R for Data Science Solutions

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.

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
```

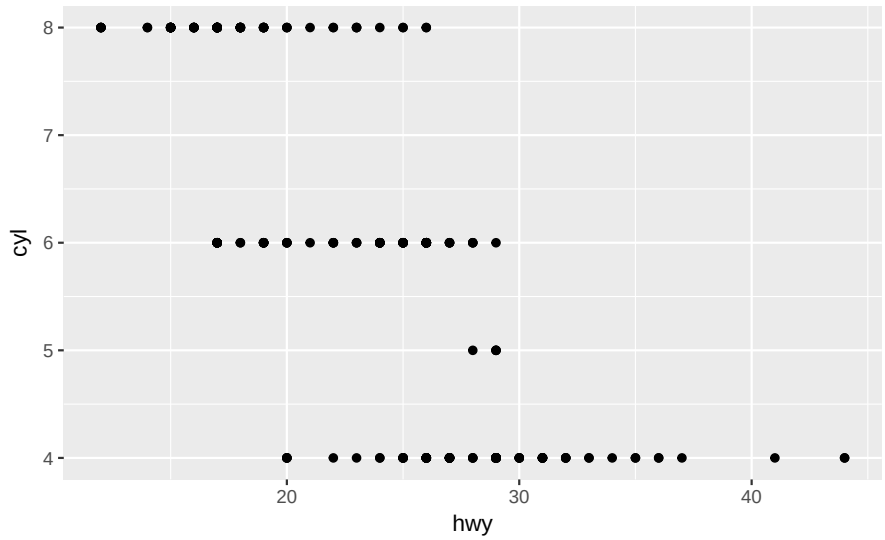
The `drv` variable takes the following values

- `"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`

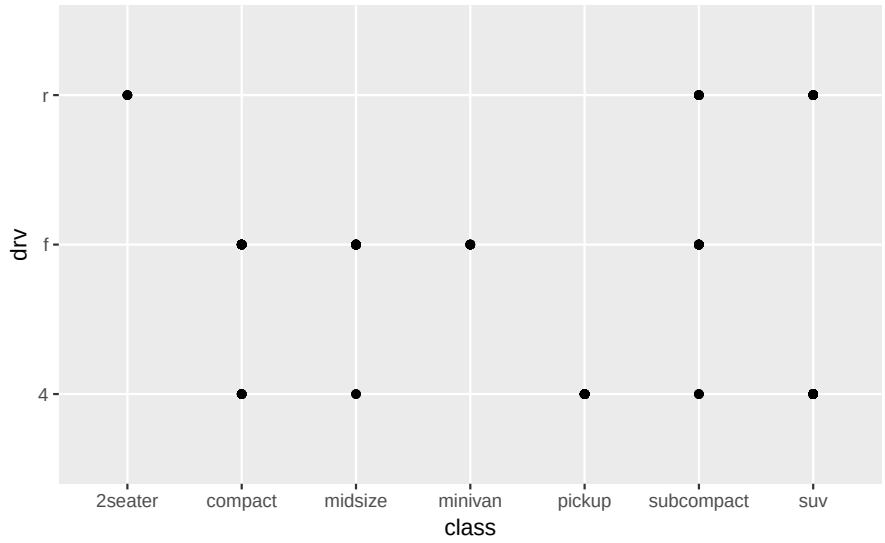
```
ggplot(mpg, aes(x = hwy, y = cyl)) +
  geom_point()
```



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
```

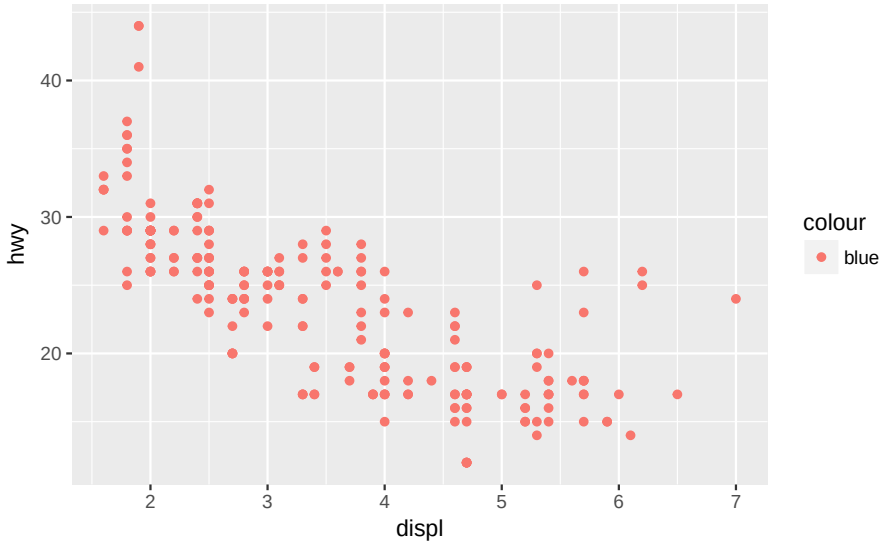
The scatter plot cannot show which are overlapping or not. Later chapters discuss means to deal with this, including alternative plots and jittering the points so they don't overlap.

3.3 Aesthetic mappings

3.3.1 Exercise 1.

::::question 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:

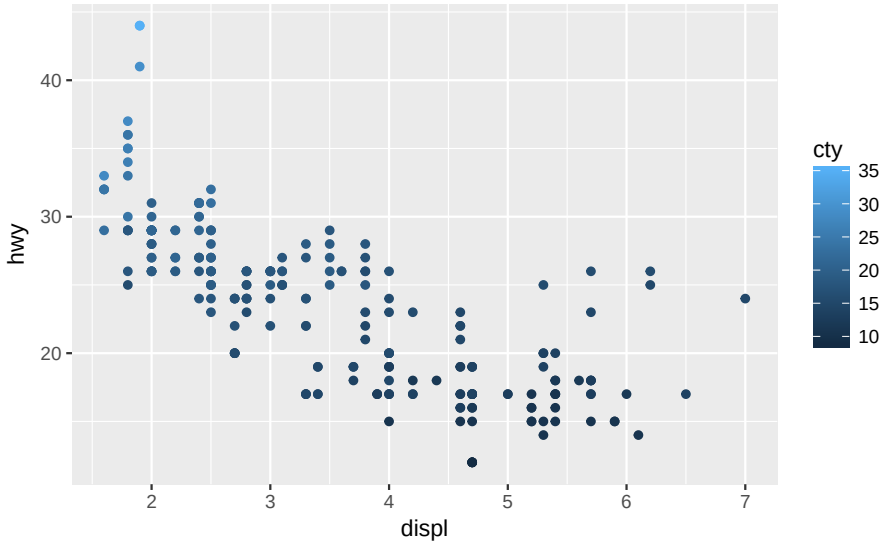
```
glimpse(mpg)
#> Observations: 234
#> Variables: 11
#> $ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "...
#> $ model        <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 qua...
#> $ displ        <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0,...
#> $ year         <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?

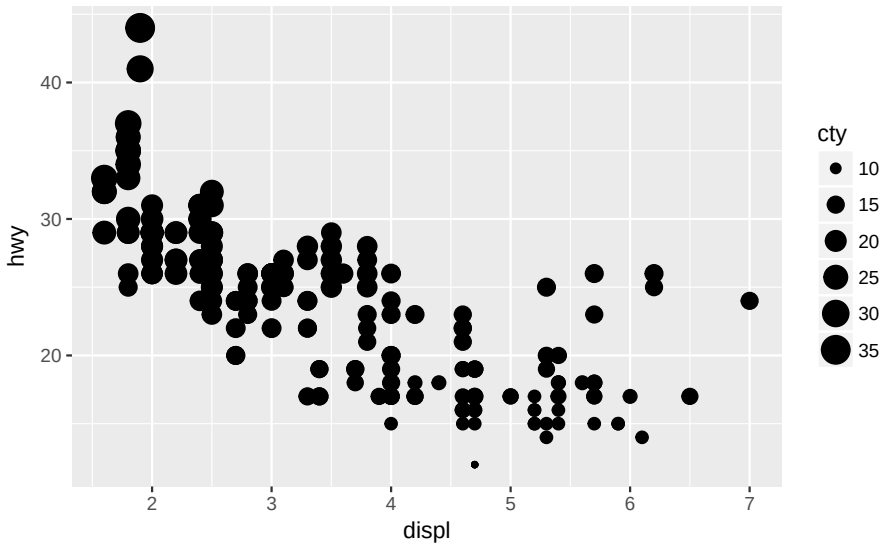
The variable `cty`, city highway miles per gallon, is a continuous variable:

```
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)

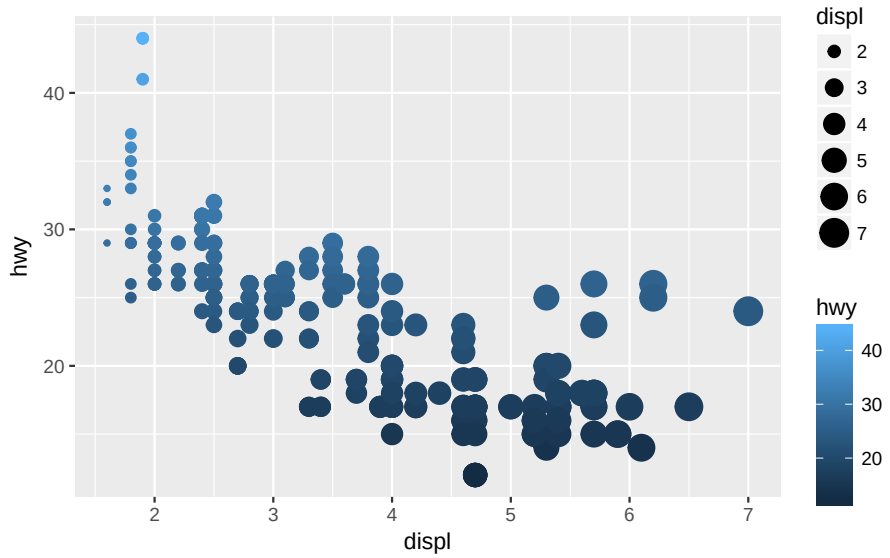
```
ggplot(mpg, aes(x = displ, y = hwy, shape = cty)) +
  geom_point()
#> Error: A continuous variable can not be mapped to shape
```

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()
```



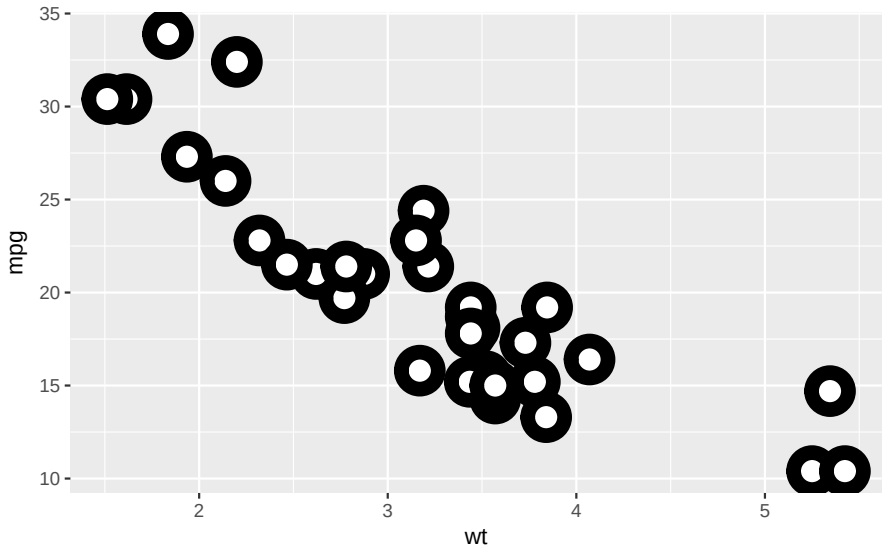
In the above plot, `hwy` is mapped to both location on the y-axis and color, and `displ` is mapped to both location on the x-axis and size. The code works and produces a plot, even if it is a bad one. Mapping a single variable to multiple aesthetics is redundant. Because it is redundant information,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

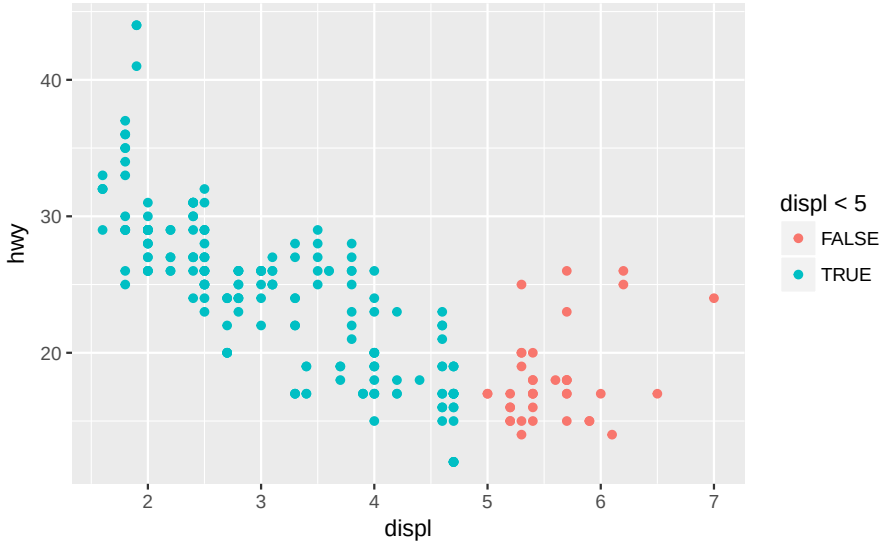


Stroke changes the color of the border for shapes (22-24).

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

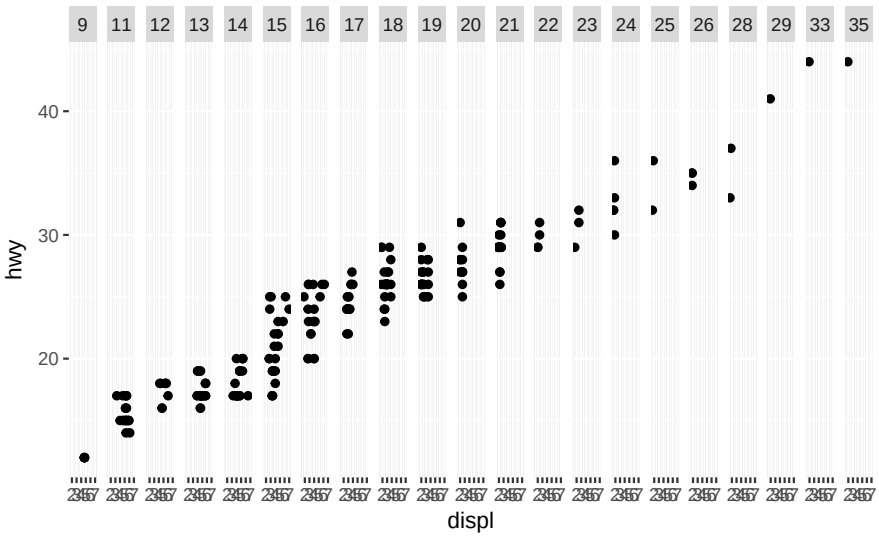
3.5 Facets

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)
```



