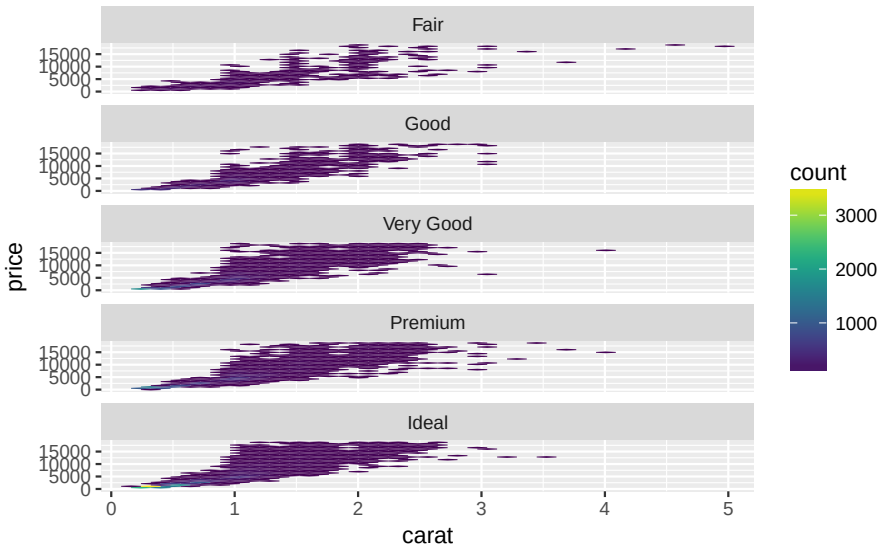
The distribution of very large diamonds is more variable. I'm not surprised, since I had a very weak prior about diamond prices. Ex post, I would reason that above a certain size other factors such as cut, clarity, color play more of a role in the price.

7.5.3.4 Exercise 4

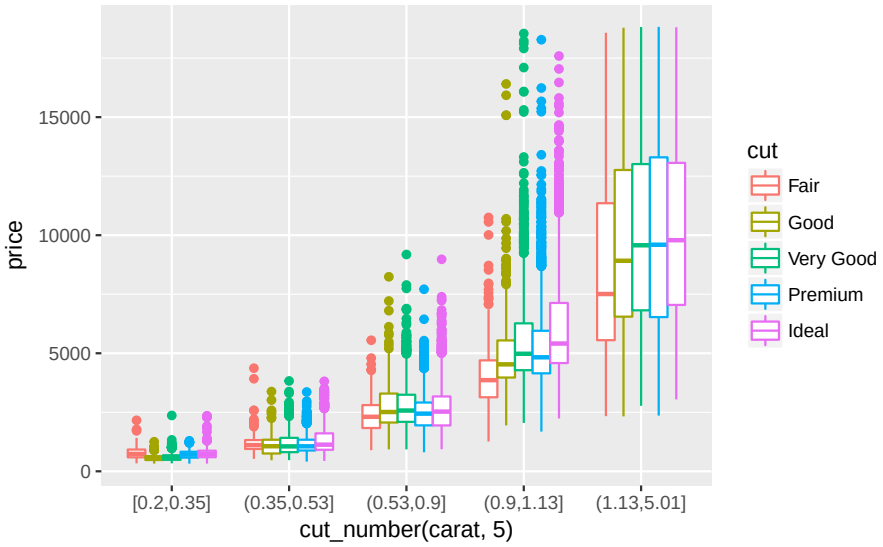
> Combine two of the techniques you've learned to visualize the combined distribution of cut, carat, and price.

There's lots of options to try, so readers may prodocue a variety of solutions. Here's a couple that I tried.

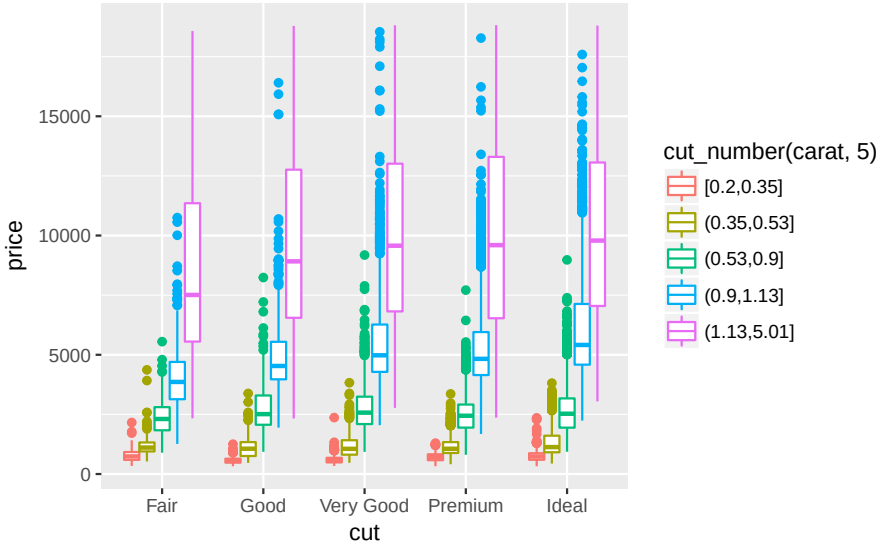
```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_hex() +
  facet_wrap(~ cut, ncol = 1) +
  scale_fill_viridis()
#> Loading required package: methods
```




```
ggplot(diamonds, aes(x = cut_number(carat, 5), y = price, color = cut)) +
  geom_boxplot()
```



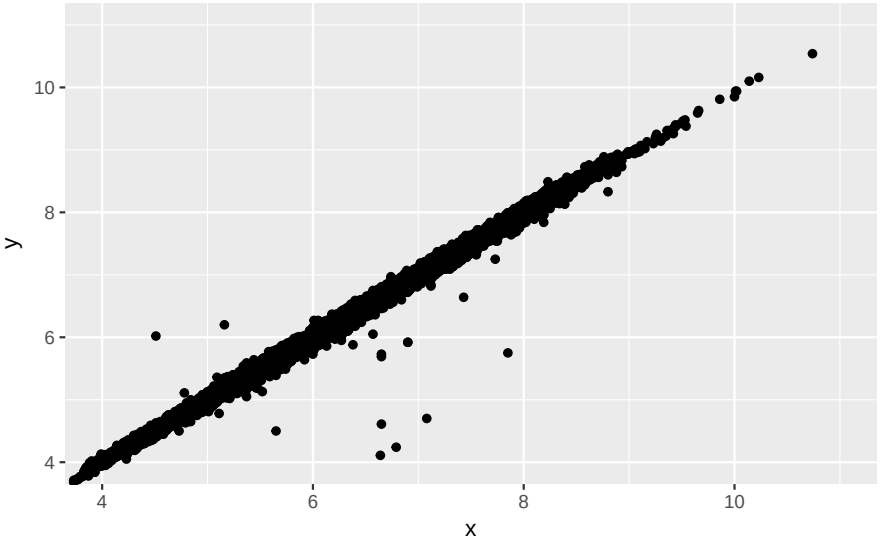

```
ggplot(diamonds, aes(color = cut_number(carat, 5), y = price, x = cut)) +
  geom_boxplot()
```



7.5.3.5 Exercise 5

Two dimensional plots reveal outliers that are not visible in one dimensional plots. For example, some points in the plot below have an unusual combination of `x` and `y` values, which makes the points outliers even though their `x` and `y` values appear normal when examined separately.

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = x, y = y)) +
  coord_cartesian(xlim = c(4, 11), ylim = c(4, 11))
```



Why is a scatterplot a better display than a binned plot for this case?

In this case, there is a strong relationship between x and y. The outliers in this case are not extreme in either x or y. A binned plot would not reveal these outliers, and may lead us to conclude that the largest value of x was an outlier even though it appears to fit the bivariate pattern well.

The later chapter Model Basics discusses fitting models to bivariate data and plotting residuals, which would reveal this outliers.

7.6 Patterns and models

No exercises

7.7 ggplot2 calls

No exercises

7.8 Learning more

No exercises.

Chapter 8

Workflow: Projects

No exercises in this chapter.

Part II

Wrangle

Chapter 9

Tibbles

9.1 Introduction

```
library("tidyverse")
```

9.2 Creating Tibbles

No exercises

9.3 Tibbles vs. data.frame

No exercises

9.4 Subsetting

No exercises

9.5 Interacting with older code

No exercises

9.6 Exercises

9.6.1 Exercise 1

How can you tell if an object is a tibble? (Hint: try printing `mtcars`, which is a regular data frame).

```
mtcars
#>                      mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> Mazda RX4           21.0   6 160.0 110 3.90 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.88 17.0  0  1    4    4
#> Datsun 710          22.8   4 108.0  93 3.85 2.32 18.6  1  1    4    1
#> Hornet 4 Drive      21.4   6 258.0 110 3.08 3.21 19.4  1  0    3    1
#> Hornet Sportabout   18.7   8 360.0 175 3.15 3.44 17.0  0  0    3    2
#> Valiant             18.1   6 225.0 105 2.76 3.46 20.2  1  0    3    1
#> Duster 360          14.3   8 360.0 245 3.21 3.57 15.8  0  0    3    4
#> Merc 240D           24.4   4 146.7  62 3.69 3.19 20.0  1  0    4    2
#> Merc 230            22.8   4 140.8  95 3.92 3.15 22.9  1  0    4    2
#> Merc 280            19.2   6 167.6 123 3.92 3.44 18.3  1  0    4    4
#> Merc 280C           17.8   6 167.6 123 3.92 3.44 18.9  1  0    4    4
#> Merc 450SE          16.4   8 275.8 180 3.07 4.07 17.4  0  0    3    3
#> Merc 450SL          17.3   8 275.8 180 3.07 3.73 17.6  0  0    3    3
#> Merc 450SLC         15.2   8 275.8 180 3.07 3.78 18.0  0  0    3    3
#> Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.25 18.0  0  0    3    4
#> Lincoln Continental 10.4   8 460.0 215 3.00 5.42 17.8  0  0    3    4
#> Chrysler Imperial   14.7   8 440.0 230 3.23 5.34 17.4  0  0    3    4
#> Fiat 128            32.4   4  78.7  66 4.08 2.20 19.5  1  1    4    1
#> Honda Civic         30.4   4  75.7  52 4.93 1.61 18.5  1  1    4    2
#> Toyota Corolla      33.9   4  71.1  65 4.22 1.83 19.9  1  1    4    1
#> Toyota Corona       21.5   4 120.1  97 3.70 2.46 20.0  1  0    3    1
#> Dodge Challenger    15.5   8 318.0 150 2.76 3.52 16.9  0  0    3    2
#> AMC Javelin         15.2   8 304.0 150 3.15 3.44 17.3  0  0    3    2
#> Camaro Z28          13.3   8 350.0 245 3.73 3.84 15.4  0  0    3    4
#> Pontiac Firebird    19.2   8 400.0 175 3.08 3.85 17.1  0  0    3    2
#> Fiat X1-9           27.3   4  79.0  66 4.08 1.94 18.9  1  1    4    1
#> Porsche 914-2       26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> Lotus Europa        30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
#> Ford Pantera L      15.8   8 351.0 264 4.22 3.17 14.5  0  1    5    4
#> Ferrari Dino        19.7   6 145.0 175 3.62 2.77 15.5  0  1    5    6
#> Maserati Bora       15.0   8 301.0 335 3.54 3.57 14.6  0  1    5    8
#> Volvo 142E          21.4   4 121.0 109 4.11 2.78 18.6  1  1    4    2
```

```
class(mtcars)
#> [1] "data.frame"
```

```
class(as_tibble(mtcars))
#> [1] "tbl_df"     "tbl"        "data.frame"
```

Tibbles will only print out a limited number of rows and show the class on top of each column. Additionally, tibbles have class `"tbl_df"` and `"tbl_"` in addition to `"data.frame"`.

9.6.2 Exercise 2

> Compare and contrast the following operations on a `data.frame` and equivalent tibble. What is different? Why might the default data frame behaviors cause you frustration?

```
df <- data.frame(abc = 1, xyz = "a")
df$x
#> [1] a
#> Levels: a
df[, "xyz"]
```

```
#> [1] a
#> Levels: a
df[, c("abc", "xyz")]
#>   abc xyz
#> 1   1   a
```

```
tbl <- as_tibble(df)
tbl$x
#> Warning: Unknown or uninitialised column: 'x'.
#> NULL
tbl[, "xyz"]
#> # A tibble: 1 x 1
#>   xyz
#>   <fct>
#> 1 a
tbl[, c("abc", "xyz")]
#> # A tibble: 1 x 2
#>     abc xyz
#>   <dbl> <fct>
#> 1  1.00 a
```

Using `$` a data.frame will partially complete the column. So even though we wrote `df$x` it returned `df$xyz`. This saves a few keystrokes, but can result in accidentally using a different variable than you thought you were using.

With data.frames, with `[` the type of object that is returned differs on the number of columns. If it is one column, it won't return a data.frame, but instead will return a vector. With more than one column, then it will return a data.frame. This is fine if you know what you are passing in, but suppose you did `df[ , vars]` where `vars` was a variable. Then you what that code does depends on `length(vars)` and you'd have to write code to account for those situations or risk bugs.

9.6.3 Exercise 3

> If you have the name of a variable stored in an object, e.g. `var <- "mpg"`, how can you extract the reference variable from a tibble?

You can use the double bracket, like `df[[var]]`. You cannot use the dollar sign, because `df$var` would look for a column named `var`.

9.6.4 Exercise 4

> Practice referring to non-syntactic names in the following data frame by:
>
> 1. Extracting the variable called 1.
> 2. Plotting a scatterplot of 1 vs 2.
> 3. Creating a new column called 3 which is 2 divided by 1.
> 4. Renaming the columns to one, two and three.

```
annoying <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

Extract the variable called 1:

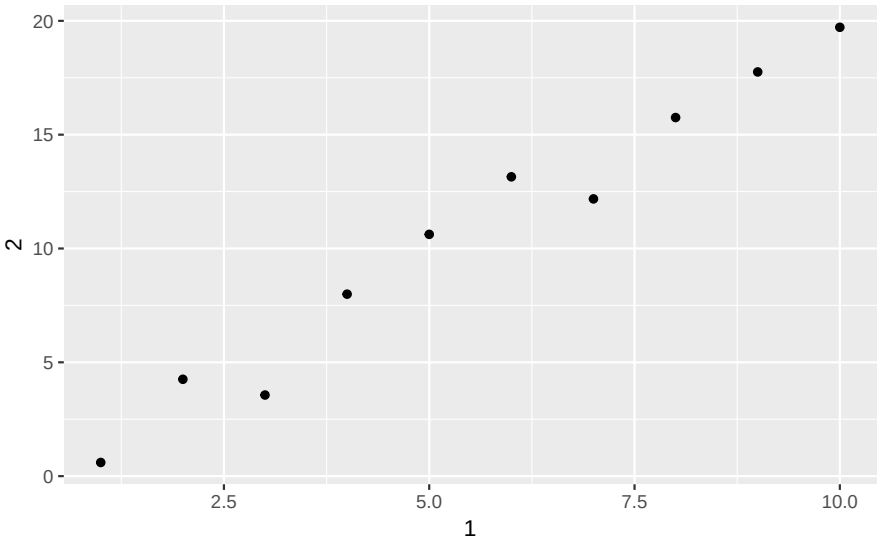
```
annoying[["1"]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

or

```
annoying$`1`
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

A scatter plot of 1 vs. 2:

```
ggplot(annoying, aes(x = `1`, y = `2`)) +
  geom_point()
```



A new column 3 with is 2 divided by 1:

```
annoying[["3"]] <- annoying$`2` / annoying$`1`
```

or

```
annoying[["3"]] <- annoying[["2"]] / annoying[["1"]]
```

Renaming the columns to one, two, and three:

```
annoying <- rename(annoying, one = `1`, two = `2`, three = `3`)
glimpse(annoying)
#> Observations: 10
#> Variables: 3
#> $ one   <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
#> $ two   <dbl> 0.60, 4.26, 3.56, 7.99, 10.62, 13.15, 12.18, 15.75, 17.7...
#> $ three <dbl> 0.60, 2.13, 1.19, 2.00, 2.12, 2.19, 1.74, 1.97, 1.97, 1.97
```

9.6.5 Exercise 5

What does `tibble::enframe()` do? When might you use it?

It converts named vectors to a data frame with names and values

```
?tibble::enframe
```

```
enframe(c(a = 1, b = 2, c = 3))
#> # A tibble: 3 x 2
```

```
#>   name  value
#>   <chr> <dbl>
#> 1 a      1.00
#> 2 b      2.00
#> 3 c      3.00
```

9.6.6 Exercise 6

What option controls how many additional column names are printed at the footer of a tibble?

The print function for tibbles is in `print.tbl_df`:

```
?print.tbl_df
```

The option `n_extra` determines the number of extra columns to print information for.

Chapter 10

Data Import

10.1 Introduction

```
library("tidyverse")
```

10.2 Getting started

10.2.1 Exercise 1

> What function would you use to read a file where fields were separated with "|"?

I'd use `read_delim` with `delim="|"`:

```
read_delim(file, delim = "|")
```

10.2.2 Exercise 2

> Apart from `file`, `skip`, and `comment`, what other arguments do `read_csv()` and `read_tsv()` have in common?

They have the following arguments in common:

```
union(names(formals(read_csv)), names(formals(read_tsv)))
#>  [1] "file"      "col_names" "col_types" "locale"    "na"
#>  [6] "quoted_na" "quote"     "comment"   "trim_ws"   "skip"
#> [11] "n_max"     "guess_max" "progress"
```

- `col_names` and `col_types` are used to specify the column names and how to parse the columns
- `locale` is important for determining things like the encoding and whether "." or "," is used as a decimal mark.
- `na` and `quoted_na` control which strings are treated as missing values when parsing vectors
- `trim_ws` trims whitespace before and after cells before parsing
- `n_max` sets how many rows to read
- `guess_max` sets how many rows to use when guessing the column type
- `progress` determines whether a progress bar is shown.

10.2.3 Exercise 3

> What are the most important arguments to `read_fwf()`?

The most important argument to `read_fwf` which reads "fixed-width formats", is `col_positions` which tells the function where data columns begin and end.

10.2.4 Exercise 4

> Sometimes strings in a CSV file contain commas. To prevent them from causing problems they need to be surrounded by a quoting character, like `"` or `'`. By convention, `read_csv()` assumes that the quoting character will be `"`, and if you want to change it you'll need to use `read_delim()` instead. What arguments do you need to specify to read the following text into a data frame?

`"x,y\n1,'a,b'"`

```
x <- "x,y\n1,'a,b'"
read_delim(x, ",", quote = "'")
#> # A tibble: 1 x 2
#>       x y
#>   <int> <chr>
#> 1     1 a,b
```

10.2.5 Exercise 6

> Identify what is wrong with each of the following inline CSV files. What happens when you run the code?

```
read_csv("a,b\n1,2,3\n4,5,6")
#> Warning: 2 parsing failures.
#> row # A tibble: 2 x 5 col     row col   expected  actual    file         expected   <int> <chr> <chr
#> # A tibble: 2 x 2
#>       a     b
#>   <int> <int>
#> 1     1     2
#> 2     4     5
```

Only two columns are specified in the header "a" and "b", but the rows have three columns, so the last column in dropped.

```
read_csv("a,b,c\n1,2\n1,2,3,4")
#> Warning: 2 parsing failures.
#> row # A tibble: 2 x 5 col     row col   expected  actual    file         expected   <int> <chr> <chr
#> # A tibble: 2 x 3
#>       a     b     c
#>   <int> <int> <int>
#> 1     1     2    NA
#> 2     1     2     3
```

The numbers of columns in the data do not match the number of columns in the header (three). In row one, there are only two values, so column `c` is set to missing. In row two, there is an extra value, and that value is dropped.

```
read_csv("a,b\n\"1")
#> Warning: 2 parsing failures.
#> row # A tibble: 2 x 5 col     row col   expected                    actual    file         expected
```

```
#> # A tibble: 1 x 2
#>       a b
#>   <int> <chr>
#> 1     1 <NA>
```

It's not clear what the intent was here. The opening quote `\\"1` is dropped because it is not closed, and `a` is treated as an integer.

```
read_csv("a,b\n1,2\na,b")
#> # A tibble: 2 x 2
#>   a     b
#>   <chr> <chr>
#> 1 1     2
#> 2 a     b
```

Both "a" and "b" are treated as character vectors since they contain non-numeric strings. This may have been intentional, or the author may have intended the values of the columns to be "1,2" and "a,b".

```
read_csv("a;b\n1;3")
#> # A tibble: 1 x 1
#>   `a;b`
#>   <chr>
#> 1 1;3
```

The values are separated by ";" rather than ",". Use `read_csv2` instead:

```
read_csv2("a;b\n1;3")
#> Using ',' as decimal and '.' as grouping mark. Use read_delim() for more control.
#> # A tibble: 1 x 2
#>       a     b
#>   <int> <int>
#> 1     1     3
```

10.3 Parsing a vector

10.3.1 Exercise 1

> What are the most important arguments to `locale()`?

The locale broadly controls the following:

- date and time formats: `date_names`, `date_format`, and `time_format`
- time_zone: `tz`
- numbers: `decimal_mark`, `grouping_mark`
- encoding: `encoding`

10.3.2 Exercise 2

> What happens if you try and set `decimal_mark` and `grouping_mark` to the same character? What happens to the default value of `grouping_mark` when you set `decimal_mark` to `","`? What happens to the default value of `decimal_mark` when you set the `grouping_mark` to `"."`?

If the decimal and grouping marks are set to the same character, `locale` throws an error:

```
locale(decimal_mark = ".", grouping_mark = ".")
#> Error: `decimal_mark` and `grouping_mark` must be different
```

If the `decimal_mark` is set to the comma "`,`", then the grouping mark is set to the period "`.`":

```
locale(decimal_mark = ",")
#> <locale>
#> Numbers:  123.456,78
#> Formats:  %AD / %AT
#> Timezone: UTC
#> Encoding: UTF-8
#> <date_names>
#> Days:   Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed),
#>         Thursday (Thu), Friday (Fri), Saturday (Sat)
#> Months: January (Jan), February (Feb), March (Mar), April (Apr), May
#>         (May), June (Jun), July (Jul), August (Aug), September
#>         (Sep), October (Oct), November (Nov), December (Dec)
#> AM/PM:  AM/PM
```

If the grouping mark is set to a period, then the decimal mark is set to a comma

```
locale(grouping_mark = ",")
#> <locale>
#> Numbers:  123,456.78
#> Formats:  %AD / %AT
#> Timezone: UTC
#> Encoding: UTF-8
#> <date_names>
#> Days:   Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed),
#>         Thursday (Thu), Friday (Fri), Saturday (Sat)
#> Months: January (Jan), February (Feb), March (Mar), April (Apr), May
#>         (May), June (Jun), July (Jul), August (Aug), September
#>         (Sep), October (Oct), November (Nov), December (Dec)
#> AM/PM:  AM/PM
```

10.3.3 Exercise 3

> I didn't discuss the `date_format` and `time_format` options to `locale()`. What do they do? Construct an example that shows when they might be useful.