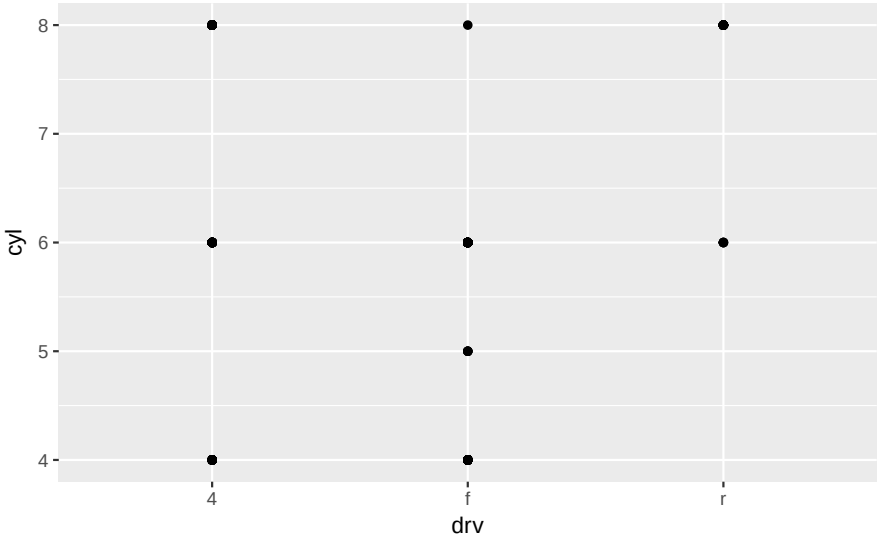
It converts the continuous variable to a factor and creates facets for **all** unique values of it.

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.

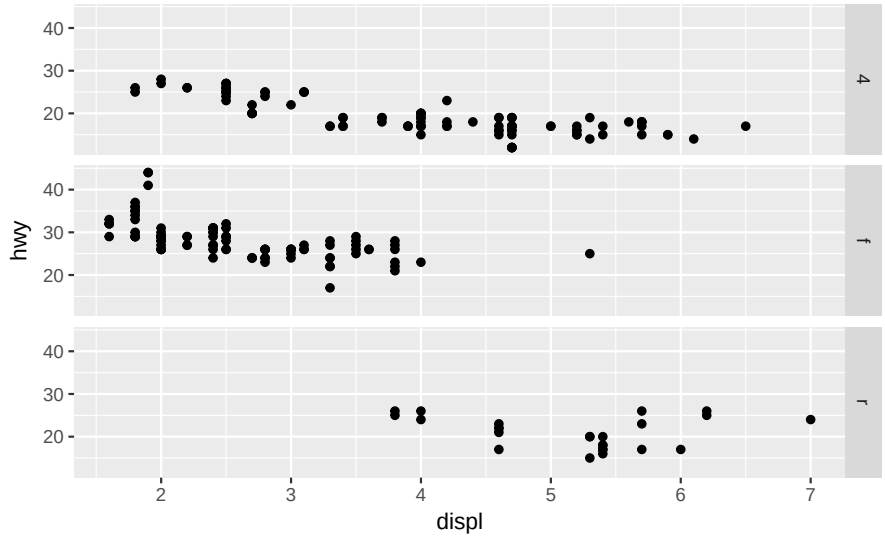
3.5.3 Exercise 3.

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

The symbol `.` ignores that dimension for faceting.

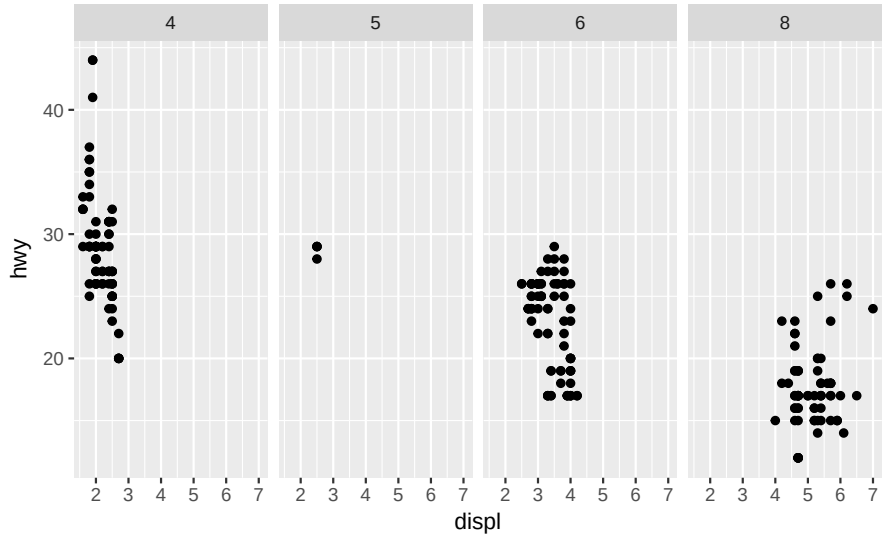
This plot facets by values of `drv` on the y-axis:

```
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`

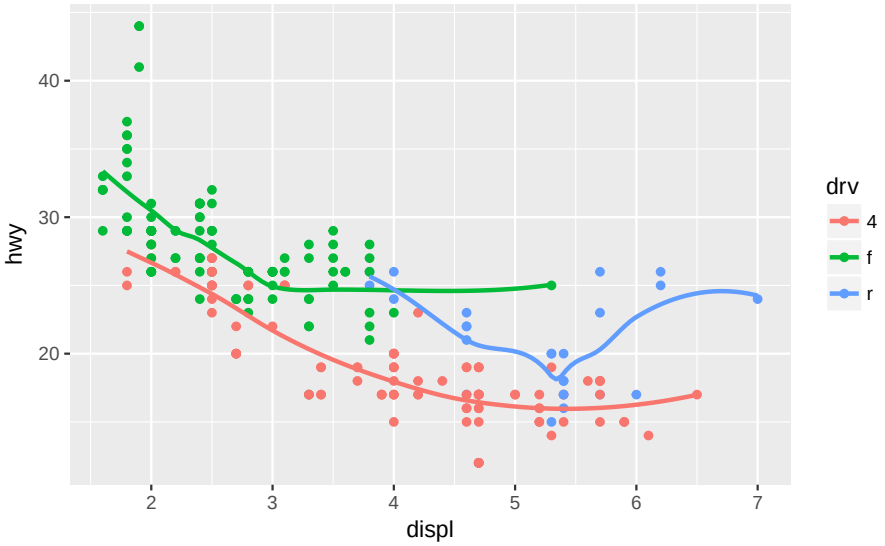
3.6.2 Exercise 2.

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'
```

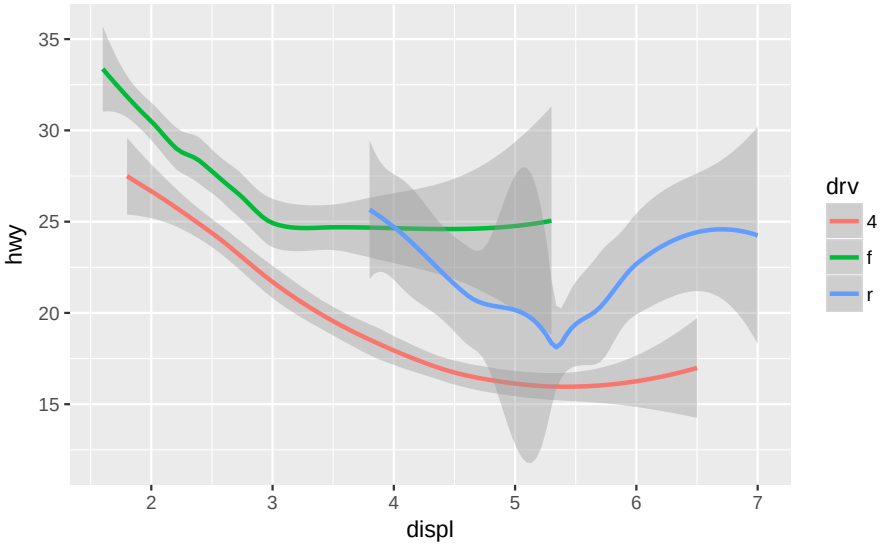


3.6.3 Exercise 3.

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

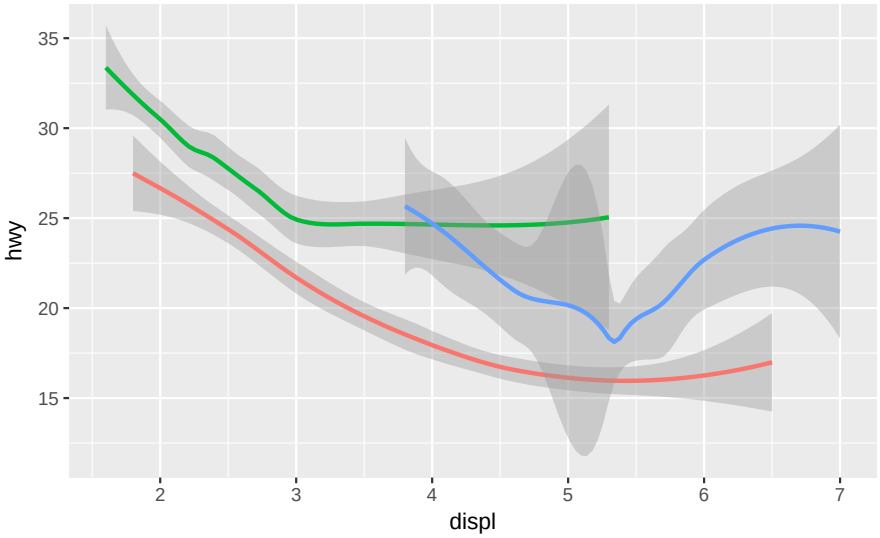
Show legend hides the legend box. In this code, without show legend, there is a legend.

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



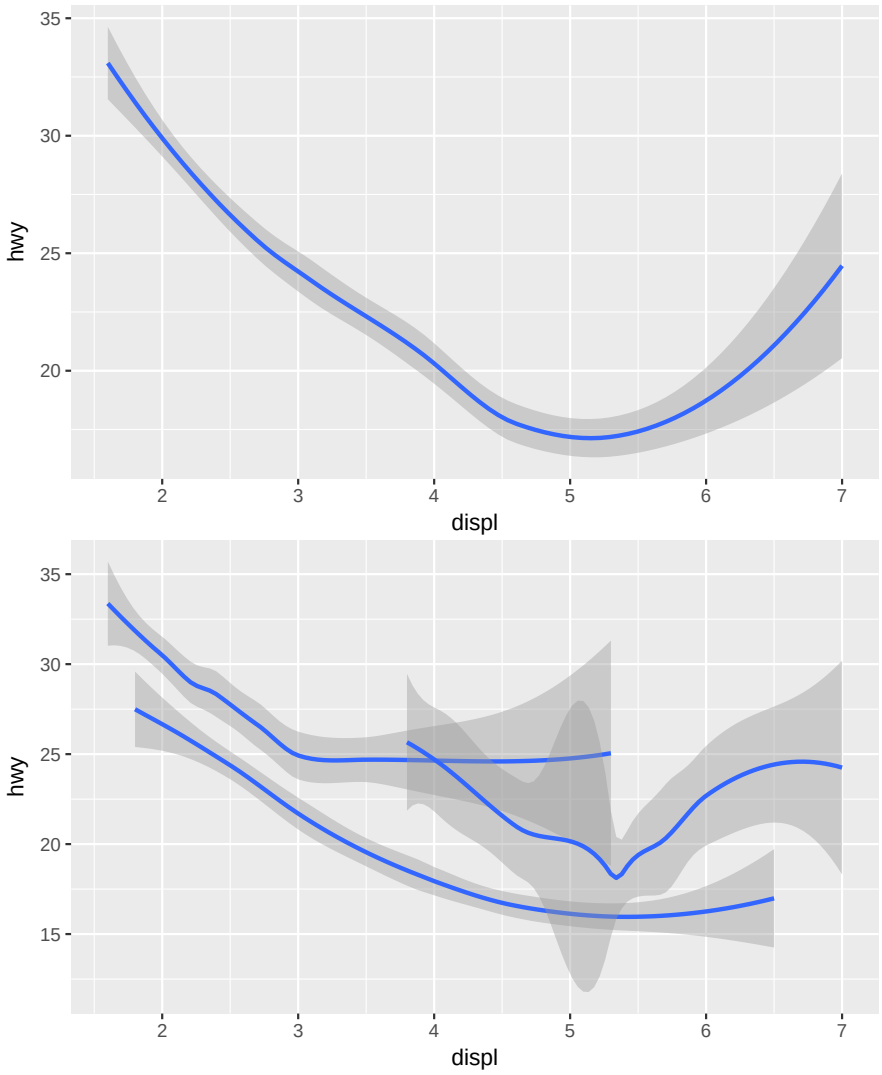
But there is no legend in this code:

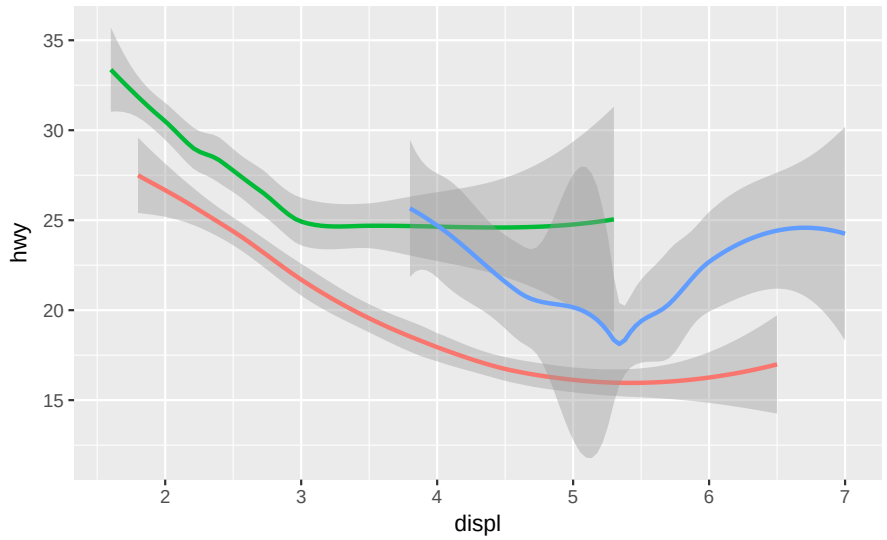
```
ggplot(data = mpg) +
  geom_smooth(
    mapping = aes(x = displ, y = hwy, color = drv),
    show.legend = FALSE
  )
#> `geom_smooth()` using method = 'loess'
```



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'
```





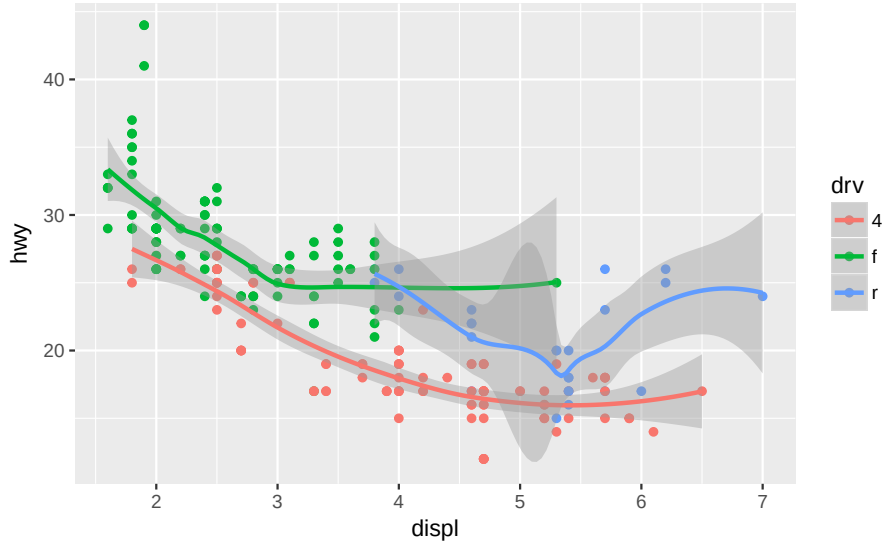
the legend is suppressed because there are three plots, and adding a legend that only appears in the last one would make the presentation asymmetric. Additionally, the purpose of this plot is to illustrate the difference between not grouping, using a `group` aesthetic, and using a `color` aesthetic (with 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?

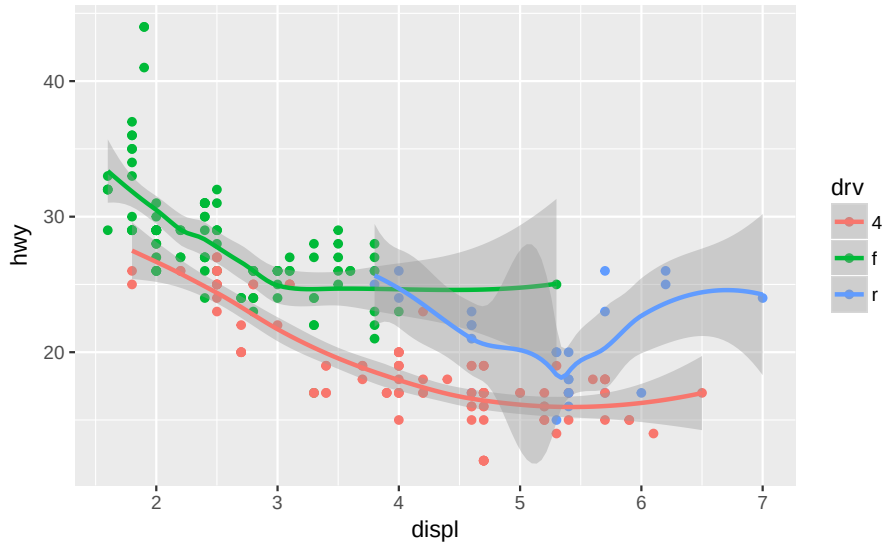
It adds standard error bands to the lines.

```
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'
```

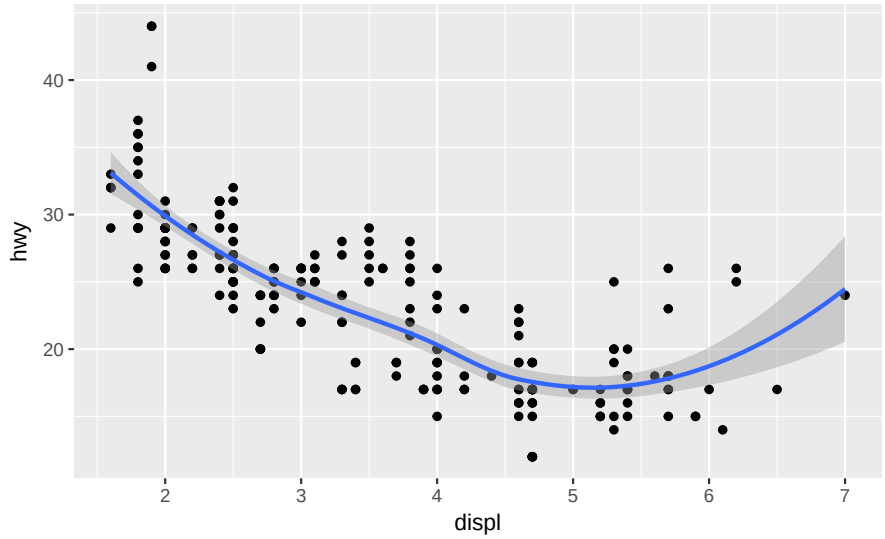


3.6.5 Exercise 5.

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

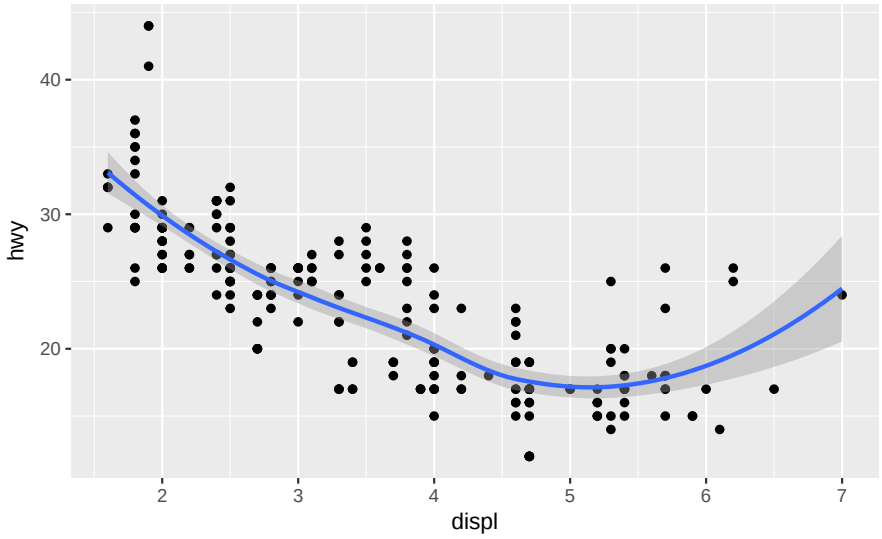
No. Because both `geom_point` and `geom_smooth` use the same data and mappings. They will inherit those options from the `ggplot` object, and thus don't need to specified again (or twice).

```
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)) +
```

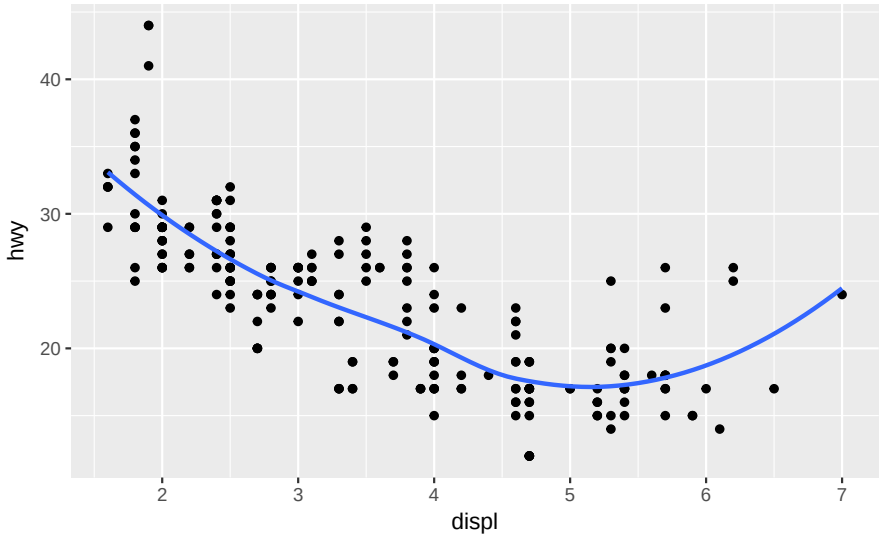
```
  geom_smooth(data = mpg, mapping = aes(x = displ, y = hwy))
#> `geom_smooth()` using method = 'loess'
```



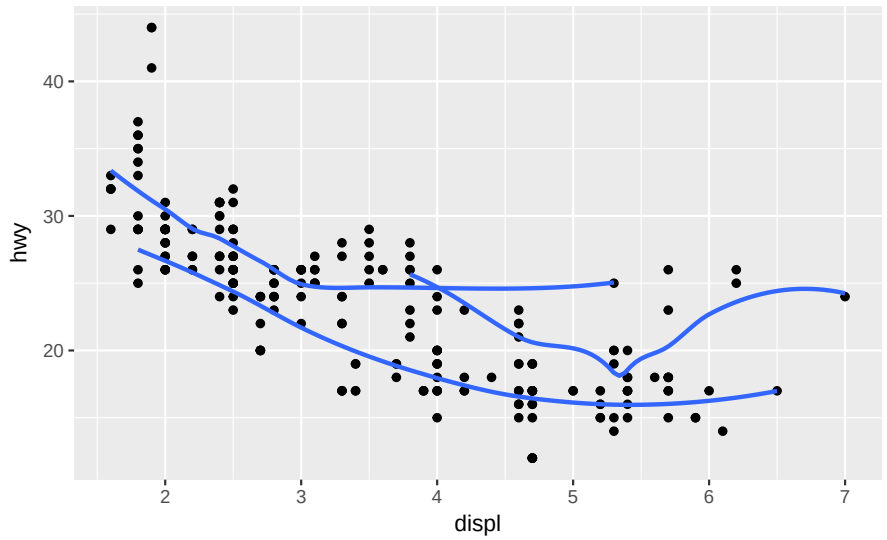
3.6.6 Exercise 6.

Recreate the R code necessary to generate the following graphs.

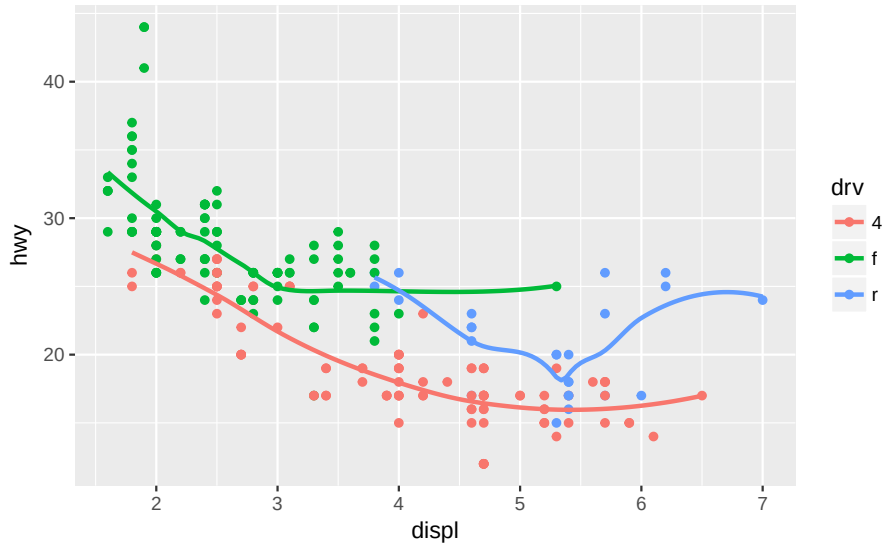
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```



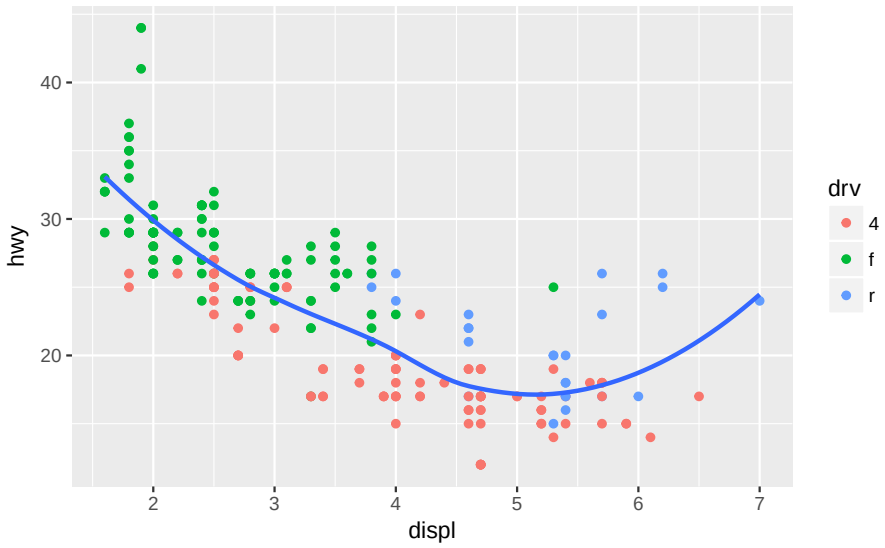

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(mapping = aes(group = drv), se = FALSE)
#> `geom_smooth()` using method = 'loess'
```