They provide default date and time formats. The readr vignette discusses using these to parse dates: since dates can include languages specific weekday and month names, and different conventions for specifying AM/PM

```
locale()
#> <locale>
#> Numbers:  123,456.78
#> Formats:  %AD / %AT
#> Timezone: UTC
#> Encoding: UTF-8
#> <date_names>
#> Days:   Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed),
#>         Thursday (Thu), Friday (Fri), Saturday (Sat)
#> Months: January (Jan), February (Feb), March (Mar), April (Apr), May
#>         (May), June (Jun), July (Jul), August (Aug), September
#>         (Sep), October (Oct), November (Nov), December (Dec)
#> AM/PM:  AM/PM
```

Examples from the **readr** vignette of parsing French dates

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
#> [1] "2015-01-01"
parse_date("14 oct. 1979", "%d %b %Y", locale = locale("fr"))
#> [1] "1979-10-14"
```

Apparently the time format is not used for anything, but the date format is used for guessing column types.

10.3.4 Exercise 4

> If you live outside the US, create a new locale object that encapsulates the settings for the types of file you read most commonly.

```
?locale
```

10.3.5 Exercise 5

> What's the difference between `read_csv()` and `read_csv2()`?

The delimiter. The function `read_csv` uses a comma, while `read_csv2` uses a semi-colon (`;`). Using a semi-colon is useful when commas are used as the decimal point (as in Europe).

10.3.6 Exercise 6

> What are the most common encodings used in Europe? What are the most common encodings used in Asia? Do some googling to find out.

UTF-8 is standard now, and ASCII has been around forever.

For the European languages, there are separate encodings for Romance languages and Eastern European languages using Latin script, Cyrillic, Greek, Hebrew, Turkish: usually with separate ISO and Windows encoding standards. There is also Mac OS Roman.

For Asian languages Arabic and Vietnamese have ISO and Windows standards. The other major Asian scripts have their own:

- Japanese: JIS X 0208, Shift JIS, ISO-2022-JP
- Chinese: GB 2312, GBK, GB 18030
- Korean: KS X 1001, EUC-KR, ISO-2022-KR

The list in the documentation for `stringi::stri_enc_detect` is pretty good since it supports the most common encodings:

- Western European Latin script languages: ISO-8859-1, Windows-1250 (also CP-1250 for code-point)
- Eastern European Latin script languages: ISO-8859-2, Windows-1252
- Greek: ISO-8859-7
- Turkish: ISO-8859-9, Windows-1254
- Hebrew: ISO-8859-8, IBM424, Windows 1255
- Russian: Windows 1251
- Japanese: Shift JIS, ISO-2022-JP, EUC-JP
- Korean: ISO-2022-KR, EUC-KR
- Chinese: GB18030, ISO-2022-CN (Simplified), Big5 (Traditional)
- Arabic: ISO-8859-6, IBM420, Windows 1256

For more information on character encodings:

- The Wikipedia page Character encoding, has a good list of encodings.
- Unicode CLDR project

- What is the most common encoding of each language (Stack Overflow)
- "What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text", http://kunststube.net/encoding/.

Some program that identify the encoding of text are:

- In R see `readr::guess_encoding` and the **stringi** package with `str_enc_detect`
- iconv
- chardet (Python)

10.3.7 Exercise 7

Generate the correct format string to parse each of the following dates and times:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

The correct formats are:

```
parse_date(d1, "%B %d, %Y")
#> [1] "2010-01-01"
parse_date(d2, "%Y-%b-%d")
#> [1] "2015-03-07"
parse_date(d3, "%d-%b-%Y")
#> [1] "2017-06-06"
parse_date(d4, "%B %d (%Y)")
#> [1] "2015-08-19" "2015-07-01"
parse_date(d5, "%m/%d/%y")
#> [1] "2014-12-30"
parse_time(t1, "%H%M")
#> 17:05:00
```

The time `t2` uses real seconds,

```
parse_time(t2, "%H:%M:%OS %p")
#> 23:15:10.12
```

10.4 Parsing a file

No exercises

10.5 Writing to a file

No exercises

10.6 Other Types of Data

No code

Chapter 11

Tidy Data

11.1 Introduction

```
library(tidyverse)
```

11.2 Tidy Data

11.2.1 Exercise 1

> Using prose, describe how the variables and observations are organized in each of the sample tables.

In `table1` each row is a (country, year) with variables `cases` and `population`.

```
table1
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3 Brazil       1999  37737  172006362
#> 4 Brazil       2000  80488  174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

In `table2`, each row is country, year , variable ("cases", "population") combination, and there is a `count` variable with the numeric value of the variable.

```
table2
#> # A tibble: 12 x 4
#>   country      year type           count
#>   <chr>       <int> <chr>          <int>
#> 1 Afghanistan  1999 cases            745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000 cases           2666
#> 4 Afghanistan  2000 population  20595360
#> 5 Brazil       1999 cases          37737
```

```
#> 6 Brazil       1999 population 172006362
#> # ... with 6 more rows
```

In `table3`, each row is a (country, year) combination with the column `rate` having the rate of cases to population as a character string in the format `"cases/rate"`.

```
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
#> 4 Brazil       2000 80488/174504898
#> 5 China        1999 212258/1272915272
#> 6 China        2000 213766/1280428583
```

Table 4 is split into two tables, one table for each variable: `table4a` is the table for cases, while `table4b` is the table for population. Within each table, each row is a country, each column is a year, and the cells are the value of the variable for the table.

```
table4a
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
```

```
table4b
#> # A tibble: 3 x 3
#>   country         `1999`     `2000`
#> * <chr>            <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2 Brazil       172006362  174504898
#> 3 China       1272915272 1280428583
```

11.2.2 Exercise 2

Compute the `rate` for `table2`, and `table4a` + `table4b`. You will need to perform four operations:

1. Extract the number of TB cases per country per year.
2. Extract the matching population per country per year.
3. Divide cases by population, and multiply by 10000.
4. Store back in the appropriate place.

Which representation is easiest to work with? Which is hardest? Why?

Without using the join functions introduced in Ch 12:

```
tb2_cases <- filter(table2, type == "cases")[["count"]]
tb2_country <- filter(table2, type == "cases")[["country"]]
tb2_year <- filter(table2, type == "cases")[["year"]]
tb2_population <- filter(table2, type == "population")[["count"]]
table2_clean <- tibble(country = tb2_country,
       year = tb2_year,
       rate = tb2_cases / tb2_population)
```

```
table2_clean
#> # A tibble: 6 x 3
#>   country      year      rate
#>   <chr>       <int>     <dbl>
#> 1 Afghanistan  1999 0.0000373
#> 2 Afghanistan  2000 0.000129
#> 3 Brazil       1999 0.000219
#> 4 Brazil       2000 0.000461
#> 5 China        1999 0.000167
#> 6 China        2000 0.000167
```

Note, that this assumes that all observations are sorted so that each country, year will have the observation for cases followed by population.

```
tibble(country = table4a[["country"]],
       `1999` = table4a[["1999"]] / table4b[["1999"]],
       `2000` = table4b[["2000"]] / table4b[["2000"]])
#> # A tibble: 3 x 3
#>   country        `1999` `2000`
#>   <chr>           <dbl>  <dbl>
#> 1 Afghanistan 0.0000373   1.00
#> 2 Brazil      0.000219    1.00
#> 3 China       0.000167    1.00
```

or

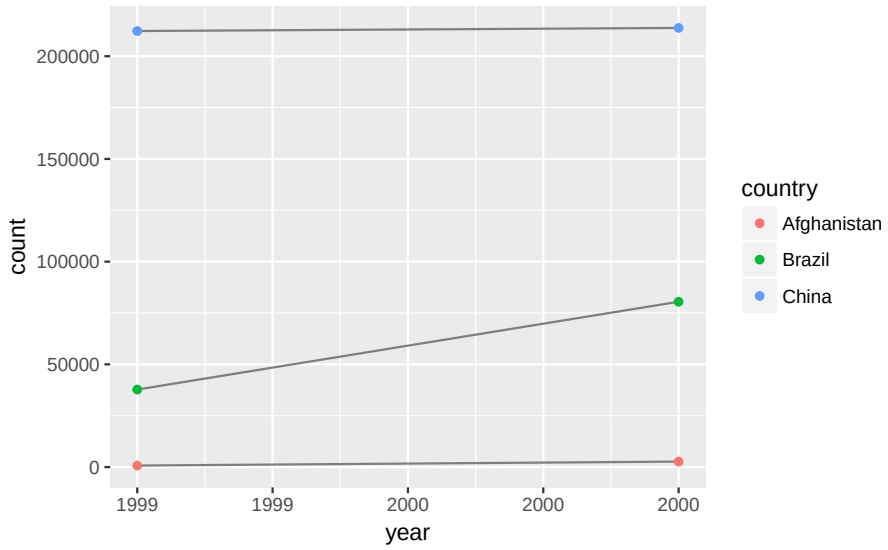
```
tibble(country = rep(table4a[["country"]], 2),
       year = rep(c(1999, 2000), each = nrow(table4a)),
       `rate` = c(table4a[["1999"]] / table4b[["1999"]],
                  table4b[["2000"]] / table4b[["2000"]]))
#> # A tibble: 6 x 3
#>   country      year      rate
#>   <chr>       <dbl>     <dbl>
#> 1 Afghanistan  1999 0.0000373
#> 2 Brazil       1999 0.000219
#> 3 China        1999 0.000167
#> 4 Afghanistan  2000 1.00
#> 5 Brazil       2000 1.00
#> 6 China        2000 1.00
```

11.2.3 Exercise 3

> Recreate the plot showing change in cases over time using `table2` instead of `table1`. What do you need to do first?

First, I needed to filter the tibble to only include those rows that represented the "cases" variable.

```
table2 %>%
  filter(type == "cases") %>%
  ggplot(aes(year, count)) +
  geom_line(aes(group = country), colour = "grey50") +
  geom_point(aes(colour = country))
```



11.3 Spreading and Gathering

This code is reproduced from the chapter because it is needed by the exercises:

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

11.3.1 Exercise 1

Why are `gather()` and `spread()` not perfectly symmetrical? Carefully consider the following example:

```
stocks <- tibble(
  year   = c(2015, 2015, 2016, 2016),
  half   = c(   1,    2,    1,    2),
  return = c(1.88, 0.59, 0.92, 0.17)
)
stocks %>%
  spread(year, return) %>%
  gather("year", "return", `2015`:`2016`)
#> # A tibble: 4 x 3
#>    half year  return
#>   <dbl> <chr>  <dbl>
#> 1  1.00 2015   1.88
#> 2  2.00 2015   0.590
#> 3  1.00 2016   0.920
#> 4  2.00 2016   0.170
```

The functions `spread` and `gather` are not perfectly symmetrical because column type information is not transferred between them. In the original table the column `year` was numeric, but after running `spread()` and `gather()` it is a character vector. This is because variable names are always converted to a character vector by `gather()`.

The `convert` argument tries to convert character vectors to the appropriate type. In the background this uses the `type.convert` function.

```
stocks %>%
  spread(year, return) %>%
  gather("year", "return", `2015`:`2016`, convert = TRUE)
#> # A tibble: 4 x 3
#>    half  year return
#>   <dbl> <int>  <dbl>
#> 1  1.00  2015  1.88
#> 2  2.00  2015  0.590
#> 3  1.00  2016  0.920
#> 4  2.00  2016  0.170
```

11.3.2 Exercise 2

Why does this code fail?

```
table4a %>%
  gather(1999, 2000, key = "year", value = "cases")
#> Error in inds_combine(.vars, ind_list): Position must be between 0 and n
```

The code fails because the column names `1999` and `2000` are not standard and thus needs to be quoted. The tidyverse functions will interpret `1999` and `2000` without quotes as looking for the 1999th and 2000th column of the data frame. This will work:

```
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
#> # A tibble: 6 x 3
#>   country     year   cases
#>   <chr>       <chr>  <int>
#> 1 Afghanistan 1999     745
#> 2 Brazil      1999   37737
#> 3 China       1999  212258
#> 4 Afghanistan 2000    2666
#> 5 Brazil      2000   80488
#> 6 China       2000  213766
```

11.3.3 Exercise 3

Why does spreading this tibble fail? How could you add a new column to fix the problem?

```
people <- tribble(
  ~name,             ~key,    ~value,
  #-----------------/--------/------
  "Phillip Woods",   "age",       45,
  "Phillip Woods",   "height",   186,
  "Phillip Woods",   "age",       50,
  "Jessica Cordero", "age",       37,
  "Jessica Cordero", "height",   156
)
glimpse(people)
#> Observations: 5
#> Variables: 3
```

```
#> $ name  <chr> "Phillip Woods", "Phillip Woods", "Phillip Woods", "Jess...
#> $ key   <chr> "age", "height", "age", "age", "height"
#> $ value <dbl> 45, 186, 50, 37, 156
```

```
spread(people, key, value)
#> Error: Duplicate identifiers for rows (1, 3)
```

Spreading the data frame fails because there are two rows with "age" for "Phillip Woods". We would need to add another column with an indicator for the number observation it is,

```
people <- tribble(
  ~name,             ~key,    ~value, ~obs,
  #-----------------/--------/------/------
  "Phillip Woods",   "age",       45, 1,
  "Phillip Woods",   "height",   186, 1,
  "Phillip Woods",   "age",       50, 2,
  "Jessica Cordero", "age",       37, 1,
  "Jessica Cordero", "height",   156, 1
)
spread(people, key, value)
#> # A tibble: 3 x 4
#>   name              obs   age height
#>   <chr>           <dbl> <dbl>  <dbl>
#> 1 Jessica Cordero  1.00  37.0    156
#> 2 Phillip Woods    1.00  45.0    186
#> 3 Phillip Woods    2.00  50.0     NA
```

11.3.4 Exercise 4

Tidy the simple tibble below. Do you need to spread or gather it? What are the variables?

```
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes",     NA,    10,
  "no",      20,    12
)
```

You need to gather it. The variables are:

- pregnant: logical ("yes", "no")
- female: logical
- count: integer

```
gather(preg, sex, count, male, female) %>%
  mutate(pregnant = pregnant == "yes",
         female = sex == "female") %>%
  select(-sex)
#> # A tibble: 4 x 3
#>   pregnant count female
#>   <lgl>    <dbl> <lgl>
#> 1 T         NA   F
#> 2 F         20.0 F
#> 3 T         10.0 T
#> 4 F         12.0 T
```

Converting `pregnant` and `female` from character vectors to logical was not necessary to tidy it, but it makes it easier to work with.

11.4 Separating and Uniting

11.4.1 Exercise 1

> What do the extra and fill arguments do in `separate()`? Experiment with the various options for the following two toy datasets.

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))
#> Warning: Expected 3 pieces. Additional pieces discarded in 1 rows [2].
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 d     e     f
#> 3 h     i     j

tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
#> Warning: Expected 3 pieces. Missing pieces filled with `NA` in 1 rows [2].
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 d     e     <NA>
#> 3 f     g     i
```

```
?separate
```

The `extra` argument tells `separate` what to do if there are too many pieces, and the `fill` argument if there aren't enough.

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))
#> Warning: Expected 3 pieces. Additional pieces discarded in 1 rows [2].
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 d     e     f
#> 3 h     i     j
```

By default `separate` drops the extra values with a warning.

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"), extra = "drop")
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 d     e     f
#> 3 h     i     j
```

This produces the same result as above, dropping extra values, but without the warning.

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"), extra = "merge")
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 d     e     f,g
#> 3 h     i     j
```

In this, the extra values are not split, so `"f,g"` appears in column three.

In this, one of the entries for column, `"d,e"`, has too few elements. The default for `fill` is similar to `separate`; it fills with missing values but emits a warning. In this, row 2 of column "three", is `NA`.

```
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
#> Warning: Expected 3 pieces. Missing pieces filled with `NA` in 1 rows [2].
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 d     e     <NA>
#> 3 f     g     i
```

Alternative options for `fill` are `"right"`, to fill with missing values from the right, but without a warning

```
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"), fill = "right")
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 d     e     <NA>
#> 3 f     g     i
```

The option `fill = "left"` also fills with missing values without a warning, but this time from the left side. Now, column "one" of row 2 will be missing, and the other values in that row are shifted over.

```
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"), fill = "left")
#> # A tibble: 3 x 3
#>   one   two   three
#>   <chr> <chr> <chr>
#> 1 a     b     c
#> 2 <NA>  d     e
#> 3 f     g     i
```

11.4.2 Exercise 2

> Both `unite()` and `separate()` have a remove argument. What does it do? Why would you set it to `FALSE`?

You would set it to `FALSE` if you want to create a new variable, but keep the old one.

11.4.3 Exercise 3

> Compare and contrast `separate()` and `extract()`, Why are there three variations of separation (by position, by separator, and with groups), but only one unite?

The function `extract` uses a regular expression to find groups and split into columns. In `unite` it is unambiguous since it is many columns to one, and once the columns are specified, there is only one way to do it, the only choice is the `sep`. In `separate`, it is one to many, and there are multiple ways to split the character string.

11.5 Missing Values

11.5.1 Exercise 1

> Compare and contrast the `fill` arguments to `spread()` and `complete()`.

```
?spread
```

```
?complete
```

In `spread`, the fill argument explicitly sets the value to replace `NA`s. In `complete`, the fill argument also sets a value to replace `NA`s but it is named list, allowing for different values for different variables. Also, both cases replace both implicit and explicit missing values.

11.5.2 Exercise 2

> What does the direction argument to `fill()` do?

With `fill`, it determines whether `NA` values should be replaced by the previous non-missing value (`"down"`) or the next non-missing value (`"up"`).

11.6 Case Study

This code is repeated from the chapter because it is needed by the exercises.

```
who1 <- who %>%
  gather(new_sp_m014:newrel_f65, key = "key", value = "cases", na.rm = TRUE)
glimpse(who1)
#> Observations: 76,046
#> Variables: 6
#> $ country <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanis...
#> $ iso2    <chr> "AF", "AF", "AF", "AF", "AF", "AF", "AF", "AF", "AF", ...
#> $ iso3    <chr> "AFG", "AFG", "AFG", "AFG", "AFG", "AFG", "AFG", "AFG"...
#> $ year    <int> 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, ...
#> $ key     <chr> "new_sp_m014", "new_sp_m014", "new_sp_m014", "new_sp_m...
#> $ cases   <int> 0, 30, 8, 52, 129, 90, 127, 139, 151, 193, 186, 187, 2...
```

```
who2 <- who1 %>%
 mutate(key = stringr::str_replace(key, "newrel", "new_rel"))
```

```
who3 <- who2 %>%
  separate(key, c("new", "type", "sexage"), sep = "_")
who3
#> # A tibble: 76,046 x 8
```

```
#>   country     iso2  iso3   year new   type  sexage cases
#>   <chr>       <chr> <chr> <int> <chr> <chr> <chr>  <int>
#> 1 Afghanistan AF    AFG    1997 new   sp    m014       0
#> 2 Afghanistan AF    AFG    1998 new   sp    m014      30
#> 3 Afghanistan AF    AFG    1999 new   sp    m014       8
#> 4 Afghanistan AF    AFG    2000 new   sp    m014      52
#> 5 Afghanistan AF    AFG    2001 new   sp    m014     129
#> 6 Afghanistan AF    AFG    2002 new   sp    m014      90
#> # ... with 7.604e+04 more rows
```

```
who3 %>%
  count(new)
#> # A tibble: 1 x 2
#>   new       n
#>   <chr> <int>
#> 1 new   76046
```

```
who4 <- who3 %>%
  select(-new, -iso2, -iso3)
```

```
who5 <- who4 %>%
  separate(sexage, c("sex", "age"), sep = 1)
who5
#> # A tibble: 76,046 x 6
#>   country      year type  sex   age   cases
#>   <chr>       <int> <chr> <chr> <chr> <int>
#> 1 Afghanistan  1997 sp    m     014       0
#> 2 Afghanistan  1998 sp    m     014      30
#> 3 Afghanistan  1999 sp    m     014       8
#> 4 Afghanistan  2000 sp    m     014      52
#> 5 Afghanistan  2001 sp    m     014     129
#> 6 Afghanistan  2002 sp    m     014      90
#> # ... with 7.604e+04 more rows
```

11.6.1 Exercise 1

> In this case study I set `na.rm = TRUE` just to make it easier to check that we had the correct values. Is this reasonable? Think about how missing values are represented in this dataset. Are there implicit missing values? What's the difference between an `NA` and zero?

Perhaps? I would need to know more about the data generation process. There are zero's in the data, which means they may explicitly be indicating no cases.

```
who1 %>%
  filter(cases == 0) %>%
  nrow()
#> [1] 11080
```

So it appears that either a country has all its values in a year as non-missing if the WHO collected data for that country, or all its values are non-missing. So it is okay to treat explicitly and implicitly missing values the same, and we don't lose any information by dropping them.

```
gather(who, new_sp_m014:newrel_f65, key = "key", value = "cases") %>%
  group_by(country, year)  %>%
  mutate(missing = is.na(cases)) %>%
```

```
  select(country, year, missing) %>%
  distinct() %>%
  group_by(country, year) %>%
  filter(n() > 1)
#> # A tibble: 6,968 x 3
#> # Groups:   country, year [3,484]
#>   country      year missing
#>   <chr>       <int> <lgl>
#> 1 Afghanistan  1997 F
#> 2 Afghanistan  1998 F
#> 3 Afghanistan  1999 F
#> 4 Afghanistan  2000 F
#> 5 Afghanistan  2001 F
#> 6 Afghanistan  2002 F
#> # ... with 6,962 more rows
```

11.6.2 Exercise 2

> What happens if you neglect the `mutate()` step? (`mutate(key = stringr::str_replace(key, "newrel", "new_rel")`)

`separate` emits the warning "too few values", and if we check the rows for keys beginning with `"newrel_"`, we see that `sexage` is messing, and `type = m014`.

```
who3a <- who1 %>%
  separate(key, c("new", "type", "sexage"), sep = "_")
#> Warning: Expected 3 pieces. Missing pieces filled with `NA` in 2580 rows
#> [73467, 73468, 73469, 73470, 73471, 73472, 73473, 73474, 73475, 73476,
#> 73477, 73478, 73479, 73480, 73481, 73482, 73483, 73484, 73485, 73486, ...].

filter(who3a, new == "newrel") %>% head()
#> # A tibble: 6 x 8
#>   country     iso2  iso3   year new    type  sexage cases
#>   <chr>       <chr> <chr> <int> <chr>  <chr> <chr>  <int>
#> 1 Afghanistan AF    AFG    2013 newrel m014  <NA>    1705
#> 2 Albania     AL    ALB    2013 newrel m014  <NA>      14
#> 3 Algeria     DZ    DZA    2013 newrel m014  <NA>      25
#> 4 Andorra     AD    AND    2013 newrel m014  <NA>       0
#> 5 Angola      AO    AGO    2013 newrel m014  <NA>     486
#> 6 Anguilla    AI    AIA    2013 newrel m014  <NA>       0
```