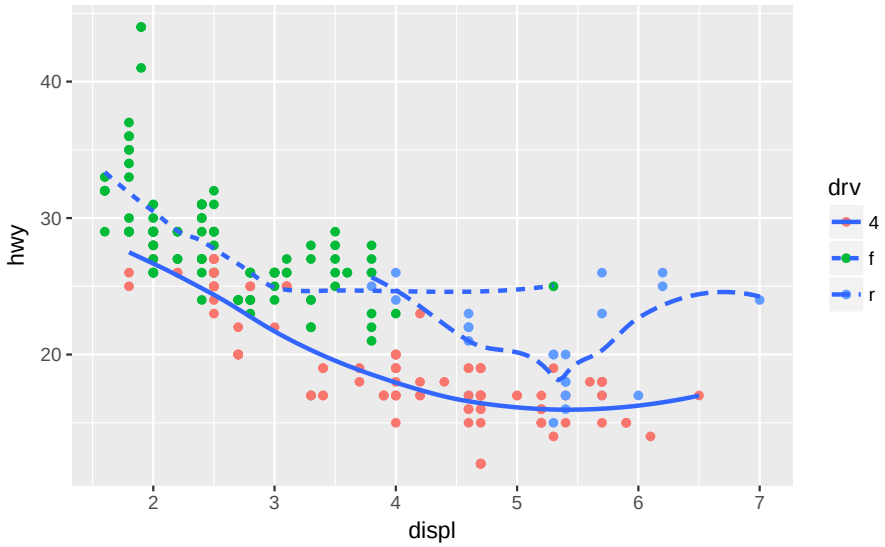

```
ggplot(mpg, aes(x = displ, y = hwy, colour = drv)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```



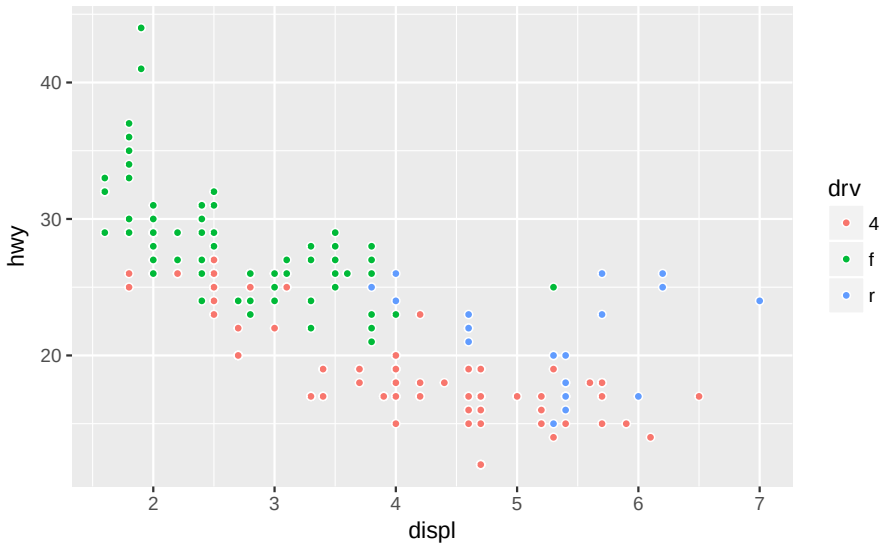

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(colour = drv)) +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```




```
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

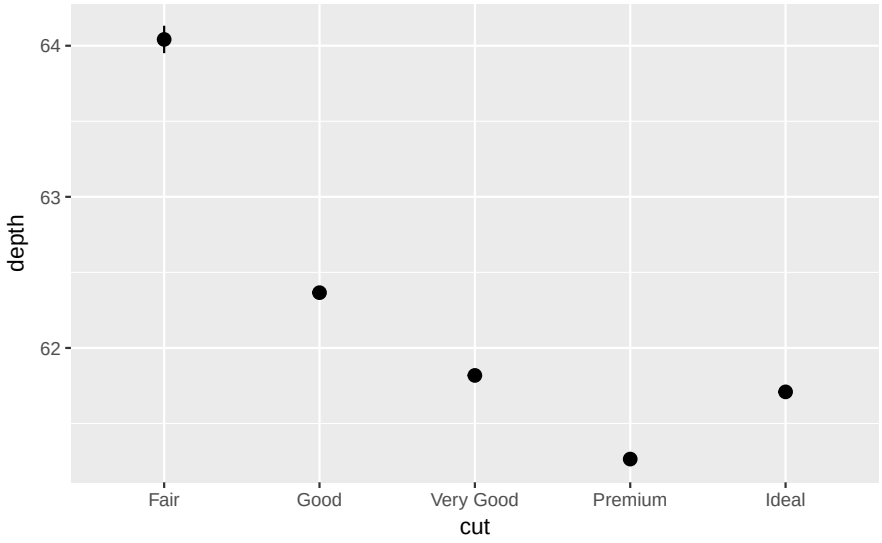
3.7.1 Exercise 1.

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.

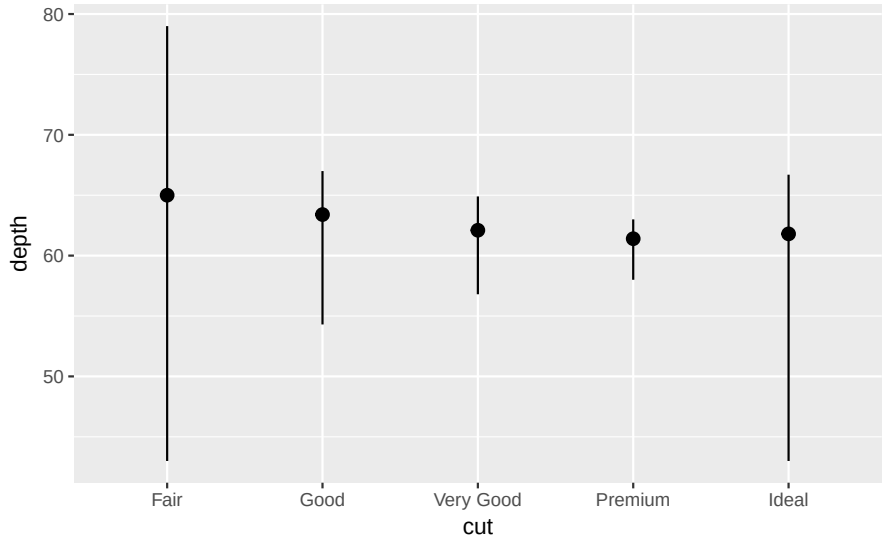
But, the default `stat` for `geom_pointrange` is `identity`, so use `geom_pointrange(stat = "summary")`.

```
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

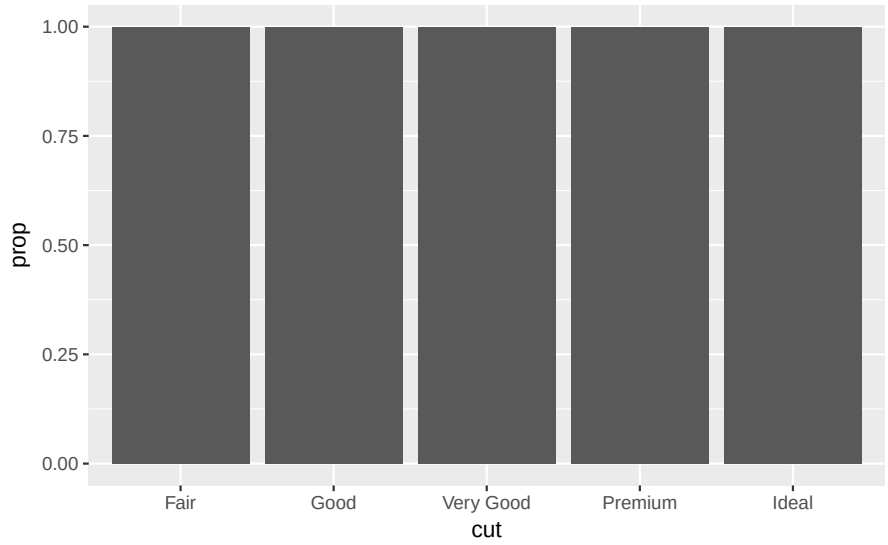
There's parameters such as `method` which determines which method is used to calculate the predictions and confidence interval, and some other arguments that are passed to that.

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?

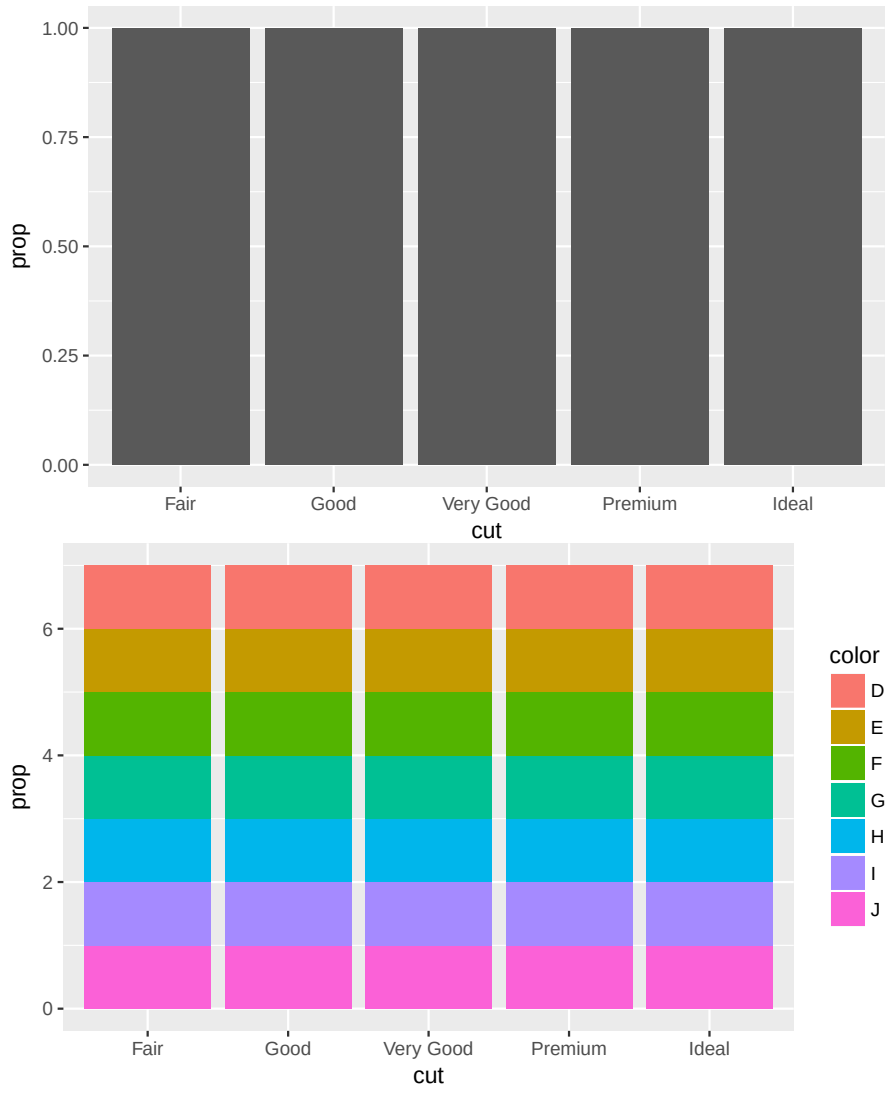
If `group` is not set to 1, then all the bars have `prop == 1`. The function `geom_bar` assumes that the groups are equal to the `x` values, since the stat computes the counts within the group.

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



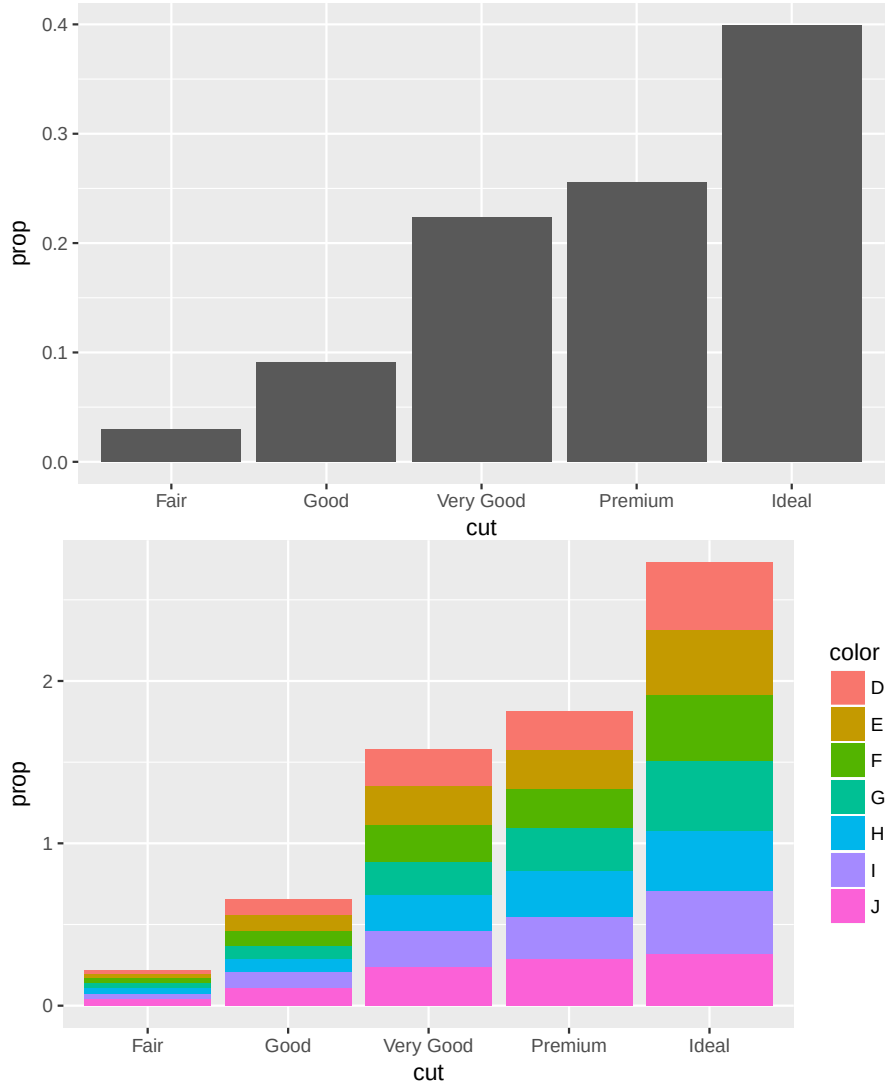
The problem with these two plots is that the proportions are calculated within the groups.