11.6.3 Exercise 3

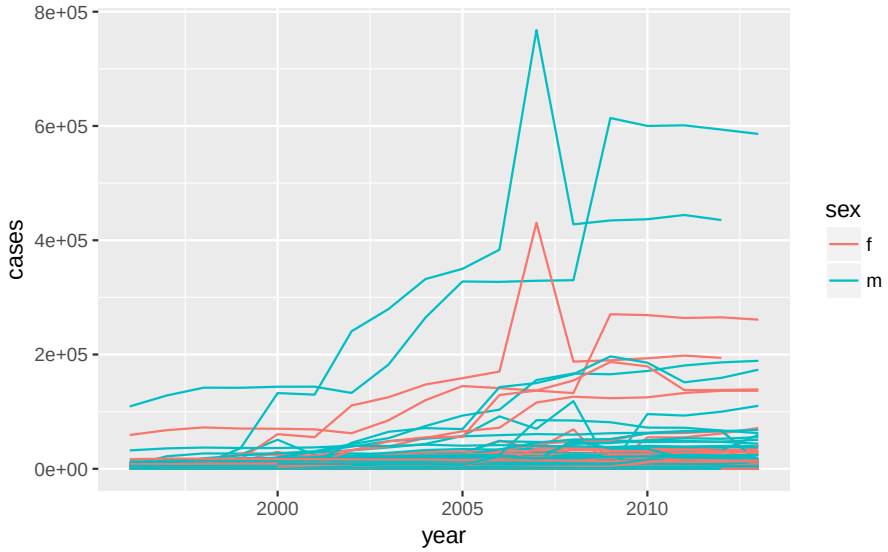
> I claimed that `iso2` and `iso3` were redundant with country. Confirm this claim.

```
select(who3, country, iso2, iso3) %>%
  distinct() %>%
  group_by(country) %>%
  filter(n() > 1)
#> # A tibble: 0 x 3
#> # Groups:   country [0]
#> # ... with 3 variables: country <chr>, iso2 <chr>, iso3 <chr>
```

11.6.4 Exercise 4

For each country, year, and sex compute the total number of cases of TB. Make an informative visualization of the data.

```
who5 %>%
  group_by(country, year, sex) %>%
  filter(year > 1995) %>%
  summarise(cases = sum(cases)) %>%
  unite(country_sex, country, sex, remove = FALSE) %>%
  ggplot(aes(x = year, y = cases, group = country_sex, colour = sex)) +
  geom_line()
```



A small multiples plot faceting by country is difficult given the number of countries. Focusing on those countries with the largest changes or absolute magnitudes after providing the context above is another option.

11.7 Non-Tidy Data

No exercises

Chapter 12

Relational Data

12.1 Introduction

```
library("tidyverse")
library("nycflights13")
```

The package datamodelr is used to draw database schema:

```
library("datamodelr")
```

12.2 nycflights13

12.2.1 Exercise 1

> Imagine you wanted to draw (approximately) the route each plane flies from its origin to its destination. What variables would you need? What tables would you need to combine?

- `flights` table: `origin` and `dest`
- `airports` table: longitude and latitude variables
- We would merge the `flights` with airports twice: once to get the location of the `origin` airport, and once to get the location of the `dest` airport.

12.2.2 Exercise 2

> I forgot to draw the relationship between weather and airports. What is the relationship and how should it appear in the diagram?

The variable `origin` in `weather` is matched with `faa` in `airports`.

12.2.3 Exercise 3

> weather only contains information for the origin (NYC) airports. If it contained weather records for all airports in the USA, what additional relation would it define with `flights`?

`year`, `month`, `day`, `hour`, `origin` in `weather` would be matched to `year`, `month`, `day`, `hour`, `dest` in `flight` (though it should use the arrival date-time values for `dest` if possible).

12.2.4 Exercise 4

> We know that some days of the year are "special", and fewer people than usual fly on them. How might you represent that data as a data frame? What would be the primary keys of that table? How would it connect to the existing tables?

I would add a table of special dates. The primary key would be date. It would match to the `year`, `month`, `day` columns of 'flights.

12.3 Keys

12.3.1 Exercise 1

> Add a surrogate key to flights.

I add the column `flight_id` as a surrogate key. I sort the data prior to making the key, even though it is not strictly necessary, so the order of the rows has some meaning.

```
flights %>%
  arrange(year, month, day, sched_dep_time, carrier, flight) %>%
  mutate(flight_id = row_number()) %>%
  glimpse()
#> Observations: 336,776
#> Variables: 20
#> $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013,...
#> $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ dep_time       <int> 517, 533, 542, 544, 554, 559, 558, 559, 558, 55...
#> $ sched_dep_time <int> 515, 529, 540, 545, 558, 559, 600, 600, 600, 60...
#> $ dep_delay      <dbl> 2, 4, 2, -1, -4, 0, -2, -1, -2, -2, -3, NA, 1, ...
#> $ arr_time       <int> 830, 850, 923, 1004, 740, 702, 753, 941, 849, 8...
#> $ sched_arr_time <int> 819, 830, 850, 1022, 728, 706, 745, 910, 851, 8...
#> $ arr_delay      <dbl> 11, 20, 33, -18, 12, -4, 8, 31, -2, -3, -8, NA,...
#> $ carrier        <chr> "UA", "UA", "AA", "B6", "UA", "B6", "AA", "AA",...
#> $ flight         <int> 1545, 1714, 1141, 725, 1696, 1806, 301, 707, 49...
#> $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N39463...
#> $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "EWR", "JFK", "LGA"...
#> $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ORD", "BOS", "ORD"...
#> $ air_time       <dbl> 227, 227, 160, 183, 150, 44, 138, 257, 149, 158...
#> $ distance       <dbl> 1400, 1416, 1089, 1576, 719, 187, 733, 1389, 10...
#> $ hour           <dbl> 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,...
#> $ minute         <dbl> 15, 29, 40, 45, 58, 59, 0, 0, 0, 0, 0, 0, 0, 0,...
#> $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013...
#> $ flight_id      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
```

12.3.2 Exercise 2

> Identify the keys in the following datasets

1. `Lahman::Batting`
2. `babynames::babynames`
3. `nasaweather::atmos`
4. `fueleconomy::vehicles`
5. `ggplot2::diamonds`

(You might need to install some packages and read some documentation.)

The primary key for `Lahman::Batting` is `playerID`, `yearID`, `stint`. It is not simply `playerID`, `yearID` because players can have different stints in different leagues within the same year.

```
Lahman::Batting %>%
  group_by(playerID, yearID, stint) %>%
  filter(n() > 1) %>%
  nrow()
#> [1] 0
```

The primary key for `babynames::babynames` is `year`, `sex`, `name`. It is no simply `year`, `name` since names can appear for both sexes with different counts.

```
babynames::babynames %>%
  group_by(year, sex, name) %>%
  filter(n() > 1) %>%
  nrow()
#> [1] 0
```

The primary key for `nasaweather::atmos` is the location and time of the measurement: `lat`, `long`, `year`, `month`.

```
nasaweather::atmos %>%
  group_by(lat, long, year, month) %>%
  filter(n() > 1) %>%
  nrow()
#> [1] 0
```

The column `id` (unique EPA identifier) is the primary key for `fueleconomy::vehicles`:

```
fueleconomy::vehicles %>%
  group_by(id) %>%
  filter(n() > 1) %>%
  nrow()
#> [1] 0
```

There is no primary key for `ggplot2::diamonds`. Using all variables in the data frame, the number of distinct rows is less than the total number of rows, meaning no combination of variables uniquely identifies the observations.

```
ggplot2::diamonds %>%
  distinct() %>%
  nrow()
#> [1] 53794
nrow(ggplot2::diamonds)
#> [1] 53940
```

12.3.3 Exercise 4

Draw a diagram illustrating the connections between the `Batting`, `Master`, and `Salaries` tables in the **Lahman** package. Draw another diagram that shows the relationship between `Master`, `Managers`, `AwardsManagers`.

Most flowchart or diagramming software can be used used to create database schema diagrams. For example, the diagrams in *R for Data Science* were created with Gliffy.

You can use anything to create these diagrams, but I'll use the R package datamodelr to programmatically create data models from R.

For the `Batting`, `Master`, and `Salaries` tables:

- `Master`
- Primary keys: `playerID`
- `Batting`
- Primary keys: `yearID`, `yearID`, `stint`
- Foreign Keys:
  - `playerID` = `Master$playerID` (many-to-1)
- `Salaries`:
- Primary keys: `yearID`, `teamID`, `playerID`
- Foreign Keys
  - `playerID` = `Master$playerID` (many-to-1)

```
dm1 <- dm_from_data_frames(list(Batting = Lahman::Batting,
                                Master = Lahman::Master,
                                Salaries = Lahman::Salaries)) %>%
  dm_set_key("Batting", c("playerID", "yearID", "stint")) %>%
  dm_set_key("Master", "playerID") %>%
  dm_set_key("Salaries", c("yearID", "teamID", "playerID")) %>%
  dm_add_references(
    Batting$playerID == Master$playerID,
    Salaries$playerID == Master$playerID
  )

dm_create_graph(dm1, rankdir = "LR", columnArrows = TRUE)
```

For the `Master`, `Manager`, and `AwardsManagers` tables:

- `Master`
- Primary keys: `playerID`
- `Managers`
- Primary keys: `yearID`, `teamID`, `inseason`
- Foreign Keys:
  - `playerID` = `Master$playerID` (many-to-1)
- `AwardsManagers`:
  - `playerID` = `Master$playerID` (many-to-1)

```
dm2 <- dm_from_data_frames(list(Master = Lahman::Master,
                                Managers = Lahman::Managers,
                                AwardsManagers = Lahman::AwardsManagers)) %>%
  dm_set_key("Master", "playerID") %>%
  dm_set_key("Managers", c("yearID", "teamID", "inseason")) %>%
  dm_set_key("AwardsManagers", c("playerID", "awardID", "yearID")) %>%
  dm_add_references(
    Managers$playerID == Master$playerID,
    AwardsManagers$playerID == Master$playerID
  )

dm_create_graph(dm2, rankdir = "LR", columnArrows = TRUE)
```

In the previous diagrams, I do not consider `teamID` and `lgID` as foreign keys even though they appear in multiple tables (and have the same meaning) because they are not primary keys in the tables considered in this exercise. The `teamID` variable references `Teams$teamID`, and `lgID` does not have its own table.

How would you characterize the relationship between the `Batting`, `Pitching`, and `Fielding` tables?

The `Batting`, `Pitching`, and `Fielding` tables all have a primary key consisting of the `playerID`, `yearID`, and `stint` variables. They all have a 1-1 relationship to each other.