```
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))
```



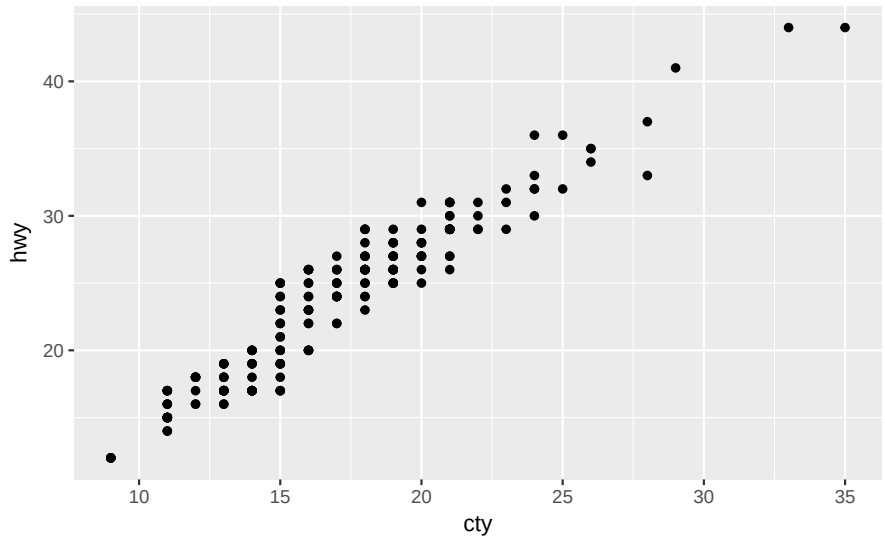
3.8 Position Adjustments

3.8.1 Exercise 1.

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

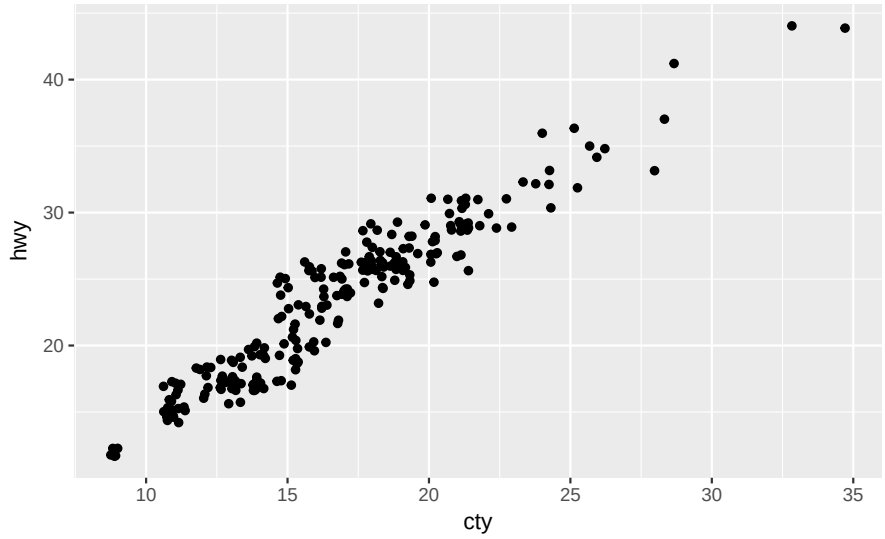
There is overplotting because there are multiple observations for each combination of `cty` and `hwy`.

```
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")
```



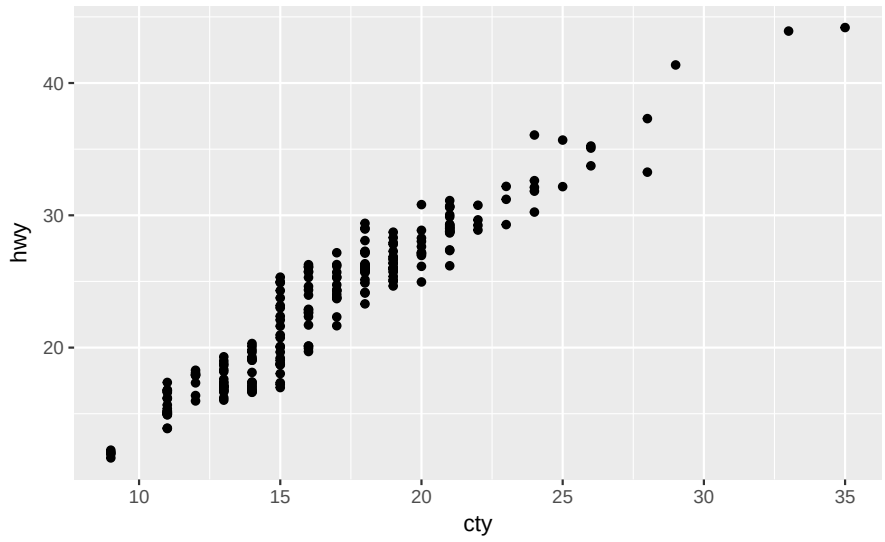
3.8.2 Exercise 2.

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.

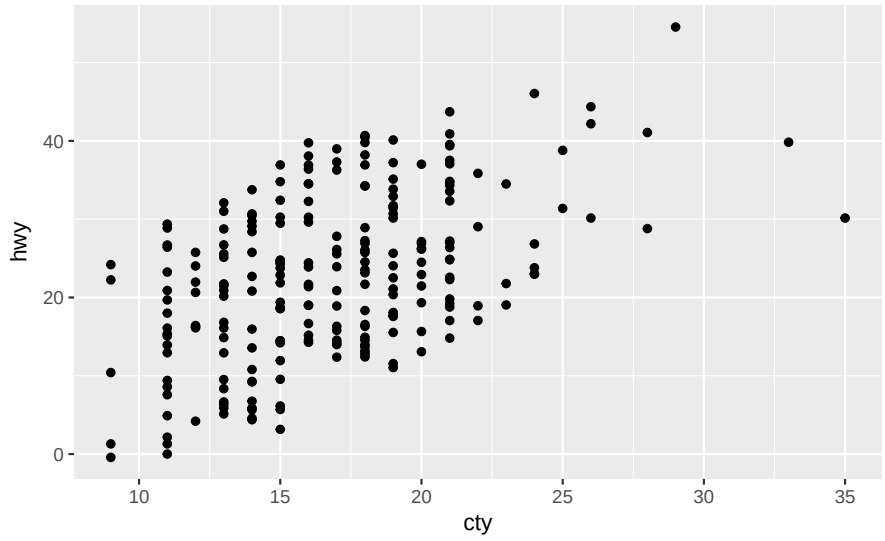
No horizontal jitter

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



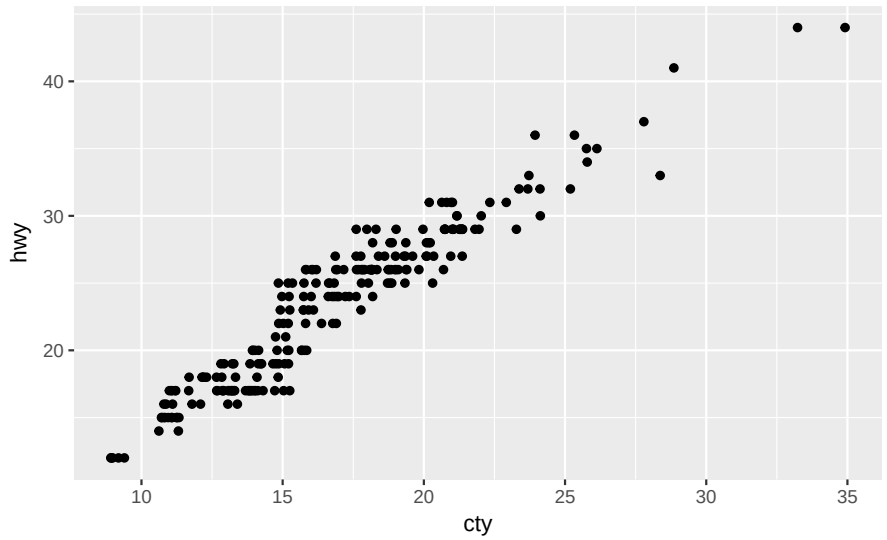
Way too much vertical jitter

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



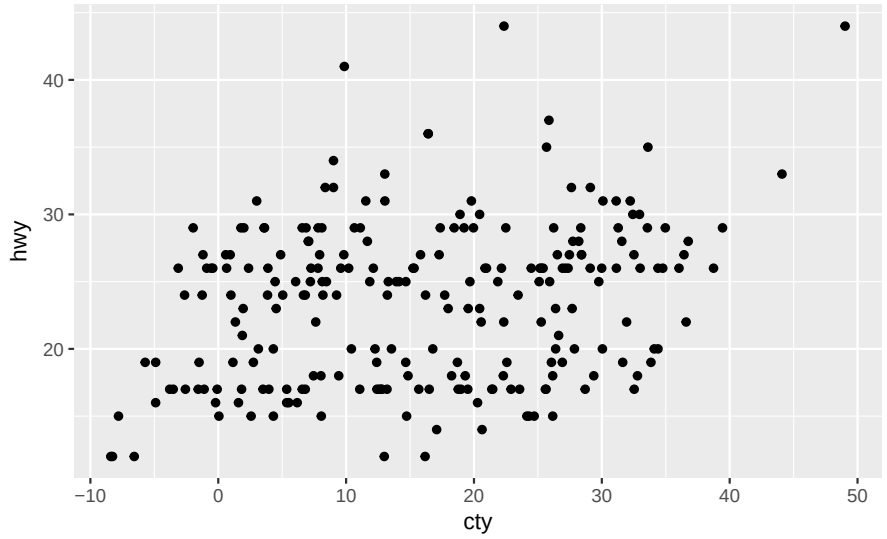
Only horizontal jitter:

```
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

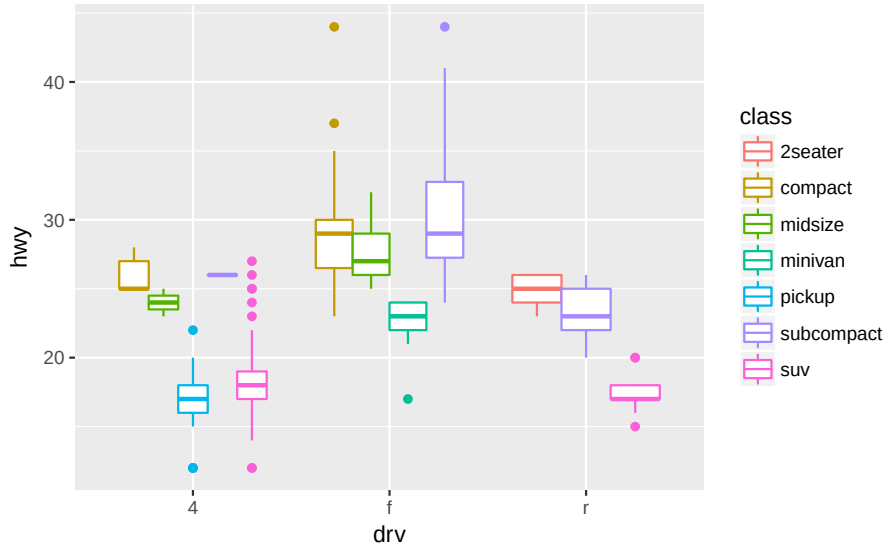
3.8.4 Exercise 4.

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

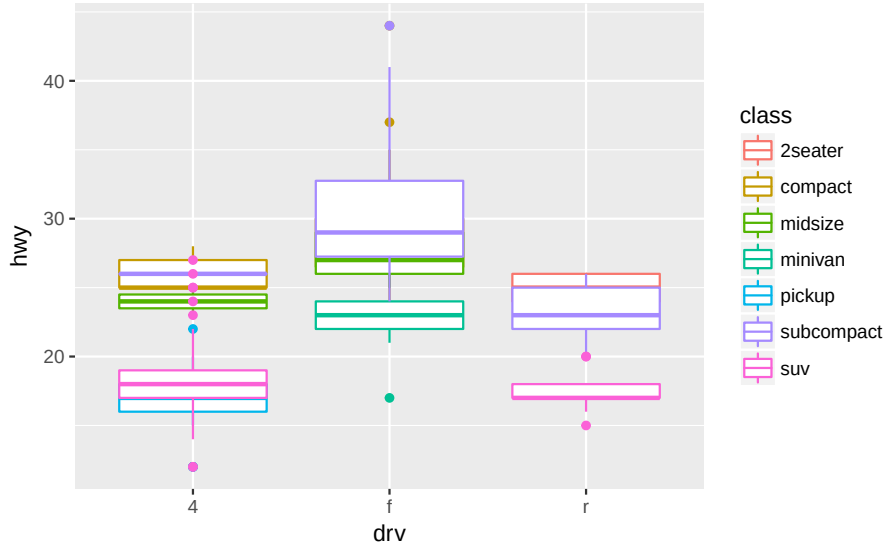
The default position for `geom_boxplot` is `position_dodge` (see its docs).

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")
```



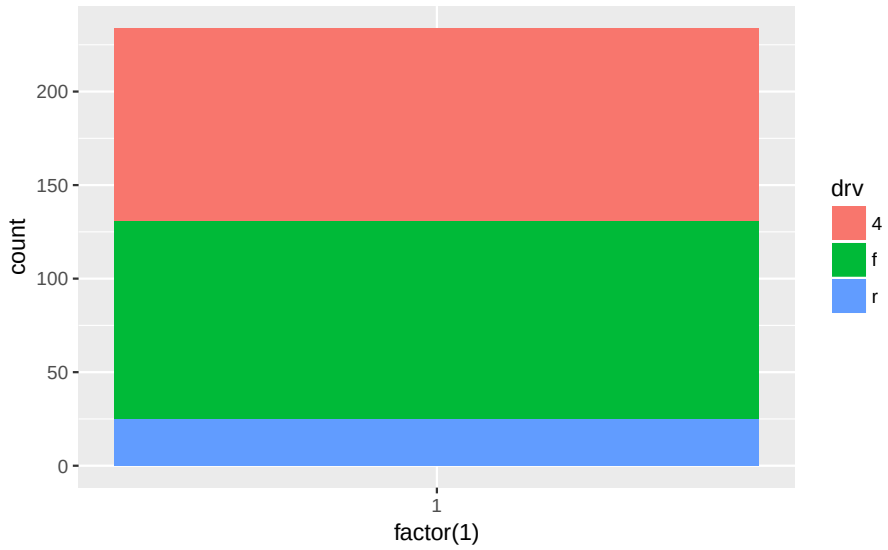
3.9 Coordinate Systems

3.9.1 Exercise 1.

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

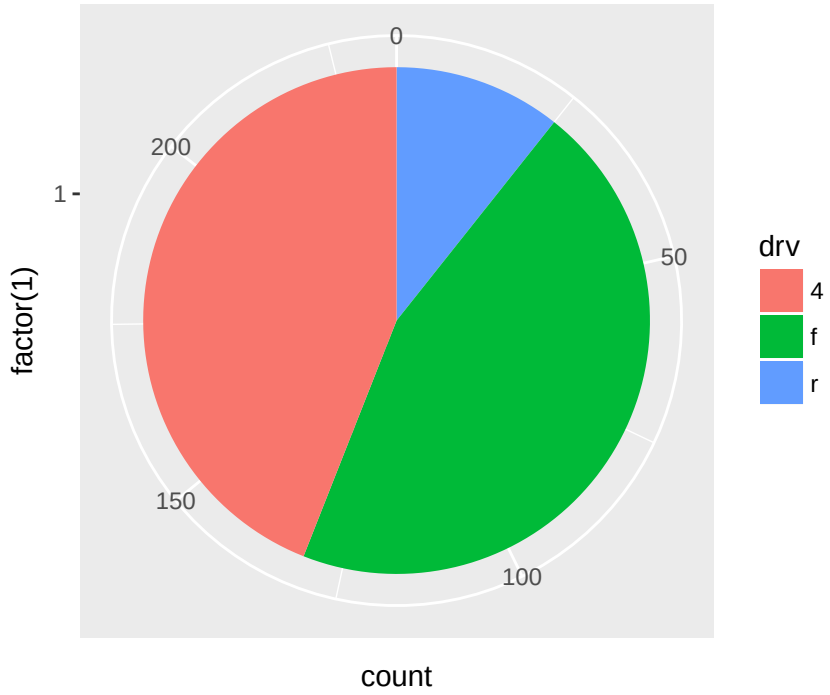
This is a stacked bar chart with a single category

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



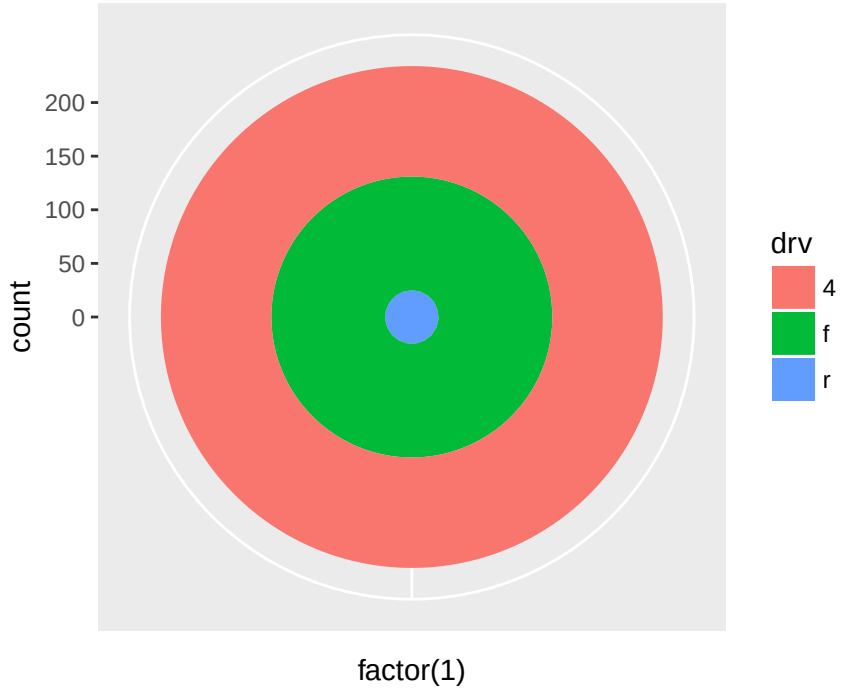
See the documentation for coord_polar for an example of making a pie chart. In particular, `theta = "y"`, meaning that the angle of the chart is the `y` variable has to be specified.

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



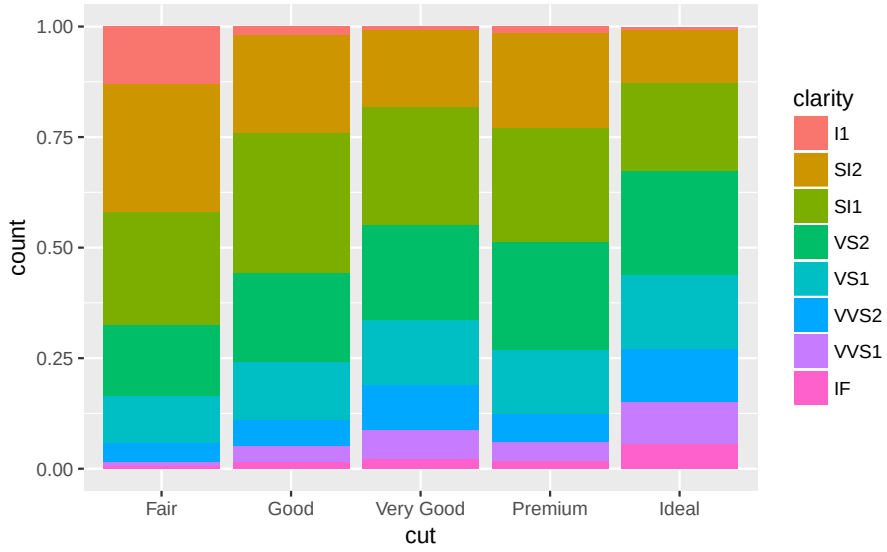
If `theta = "y"` is not specified, then you get a bull's-eye chart

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



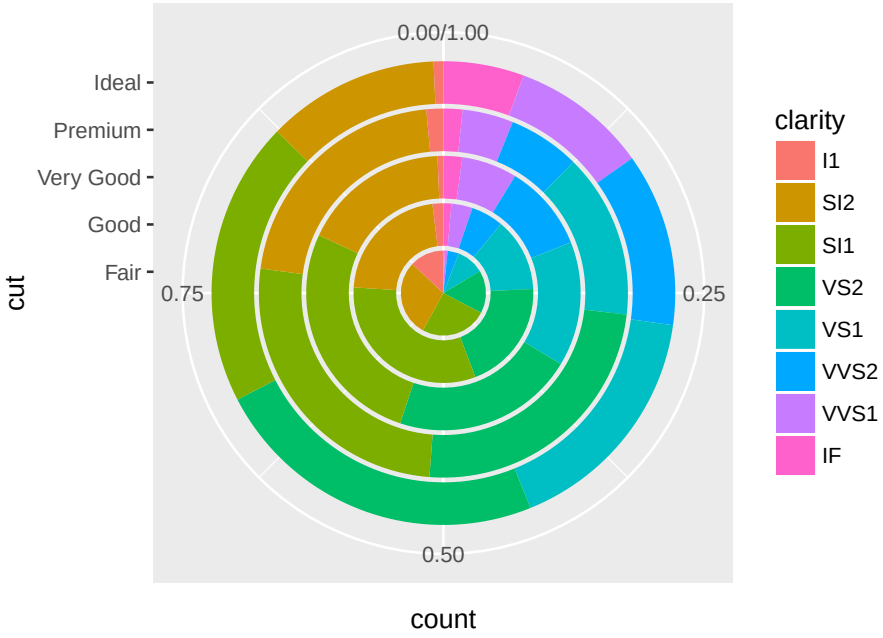
If you had a multiple stacked bar chart, like,

```
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")
```

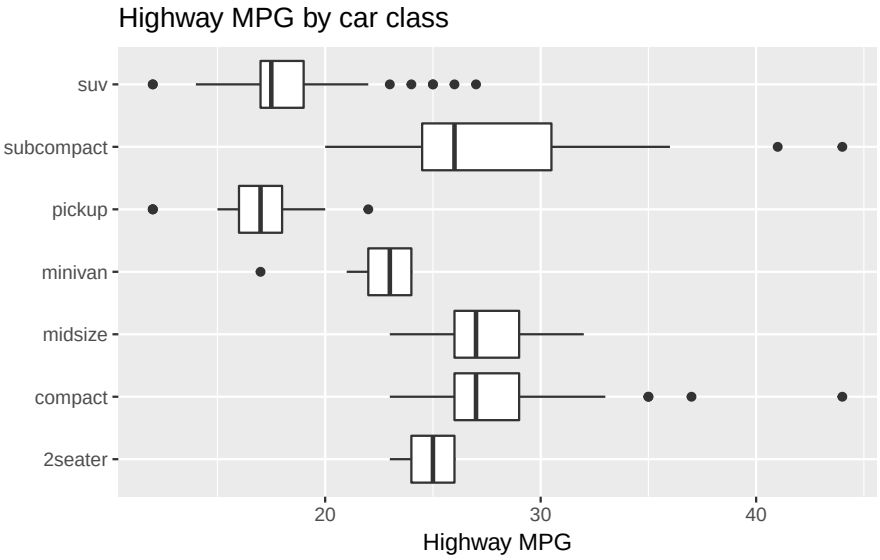


3.9.2 Exercise 2.

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:

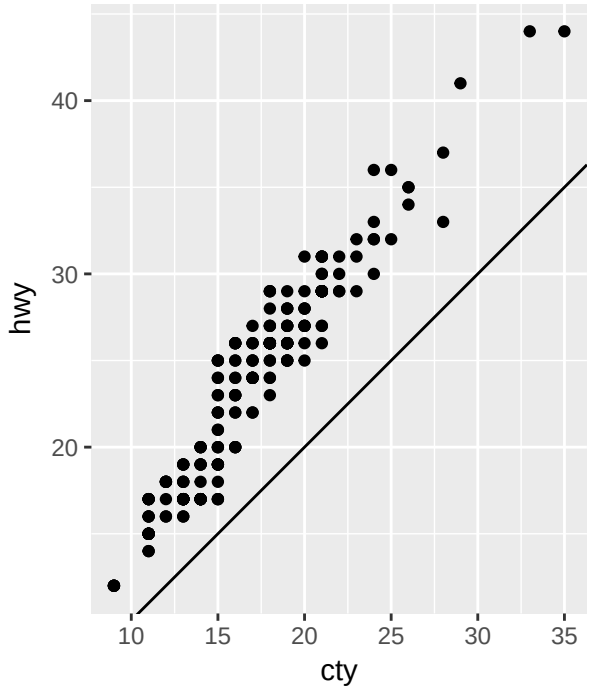
- `coord_map` uses a 2D projection: by default the Mercator project of the sphere to the plot. But this requires transforming all geoms.
- `coord_quickmap` uses a approximate, but faster, map projection using the lat/long ratio as an approximation. This is "quick" because the shapes don't need to be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

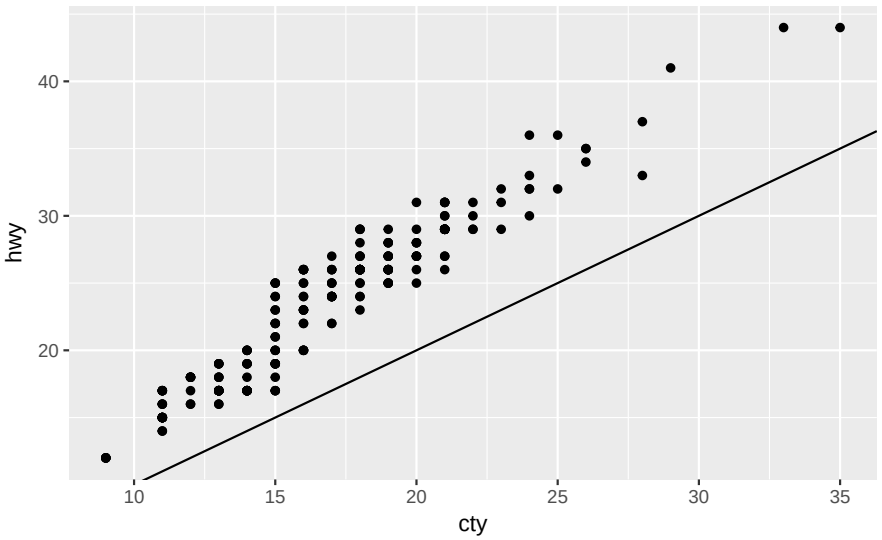
The coordinates `coord_fixed` ensures that the `abline` is at a 45 degree angle, which makes it easy to compare the highway and city mileage against what it would be if they were exactly the same.