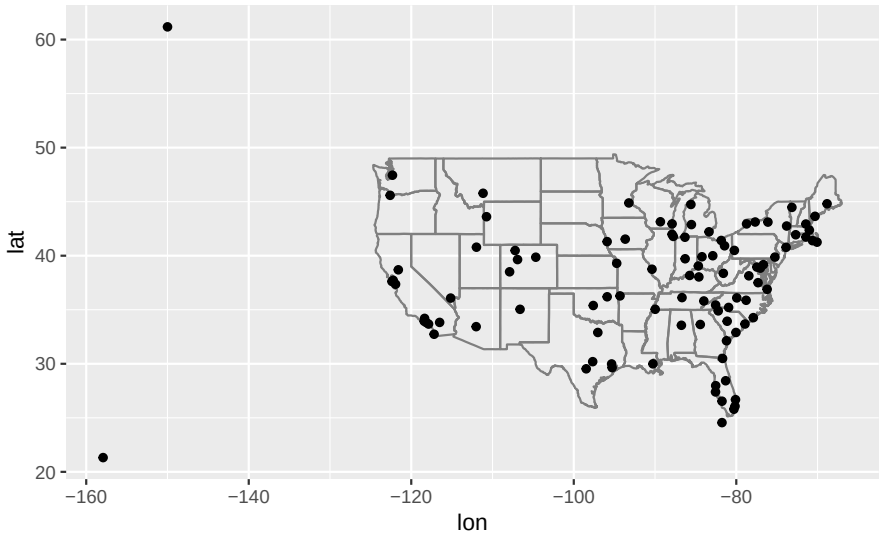
12.4 Mutating Joins

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
```

12.4.1 Exercise 1

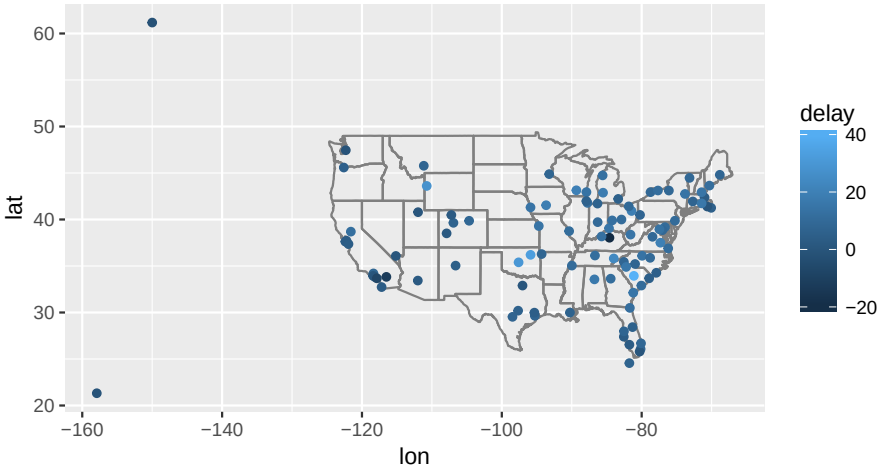
Compute the average delay by destination, then join on the `airports` data frame so you can show the spatial distribution of delays. Here's an easy way to draw a map of the United States:

```
airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  ggplot(aes(lon, lat)) +
    borders("state") +
    geom_point() +
    coord_quickmap()
#>
#> Attaching package: 'maps'
#> The following object is masked from 'package:purrr':
#>
#>     map
```



(Don't worry if you don't understand what `semi_join()` does — you'll learn about it next.)

```
avg_dest_delays <-
  flights %>%
  group_by(dest) %>%
  # arrival delay NA's are cancelled flights
  summarise(delay = mean(arr_delay, na.rm = TRUE)) %>%
  inner_join(airports, by = c(dest = "faa"))

avg_dest_delays %>%
  ggplot(aes(lon, lat, colour = delay)) +
    borders("state") +
    geom_point() +
    coord_quickmap()
```



You might want to use the size or color of the points to display the average delay for each airport.

12.4.2 Exercise 2

Add the location of the origin and destination (i.e. the `lat` and `lon`) to `flights`.

```
flights %>%
  left_join(airports, by = c(dest = "faa")) %>%
  left_join(airports, by = c(origin = "faa")) %>%
  head()
#> # A tibble: 6 x 33
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      542            540      2.00      923
#> 4  2013     1     1      544            545     -1.00     1004
#> 5  2013     1     1      554            600     -6.00      812
#> 6  2013     1     1      554            558     -4.00      740
#> # ... with 26 more variables: sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#> #   time_hour <dttm>, name.x <chr>, lat.x <dbl>, lon.x <dbl>, alt.x <int>,
#> #   tz.x <dbl>, dst.x <chr>, tzone.x <chr>, name.y <chr>, lat.y <dbl>,
#> #   lon.y <dbl>, alt.y <int>, tz.y <dbl>, dst.y <chr>, tzone.y <chr>
```

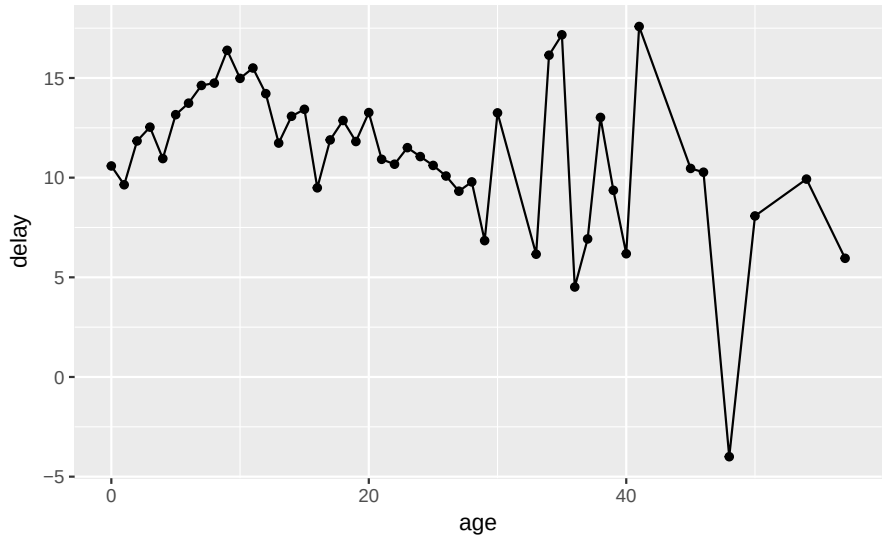
12.4.3 Exercise 3

Is there a relationship between the age of a plane and its delays?

Surprisingly not. If anything (departure) delay seems to decrease slightly with the age of the plane. This could be due to choices about how airlines allocate planes to airports.

```
plane_ages <-
  planes %>%
  mutate(age = 2013 - year) %>%
  select(tailnum, age)

flights %>%
```

```
  inner_join(plane_ages, by = "tailnum") %>%
  group_by(age) %>%
  filter(!is.na(dep_delay)) %>%
  summarise(delay = mean(dep_delay)) %>%
  ggplot(aes(x = age, y = delay)) +
  geom_point() +
  geom_line()
#> Warning: Removed 1 rows containing missing values (geom_point).
#> Warning: Removed 1 rows containing missing values (geom_path).
```

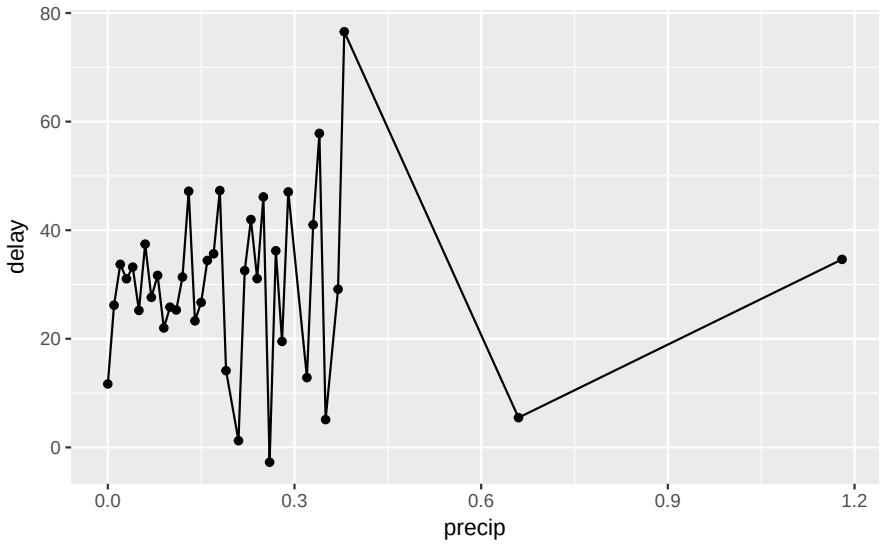


12.4.4 Exercise 4

> What weather conditions make it more likely to see a delay?

Almost any amount or precipitation is associated with a delay, though not as strong a trend after 0.02 in as one would expect

```
flight_weather <-
  flights %>%
  inner_join(weather, by = c("origin" = "origin",
                             "year" = "year",
                             "month" = "month",
                             "day" = "day",
                             "hour" = "hour"))

flight_weather %>%
  group_by(precip) %>%
  summarise(delay = mean(dep_delay, na.rm = TRUE)) %>%
  ggplot(aes(x = precip, y = delay)) +
    geom_line() + geom_point()
```



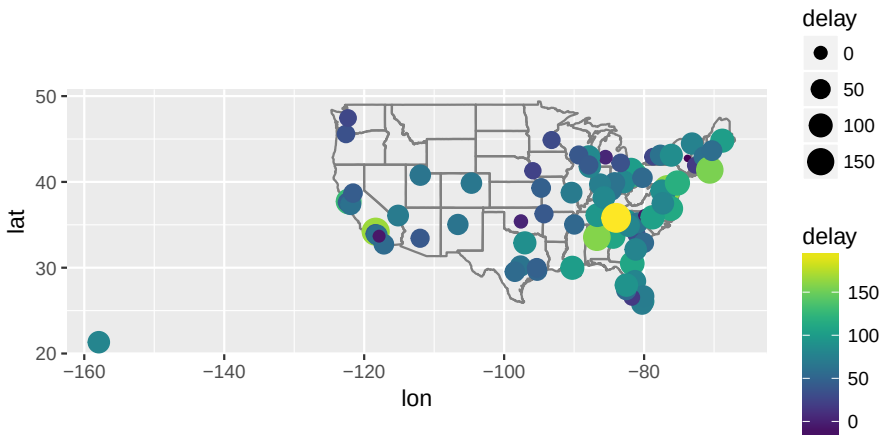
12.4.5 Exercise 5

> What happened on June 13 2013? Display the spatial pattern of delays, and then use Google to cross-reference with the weather.

There was a large series of storms (derechos) in the southeastern US (see June 12-13, 2013 derecho series)

The largest delays are in Tennessee (Nashville), the Southeast, and the Midwest, which were the locations of the derechos:

```
library(viridis)
#> Loading required package: viridisLite
flights %>%
  filter(year == 2013, month == 6, day == 13) %>%
  group_by(dest) %>%
  summarise(delay = mean(arr_delay, na.rm = TRUE)) %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
  ggplot(aes(y = lat, x = lon, size = delay, colour = delay)) +
  borders("state") +
  geom_point() +
  coord_quickmap() +
  scale_color_viridis()
#> Warning: Removed 3 rows containing missing values (geom_point).
```



12.5 Filtering Joins

12.5.1 Exercise 1

> What does it mean for a flight to have a missing `tailnum`? What do the tail numbers that don't have a matching record in planes have in common? (Hint: one variable explains ~90% of the problems.)

American Airlines (AA) and Envoy Airlines (MQ) don't report tail numbers.

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(carrier, sort = TRUE)
#> # A tibble: 10 x 2
#>   carrier     n
#>   <chr>   <int>
#> 1 MQ      25397
#> 2 AA      22558
#> 3 UA       1693
#> 4 9E       1044
#> 5 B6        830
#> 6 US        699
#> # ... with 4 more rows
```