```
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

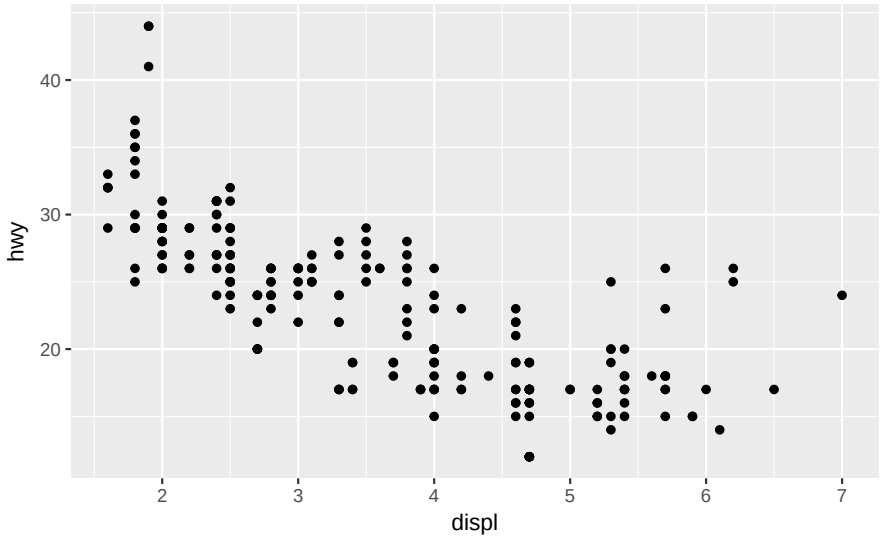
Tweak each of the following R commands so that they run correctly:

```
library("tidyverse")
#> -- Attaching packages --------------------------------------------------- tidyverse 1.2.1 --
#> ✓ ggplot2 2.2.1     ✓ purrr   0.2.4
#> ✓ tibble  1.4.2     ✓ dplyr   0.7.4
#> ✓ tidyr   0.8.0     ✓ stringr 1.3.0
#> ✓ readr   1.1.1     ✓ forcats 0.3.0
#> -- 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.  8040  9.10  8.97  5.67
#> 2  3.11 Fair    J     I1       65.9   57.  9823  9.15  9.02  5.98
#> 3  3.01 Premium F     I1       62.2   56.  9925  9.24  9.13  5.73
#> 4  3.05 Premium E     I1       60.9   58. 10453  9.26  9.25  5.66
#> 5  3.02 Fair    I     I1       65.2   56. 10577  9.11  9.02  5.91
#> 6  3.01 Fair    H     I1       56.1   62. 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.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      623            627       -4.      933
#> 4  2013     1     1      728            732       -4.     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.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      542            540        2.      923
#> 4  2013     1     1      554            600       -6.      812
#> 5  2013     1     1      554            558       -4.      740
#> 6  2013     1     1      558            600       -2.      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.     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.     1754
#> 2  2013    10     7     1350           1350         0.     1736
#> 3  2013    10     7     1357           1359        -2.     1858
#> 4  2013    10    16      657            700        -3.     1258
#> 5  2013    11     1      658            700        -2.     1329
#> 6  2013     3    18     1844           1847        -3.       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.     2056
#> 5  2013     1     3     1850           1745       65.     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.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      542            540        2.      923
#> 4  2013     1     1      544            545       -1.     1004
#> 5  2013     1     1      554            600       -6.      812
#> 6  2013     1     1      554            558       -4.      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.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      542            540        2.      923
#> 4  2013     1     1      544            545       -1.     1004
#> 5  2013     1     1      554            600       -6.      812
#> 6  2013     1     1      554            558       -4.      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.       40
#> 2  2013     2     3     2022           2055      -33.     2240
#> 3  2013    11    10     1408           1440      -32.     1549
#> 4  2013     1    11     1900           1930      -30.     2233
#> 5  2013     1    29     1703           1730      -27.     1947
#> 6  2013     8     9      729            755      -26.     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.      1442
#> 2  2013     4    13      537            527      10.       622
#> 3  2013    12     6      922            851      31.      1021
#> 4  2013     2     3     2153           2129      24.      2247
#> 5  2013     2     5     1303           1315     -12.      1342
#> 6  2013     2    12     2123           2130      -7.      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.      1516
#> 2  2013     1     2      909            900       9.      1525
#> 3  2013     1     3      914            900      14.      1504
#> 4  2013     1     4      900            900       0.      1516
#> 5  2013     1     5      858            900      -2.      1519
#> 6  2013     1     6     1019            900      79.      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.     2222
#> 3  2013     1     4     1240           1200        40.     1333
#> 4  2013     1     4     1829           1615       134.     1937
#> 5  2013     1     4     2128           2129        -1.     2218
#> 6  2013     1     5     1155           1200        -5.     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.      830       11.
#> 2      533        4.      850       20.
#> 3      542        2.      923       33.
#> 4      544       -1.     1004      -18.
#> 5      554       -6.      812      -25.
#> 6      554       -4.      740       12.
#> # ... 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.      830       11.
#> 2      533        4.      850       20.
#> 3      542        2.      923       33.
#> 4      544       -1.     1004      -18.
#> 5      554       -6.      812      -25.
#> 6      554       -4.      740       12.
#> # ... 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.      830        11.
#> 2      533         4.      850        20.
#> 3      542         2.      923        33.
#> 4      544        -1.     1004       -18.
#> 5      554        -6.      812       -25.
#> 6      554        -4.      740        12.
#> # ... 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.        11.
#> 2  2013     1     1         4.        20.
#> 3  2013     1     1         2.        33.
#> 4  2013     1     1        -1.       -18.
#> 5  2013     1     1        -6.       -25.
#> 6  2013     1     1        -4.        12.
#> # ... 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.        -46
#> 2      555            600        -5.        -45
#> 3      557            600        -3.        -43
```

```
#> 4      557            600       -3.        -43
#> 5      558            600       -2.        -42
#> 6      558            600       -2.        -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.     -1397.
#> 3      126           2250      156.     -1284.
#> 4       32           2359       33.     -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  -1.00e+ 0 -1.00e+ 0  3.67e-16
#> 2 -2.50   3.06e-16  3.06e-16 -3.27e+15
#> 3 -2.00   1.00e+ 0  1.00e+ 0  2.45e-16
#> 4 -1.50  -1.84e-16 -1.84e-16 -5.44e+15
#> 5 -1.00  -1.00e+ 0 -1.00e+ 0  1.22e-16
#> 6 -0.500  6.12e-17  6.12e-17 -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. -0.990    0.143
#> 2 -2.50     0. -0.801    0.747
#> 3 -2.00     1. -0.416    2.19
#> 4 -1.50     0.  0.0707 -14.1
#> 5 -1.00    -1.  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.

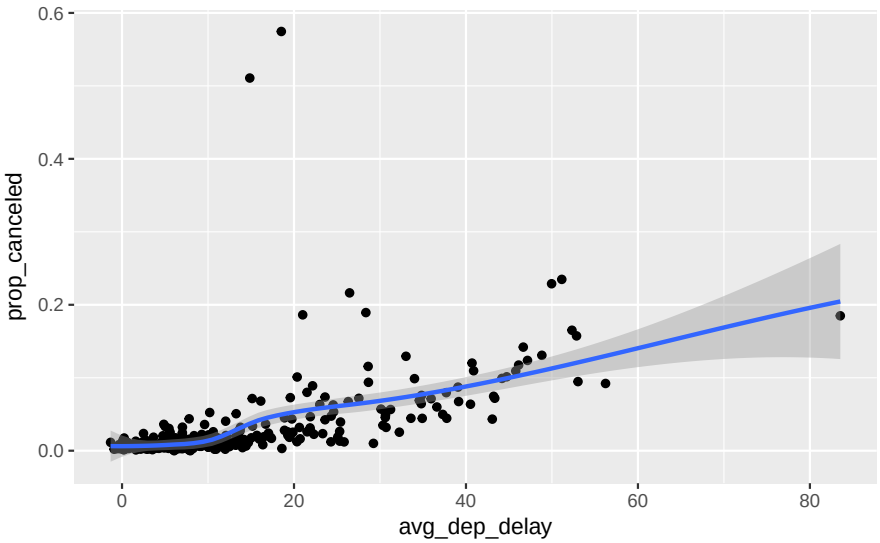
```
filter(flights, !is.na(dep_delay), is.na(arr_delay)) %>%
  select(dep_time, arr_time, sched_arr_time, dep_delay, arr_delay)
#> # A tibble: 1,175 x 5
#>   dep_time arr_time sched_arr_time dep_delay arr_delay
#>      <int>    <int>          <int>     <dbl>     <dbl>
#> 1     1525     1934           1805        -5.        NA
#> 2     1528     2002           1647        29.        NA
#> 3     1740     2158           2020        -5.        NA
#> 4     1807     2251           2103        29.        NA
#> 5     1939       29           2151        59.        NA
#> 6     1952     2358           2207        22.        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.    -5.30
#> 2    5.    -4.80
#> 3    6.    -3.38
```

```
#> 4    9.    -1.45
#> 5    8.    -1.11
#> 6   10.     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.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      542            540        2.      923
#> 4  2013     1     1      554            558       -4.      740
#> 5  2013     1     1      555            600       -5.      913
#> 6  2013     1     1      558            600       -2.      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.

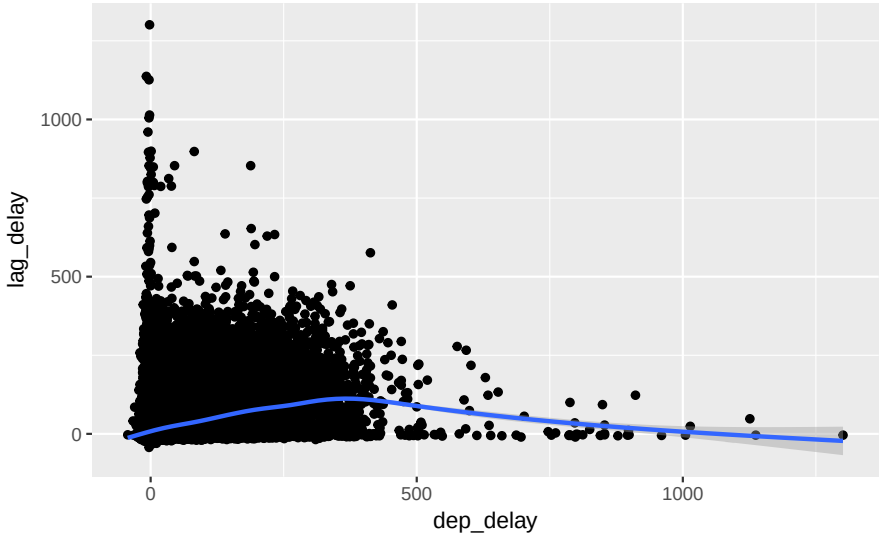
```
flights %>%
  filter(!is.na(arr_delay), arr_delay > 0) %>%
  group_by(dest) %>%
  mutate(total_delay = sum(arr_delay - min(arr_delay)),
         prop_delay = arr_delay / sum(arr_delay))
#> # A tibble: 133,004 x 21
#> # Groups:   dest [103]
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      517            515        2.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      542            540        2.      923
#> 4  2013     1     1      554            558       -4.      740
#> 5  2013     1     1      555            600       -5.      913
#> 6  2013     1     1      558            600       -2.      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.      38. -0.447     1450           1500     1547
#> 2 ATL        65.     112. -0.420     1709           1700     1923
#> 3 GSP        55.      92. -0.402     2040           2025     2225
#> 4 BOS        23.      38. -0.395     1954           2000     2131
#> 5 BNA        70.     113. -0.381     1914           1910     2045
#> 6 MSP        93.     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.     1558           1513
#> 2 ATL     -4.88         113.         9.81      65.     1709           1700
#> 3 GSP     -4.72          93.4        8.13      55.     2040           2025
#> 4 BNA     -4.05         114.        11.0       70.     1914           1910
#> 5 CVG     -3.98          96.0        8.52      62.     1359           1343
#> 6 BOS     -3.63          39.0        4.95      21.     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

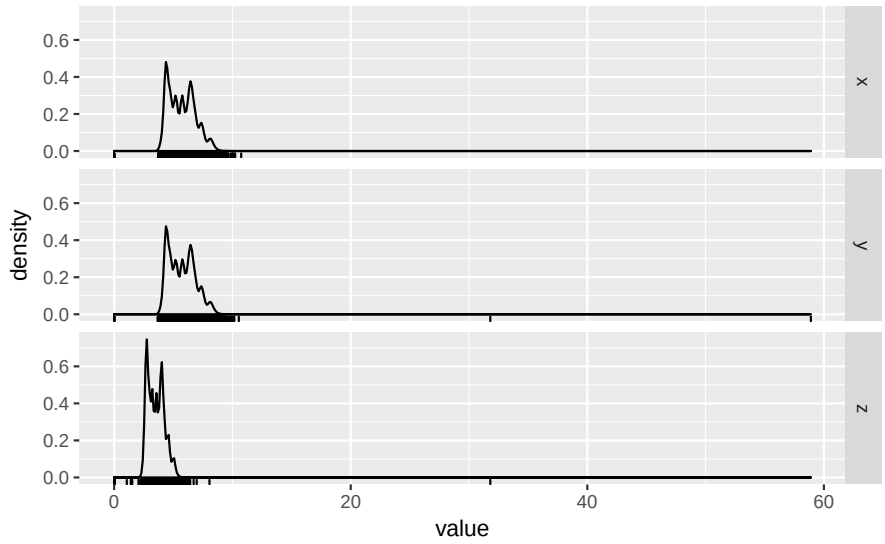
7.3 Variation

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)

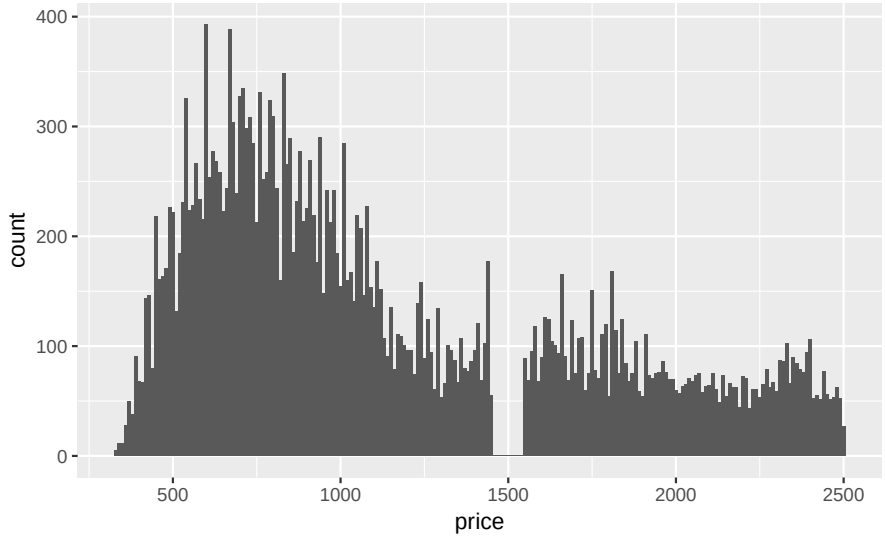
According to the documentation for `diamonds`: `x` is length, `y` is width, and `z` is depth. I don't know if I would have figured that out before; maybe if there was data on the type of cuts.

7.3.2 Exercise 2.

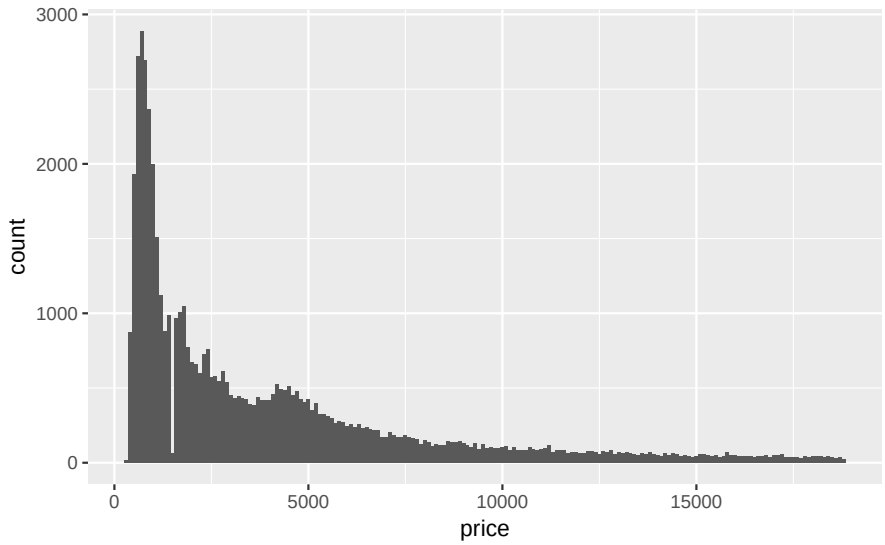
Explore the distribution of price. Do you discover anything unusual or surprising? (Hint: Carefully think about the `binwidth` and make sure you try a wide range of values.)

- 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)
```