12.5.2 Exercise 2

> Filter flights to only show flights with planes that have flown at least 100 flights.

```
planes_gt100 <-
  filter(flights) %>%
  group_by(tailnum) %>%
  count() %>%
  filter(n > 100)

flights %>%
  semi_join(planes_gt100, by = "tailnum")
#> # A tibble: 229,202 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515      2.00      830
#> 2  2013     1     1      533            529      4.00      850
#> 3  2013     1     1      544            545     -1.00     1004
#> 4  2013     1     1      554            558     -4.00      740
#> 5  2013     1     1      555            600     -5.00      913
#> 6  2013     1     1      557            600     -3.00      709
#> # ... with 2.292e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

12.5.3 Exercise 3

> Combine `fueleconomy::vehicles` and `fueleconomy::common` to find only the records for the most common models.

The table `fueleconomy::common` identifies vehicles by `make` and `model`:

```
glimpse(fueleconomy::vehicles)
#> Observations: 33,442
#> Variables: 12
#> $ id    <int> 27550, 28426, 27549, 28425, 1032, 1033, 3347, 13309, 133...
#> $ make  <chr> "AM General", "AM General", "AM General", "AM General", ...
#> $ model <chr> "DJ Po Vehicle 2WD", "DJ Po Vehicle 2WD", "FJ8c Post Off...
#> $ year  <int> 1984, 1984, 1984, 1984, 1985, 1985, 1987, 1997, 1997, 19...
#> $ class <chr> "Special Purpose Vehicle 2WD", "Special Purpose Vehicle ...
#> $ trans <chr> "Automatic 3-spd", "Automatic 3-spd", "Automatic 3-spd",...
#> $ drive <chr> "2-Wheel Drive", "2-Wheel Drive", "2-Wheel Drive", "2-Wh...
#> $ cyl   <int> 4, 4, 6, 6, 4, 6, 6, 4, 4, 6, 4, 4, 6, 4, 4, 6, 5, 5, 6,...
#> $ displ <dbl> 2.5, 2.5, 4.2, 4.2, 2.5, 4.2, 3.8, 2.2, 2.2, 3.0, 2.3, 2...
#> $ fuel  <chr> "Regular", "Regular", "Regular", "Regular", "Regular", "...
#> $ hwy   <int> 17, 17, 13, 13, 17, 13, 21, 26, 28, 26, 27, 29, 26, 27, ...
#> $ cty   <int> 18, 18, 13, 13, 16, 13, 14, 20, 22, 18, 19, 21, 17, 20, ...
glimpse(fueleconomy::common)
#> Observations: 347
#> Variables: 4
#> $ make  <chr> "Acura", "Acura", "Acura", "Acura", "Acura", "Audi", "Au...
#> $ model <chr> "Integra", "Legend", "MDX 4WD", "NSX", "TSX", "A4", "A4 ...
#> $ n     <int> 42, 28, 12, 28, 27, 49, 49, 66, 20, 12, 46, 20, 30, 29, ...
#> $ years <int> 16, 10, 12, 14, 11, 19, 15, 19, 19, 12, 20, 15, 16, 16, ...
```

```
fueleconomy::vehicles %>%
  semi_join(fueleconomy::common, by = c("make", "model"))
#> # A tibble: 14,531 x 12
#>       id make  model   year class trans drive   cyl displ fuel    hwy   cty
#>    <int> <chr> <chr>  <int> <chr> <chr> <chr> <int> <dbl> <chr> <int> <int>
#> 1  1833 Acura Integ~  1986 Subc~ Auto~ Fron~     4  1.60 Regu~    28    22
#> 2  1834 Acura Integ~  1986 Subc~ Manu~ Fron~     4  1.60 Regu~    28    23
#> 3  3037 Acura Integ~  1987 Subc~ Auto~ Fron~     4  1.60 Regu~    28    22
#> 4  3038 Acura Integ~  1987 Subc~ Manu~ Fron~     4  1.60 Regu~    28    23
#> 5  4183 Acura Integ~  1988 Subc~ Auto~ Fron~     4  1.60 Regu~    27    22
#> 6  4184 Acura Integ~  1988 Subc~ Manu~ Fron~     4  1.60 Regu~    28    23
#> # ... with 1.452e+04 more rows
```

12.5.4 Exercise 3

> Find the 48 hours (over the course of the whole year) that have the worst delays. Cross-reference it with the weather data. Can you see any patterns?

```
flights %>%
  group_by(year, month, day) %>%
  summarise(total_24 = sum(dep_delay, na.rm = TRUE)+ sum(arr_delay, na.rm = TRUE)) %>%
  mutate(total_48 = total_24 + lag(total_24)) %>%
  arrange(desc(total_48))
#> # A tibble: 365 x 5
#> # Groups:   year, month [12]
```

```
#>    year month   day total_24 total_48
#>   <int> <int> <int>    <dbl>    <dbl>
#> 1  2013     7    23    80641   175419
#> 2  2013     3     8   135264   167530
#> 3  2013     6    25    80434   166649
#> 4  2013     8     9    72866   165287
#> 5  2013     6    28    81389   157910
#> 6  2013     7    10    97120   157396
#> # ... with 359 more rows
```

12.5.5 Exercise 4

> What does `anti_join(flights, airports, by = c("dest" = "faa"))` tell you? What does `anti_join(airports, flights, by = c("faa" = "dest"))` tell you?

`anti_join(flights, airports, by = c("dest" = "faa"))` are flights that go to an airport that is not in FAA list of destinations, likely foreign airports.

`anti_join(airports, flights, by = c("faa" = "dest"))` are US airports that don't have a flight in the data, meaning that there were no flights to that airport **from** New York in 2013.

12.5.6 Exercise 5

> You might expect that there's an implicit relationship between plane and airline, because each plane is flown by a single airline. Confirm or reject this hypothesis using the tools you've learned above.

There isn't such a relationship over the lifetime of an airplane since planes can be sold or leased and airlines can merge. It should be the case that an airplane is associated with only airline at a given time, though may However, even though that's a possibility, it doesn't necessarily mean that plane associated with more than one appear in this data. Let's check:

```
airplane_multi_carrier <-
  flights %>%
  group_by(tailnum, carrier) %>%
  count() %>%
  filter(n() > 1) %>%
  select(tailnum) %>%
  distinct()
#> Adding missing grouping variables: `carrier`
airplane_multi_carrier
#> # A tibble: 0 x 2
#> # Groups:   tailnum, carrier [0]
#> # ... with 2 variables: carrier <chr>, tailnum <chr>
```

There are 0 airplanes in this dataset that have had more than one carrier. Even if there were none, the substantive reasons why an airplane *could* have more than one carrier would hold.

It is quite possible that we could have looked at the data, seen that each airplane only has one carrier, not thought much about it, and proceeded with some analysis that implicitly or explicitly relies on that one-to-one relationship. Then we apply our analysis to a larger set of data where that one-to-one relationship no longer holds, and it breaks. There is rarely a substitute for understanding the data which you are using as an analyst.

12.6 Join problems

No exercises

12.7 Set operations

No exercises

Chapter 13

Strings

13.1 Introduction

```
library(tidyverse)
library(stringr)
```

13.2 String Basics

13.2.1 Exercise 1

> In code that doesn't use **stringr**, you'll often see `paste()` and `paste0()`. What's the difference between the two functions? What **stringr** function are they equivalent to? How do the functions differ in their handling of NA?

The function `paste` separates strings by spaces by default, while `paste0` does not separate strings with spaces by default.

```
paste("foo", "bar")
#> [1] "foo bar"
paste0("foo", "bar")
#> [1] "foobar"
```

Since `str_c` does not separate strings with spaces by default it is closer in behavior to `paste0`.

```
str_c("foo", "bar")
#> [1] "foobar"
```

However, `str_c` and the paste function handle NA differently. The function `str_c` propagates `NA`, if any argument is a missing value, it returns a missing value. This is in line with how the numeric R functions, e.g. `sum`, `mean`, handle missing values. However, the paste functions, convert `NA` to the string `"NA"` and then treat it as any other character vector.

```
str_c("foo", NA)
#> [1] NA
paste("foo", NA)
#> [1] "foo NA"
paste0("foo", NA)
#> [1] "fooNA"
```

13.2.2 Exercise 2

> In your own words, describe the difference between the `sep` and `collapse` arguments to `str_c()`.

The `sep` argument is the string inserted between arguments to `str_c`, while `collapse` is the string used to separate any elements of the character vector into a character vector of length one.

13.2.3 Exercise 3

> Use `str_length()` and `str_sub()` to extract the middle character from a string. What will you do if the string has an even number of characters?

The following function extracts the middle character. If the string has an even number of characters the choice is arbitrary. We choose to select $\lceil n/2 \rceil$, because that case works even if the string is only of length one. A more general method would allow the user to select either the floor or ceiling for the middle character of an even string.

```
x <- c("a", "abc", "abcd", "abcde", "abcdef")
L <- str_length(x)
m <- ceiling(L / 2)
str_sub(x, m, m)
#> [1] "a" "b" "b" "c" "c"
```

13.2.4 Exercise 4

> What does `str_wrap()` do? When might you want to use it?

The function `str_wrap` wraps text so that it fits within a certain width. This is useful for wrapping long strings of text to be typeset.

13.2.5 Exercise 5

> What does `str_trim()` do? What's the opposite of `str_trim()`?

The function `str_trim` trims the whitespace from a string.

```
str_trim(" abc ")
#> [1] "abc"
str_trim(" abc ", side = "left")
#> [1] "abc "
str_trim(" abc ", side = "right")
#> [1] " abc"
```

The opposite of `str_trim` is `str_pad` which adds characters to each side.

```
str_pad("abc", 5, side = "both")
#> [1] " abc "
str_pad("abc", 4, side = "right")
#> [1] "abc "
str_pad("abc", 4, side = "left")
#> [1] " abc"
```

13.2.6 Exercise 6

> Write a function that turns (e.g.) a vector c("a", "b", "c") into the string a, b, and c. Think carefully about what it should do if given a vector of length 0, 1, or 2.

See the Chapter Functions for more details on writing R functions.

```
str_commasep <- function(x, sep = ", ", last = ", and ") {
  if (length(x) > 1) {
    str_c(str_c(x[-length(x)], collapse = sep),
                x[length(x)],
                sep = last)
  } else {
    x
  }
}
str_commasep("")
#> [1] ""
str_commasep("a")
#> [1] "a"
str_commasep(c("a", "b"))
#> [1] "a, and b"
str_commasep(c("a", "b", "c"))
#> [1] "a, b, and c"
```

13.3 Matching Patterns and Regular Expressions

13.3.1 Basic Matches

13.3.1.1 Exercise 1

> Explain why each of these strings don't match a \: "\", "\\", "\\\".

- "\": This will escape the next character in the R string.
- "\\": This will resolve to \ in the regular expression, which will escape the next character in the regular expression.
- "\\\": The first two backslashes will resolve to a literal backslash in the regular expression, the third will escape the next character. So in the regular expression, this will escape some escaped character.

13.3.1.2 Exercise 2

> How would you match the sequence "'\ ?

```
str_view("\"'\\", "\"'\\\\")
```

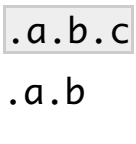
\begin{center} 

13.3.1.3 Exercise 3

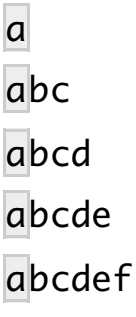
> What patterns will the regular expression \..\..\.. match? How would you represent it as a string?

It will match any patterns that are a dot followed by any character, repeated three times.

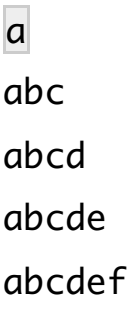
```
str_view(c(".a.b.c", ".a.b", "....."), c("\\..\\..\\.."))
```



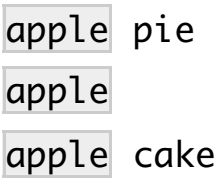

```
x < c("apple", "banana", "pear")
#> Warning in x < c("apple", "banana", "pear"): longer object length is not a
#> multiple of shorter object length
str_view(x, "^a")
```



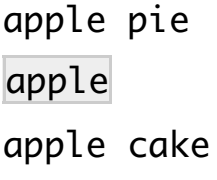

```
str_view(x, "a$")
```




```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```




```
str_view(x, "^apple$")
```

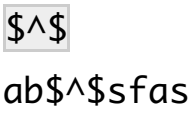


13.3.2 Anchors

13.3.2.1 Exercise 1

How would you match the literal string `"$^$"`?

```
str_view(c("$^$", "ab$^$sfas"), "^\\$\\^\\$$")
```



\begin{center}

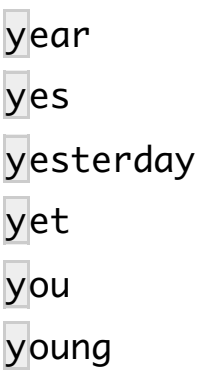
13.3.2.2 Exercise 2

Given the corpus of common words in `stringr::words`, create regular expressions that find all words that:

1. Start with "y".
2. End with "x"
3. Are exactly three letters long. (Don't cheat by using `str_length()`!)
4. Have seven letters or more.

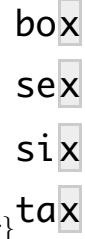
Since this list is long, you might want to use the match argument to `str_view()` to show only the matching or non-matching words.

```
str_view(stringr::words, "^y", match =TRUE)
```



\begin{center}

```
str_view(stringr::words, "x$", match = TRUE)
```



\begin{center}

```
str_view(stringr::words, "^...$", match = TRUE)
```

```
act
add
age
ago
air
all
and
any
arm
art
ask
bad
bag
bar
bed
bet
big
bit
box
boy
bus
but
buy
can
car
cat
cup
cut
dad
day
die
dog
dry
due
eat
egg
end
eye
far
few
fit
fly
for
fun
gas
get
god
guy
hit
hot
how
job
key
kid
lad
law
lay
leg
let
lie
lot
low
man
may
mrs
new
non
not
now
odd
off
old
one
out
own
pay
per
put
red
rid
run
say
see
set
sex
she
sir
sit
six
son
sun
tax
tea
ten
the
tie
too
top
try
two
use
war
way
wee
who
why
win
yes
yet
you
```

\begin{center}

A simpler way, shown later is

```
str_view(stringr::words, "^.{3}$", match = TRUE)
```

```
act
add
age
ago
air
all
and
any
arm
art
ask
bad
bag
bar
bed
bet
big
bit
box
boy
bus
but
buy
can
car
cat
cup
cut
dad
day
die
dog
dry
due
eat
egg
end
eye
far
few
fit
fly
for
fun
gas
get
god
guy
hit
hot
how
job
key
kid
lad
law
lay
leg
let
lie
lot
low
man
may
mrs
new
non
not
now
odd
off
old
one
out
own
pay
per
put
red
rid
run
say
see
set
sex
she
sir
sit
six
son
sun
tax
tea
ten
the
tie
too
top
try
two
use
war
way
wee
who
why
win
yes
yet
you
```

\begin{center}

```
str_view(stringr::words, ".......", match = TRUE)
```

```
absolute
account
achieve
address
advertise
afternoon
against
already
alright
although
america
another
approach
appoint
approach
appropriate
arrange
associate
authority
available
balance
because
believe
benefit
between
brilliant
britain
brother
business
certain
chairman
character
christmas
colleague
collect
college
comment
committee
community
company
compare
complete
computer
concern
condition
consider
consult
contact
continue
contract
control
converse
correct
council
country
current
decision
definite
department
describe
develop
difference
difficult
discuss
district
document
economy
educator
electric
encourage
english
environment
especial
evening
evidence
example
exercise
expense
experience
explain
express
finance
fortune
forward
function
further
general
germany
goodbye
history
holiday
hospital
however
hundred
husband
identify
imagine
important
improve
include
increase
individual
industry
instead
interest
introduce
involve
kitchen
language
machine
meaning
measure
mention
million
minister
morning
necessary
obtain
occasion
operate
opportunity
organize
original
otherwise
paragraph
particular
pension
percent
perfect
perhaps
photograph
picture
politic
position
positive
possible
practice
prepare
present
pressure
pressure
previous
private
probable
problem
proceed
process
produce
product
programme
project
propose
protect
provide
purpose
quality
quarter
question
realise
receive
recognize
recommend
relation
remember
represent
require
research
resource
respect
responsible
saturday
science
scotland
secretary
section
separate
serious
service
similar
situated
society
special
specific
standard
station
strength
strategy
structure
student
subject
succeed
suggest
support
suppose
surprise
telephone
television
terrible
therefore
thirteen
thousand
through
thursday
together
tomorrow
tonight
traffic
transport
trouble
tuesday
understand
university
various
village
wednesday
welcome
weather
without
yesterday
```

\begin{center}

13.3.3 Character classes and alternatives

13.3.3.1 Exercise 1

Create regular expressions to find all words that:

1. Start with a vowel.
2. That only contain consonants. (Hint: thinking about matching "not"-vowels.)
3. End with `ed`, but not with `eed`.
4. End with `ing` or `ise`.

Words starting with vowels

```
str_view(stringr::words, "^[aeiou]", match = TRUE)
```

```
a
able
about
absolute
accept
account
achieve
across
act
active
actual
add
address
admit
advertise
affect
afford
after
afternoon
again
against
age
agent
ago
agree
air
all
allow
almost
along
already
alright
also
although
always
america
amount
and
another
answer
any
apart
apparent
appear
apply
appoint
approach
appropriate
area
argue
arm
around
arrange
art
as
ask
associate
assume
at
attend
authority
available
aware
away
awful
each
early
east
easy
eat
economy
educate
effect
egg
eight
either
elect
electric
eleven
else
employ
encourage
end
engine
english
enjoy
enough
enter
environment
equal
especial
europe
even
evening
ever
every
evidence
exact
example
except
excuse
exercise
exist
expect
expense
experience
explain
express
extra
eye
idea
identify
if
imagine
important
improve
in
include
income
increase
indeed
individual
industry
inform
inside
instead
insure
interest
into
introduce
invest
involve
issue
it
item
obvious
occasion
odd
of
off
offer
office
often
okay
old
on
once
one
only
open
operate
opportunity
oppose
or
order
organize
original
other
otherwise
ought
out
over
own
under
understand
union
unit
unite
university
unless
until
up
upon
use
usual
```

\begin{center}

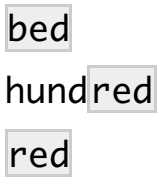
Words that contain only consonants

```
str_view(stringr::words, "^[^aeiou]+$", match=TRUE)
```

\begin{center} This seems to require using the `+` pattern introduced later, unless one wants to be very verbose and specify words of certain lengths.

Words that end with `ed` but not with `eed`. This handles the special case of "ed", as well as words with length > 2.

```
str_view(stringr::words, "^ed$|[^e]ed$", match = TRUE)
```



\begin{center}

Words ending in `ing` or `ise`:

```
str_view(stringr::words, "i(ng|se)$", match = TRUE)
```

advertise
bring
during
evening
exercise
king
meaning
morning
otherwise
practise
raise
realise
ring
rise
sing
surprise
thing

\begin{center}

13.3.3.2 Exercise 2

Empirically verify the rule “i before e except after c”.

Using only what has been introduced thus far:

```
str_view(stringr::words, "(cei|[^c]ie)", match = TRUE)
```

achieve
believe
brief
client
die
experience
field
friend
lie
piece
quiet
receive
tie
view

\begin{center}

```
str_view(stringr::words, "(cie|[^c]ei)", match = TRUE)
```

science
society
weigh

\begin{center}

Using `str_detect` we can count the number of words that follow these rules:

```
sum(str_detect(stringr::words, "(cei|[^c]ie)"))
sum(str_detect(stringr::words, "(cie|[^c]ei)"))
```

13.3.3.3 Exercise 3

> Is `q'' always followed by a`u"?

In the `stringr::words` dataset, yes. In the full English language, no.

```
str_view(stringr::words, "q[^u]", match = TRUE)
```

\begin{center}

13.3.3.4 Exercise 4

> Write a regular expression that matches a word if it's probably written in British English, not American English.

In the general case, this is hard, and could require a dictionary. But, there are a few heuristics to consider that would account for some common cases: British English tends to use the following:

- "ou" instead of "o"
- use of "ae" and "oe" instead of "a" and "o"
- ends in `ise` instead of `ize`
- ends in `yse`

The regex `ou|ise^|ae|oe|yse^` would match these.

There are other [spelling differences between American and British English] (https://en.wikipedia.org/wiki/American_and_British_English_spelling_differences) but they are not patterns amenable to regular expressions. It would require a dictionary with differences in spellings for different words.

13.3.3.5 Exercise 5

> Create a regular expression that will match telephone numbers as commonly written in your country.

The answer to this will vary by country.

For the United States, phone numbers have a format like `123-456-7890`.

```
x <- c("123-456-7890", "1235-2351")
str_view(x, "\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d")
```

```
[1] │ 123-456-7890
[2] │ 1235-2351
```

\begin{center} or

```
str_view(x, "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]")
```

123-456-7890

\begin{center}1235-2351

This regular expression can be simplified with the `{m,n}` regular expression modifier introduced in the next section,

```
str_view(x, "\\d{3}-\\d{3}-\\d{4}")
```

123-456-7890

\begin{center}1235-2351

Note that this pattern doesn't account for phone numbers that are invalid because of unassigned area code, or special numbers like 911, or extensions. See the Wikipedia page for the North American Numbering Plan for more information on the complexities of US phone numbers, and this Stack Overflow question for a discussion of using a regex for phone number validation.

13.3.4 Repetition

13.3.4.1 Exercise 1

Describe the equivalents of `?`, `+`, `*` in `{m,n}` form.

`-`	`{,1}`	Match at most 1
`+`	`{1,}`	Match one or more
`*`	None	No equivalent

The `*` pattern has no `{m,n}` equivalent since there is no upper bound on the number of matches. The expected pattern `{,}` is not valid.

```
str_view("abcd", ".{,}")
#> Error in stri_locate_first_regex(string, pattern, opts_regex = opts(pattern)): Error in {min,max} in
```

13.3.4.2 Exercise 2

Describe in words what these regular expressions match: (read carefully to see if I'm using a regular expression or a string that defines a regular expression.)

1. `^.*$`
2. `"\\{.+\\}"`
3. `\d{4}-\d{2}-\d{2}`
4. `"\\\\{4}"`

`^.*$` Any string `"\\{.+\\}"` Any string with curly braces surrounding at least one character. `\d{4}-\d{2}-\d{2}` Four digits followed by a hyphen, followed by two digits followed by a hyphen, followed by another two digits. This is a regular expression that can match dates formatted like "YYYY-MM-DD" ("%Y-%m-%d"). `"\\\\{4}"` This resolves to the regular expression `\\{4}`, which is four backslashes.

Examples:

- `^.*$`: `c("dog", "$1.23", "lorem ipsum")`