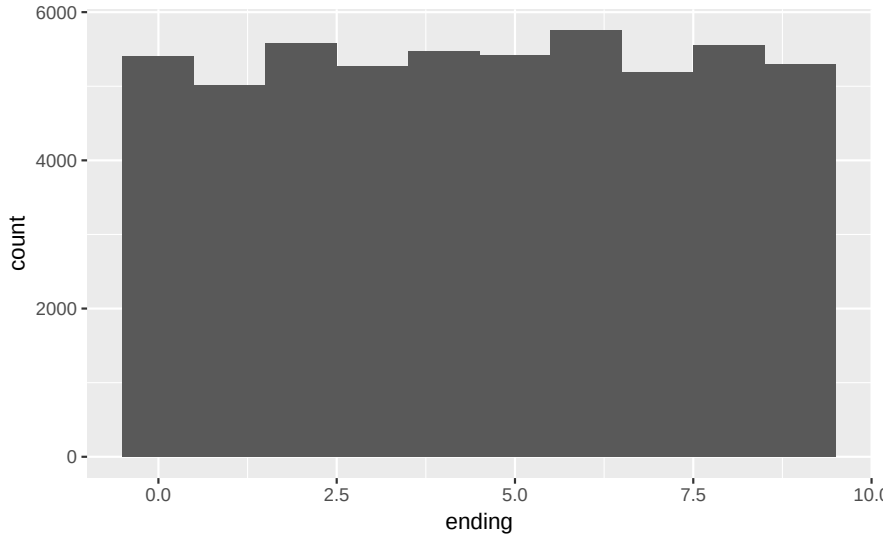


```
ggplot(filter(diamonds), aes(x = price)) +
  geom_histogram(binwidth = 100, center = 0)
```

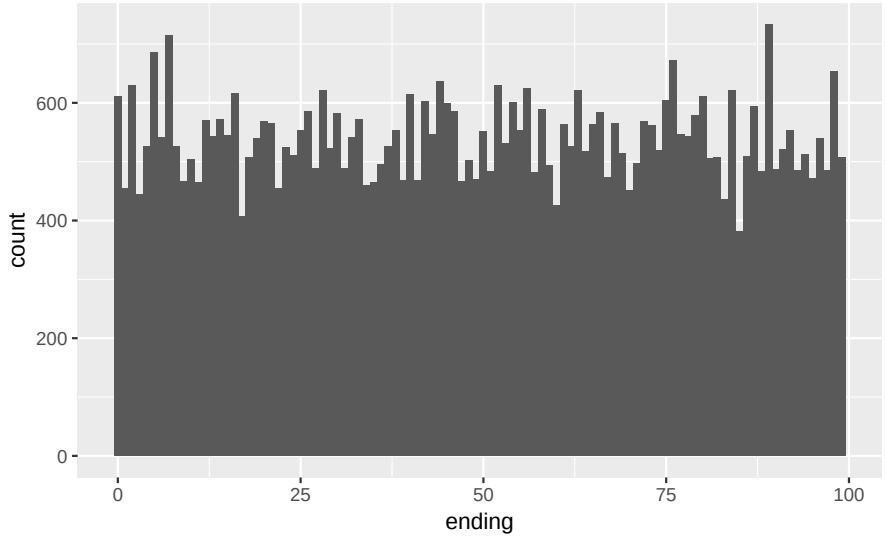


Distribution of last digit

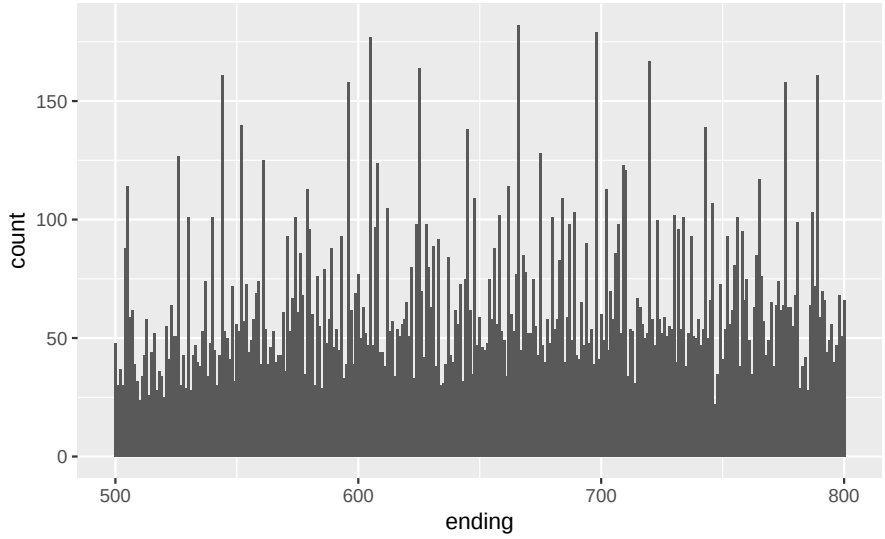
```
diamonds %>%
  mutate(ending = price %% 10) %>%
  ggplot(aes(x = ending)) +
  geom_histogram(binwidth = 1, center = 0) +
  geom_bar()
```




```
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.

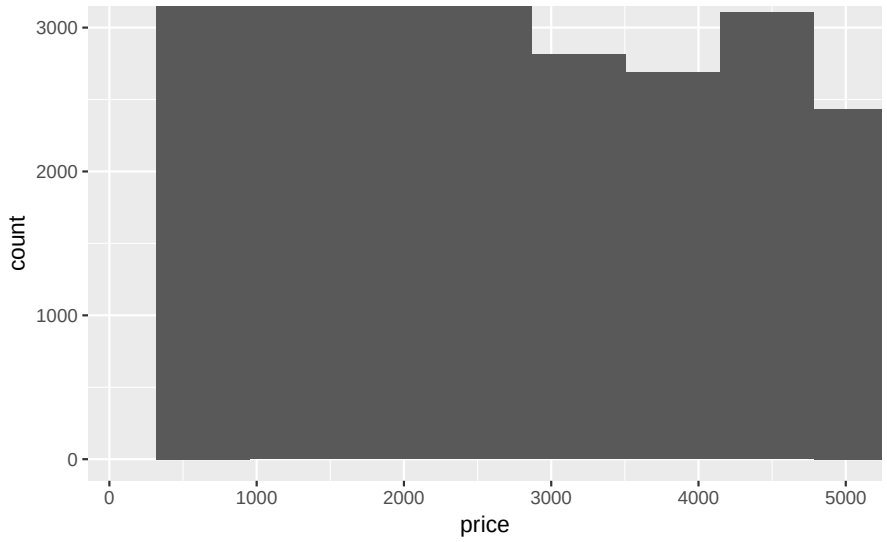
```
diamonds %>%
   filter(carat >= 0.9, carat <= 1.1) %>%
   count(carat) %>%
   print(n = 30)
#> # A tibble: 21 x 2
#>    carat     n
#>    <dbl> <int>
#>  1 0.900  1485
#>  2 0.910   570
#>  3 0.920   226
#>  4 0.930   142
#>  5 0.940    59
#>  6 0.950    65
#>  7 0.960   103
#>  8 0.970    59
#>  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?

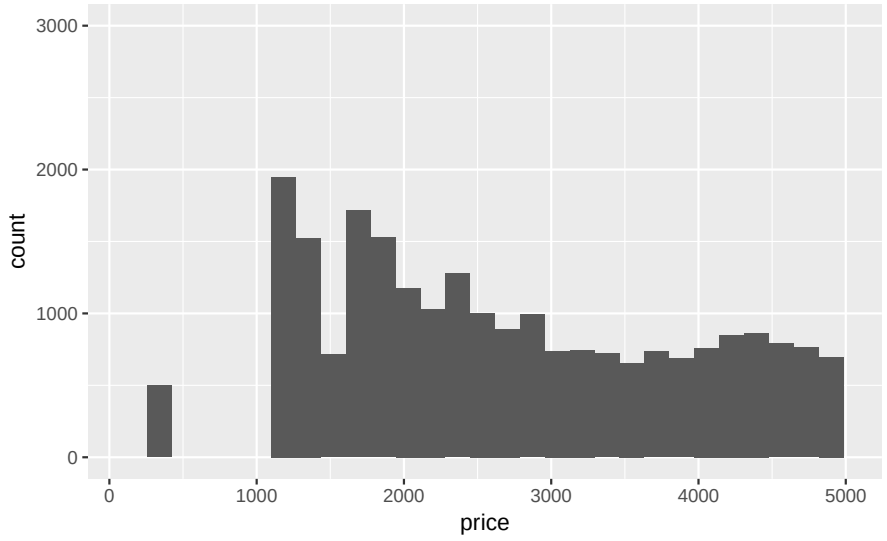
`coord_cartesian` simply zooms in on the area specified by the limits. The calculation of the histogram is unaffected.

```
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).
```



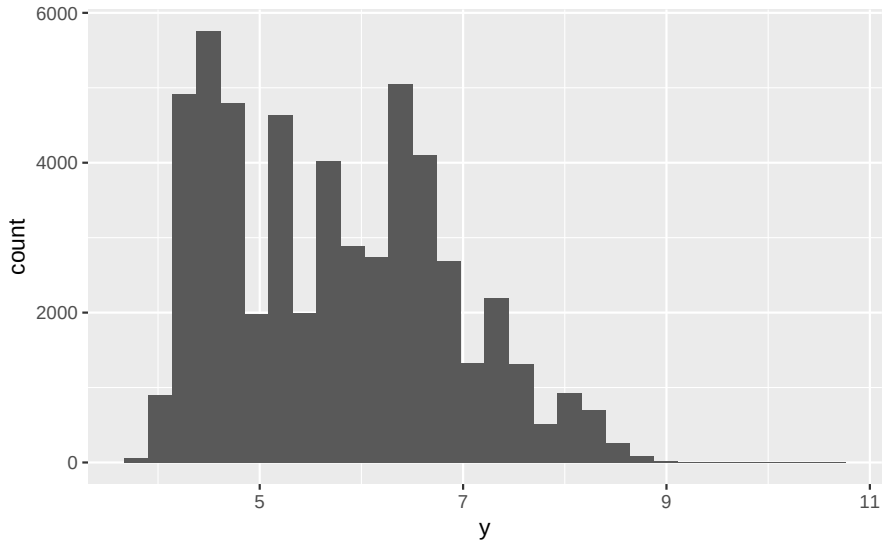
7.4 Missing Values

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?

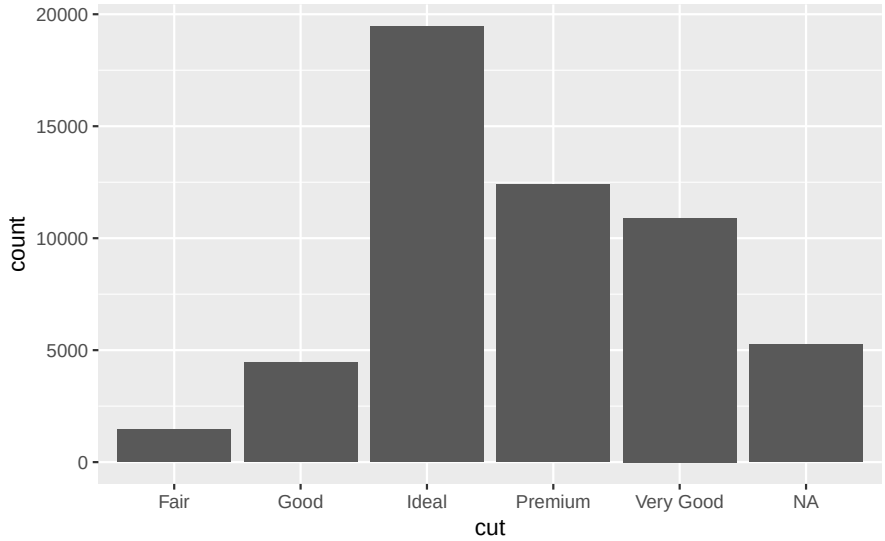
Missing values are removed when the number of observations in each bin are calculated. See the warning message: `Removed 9 rows containing non-finite values (stat_bin)`

```
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

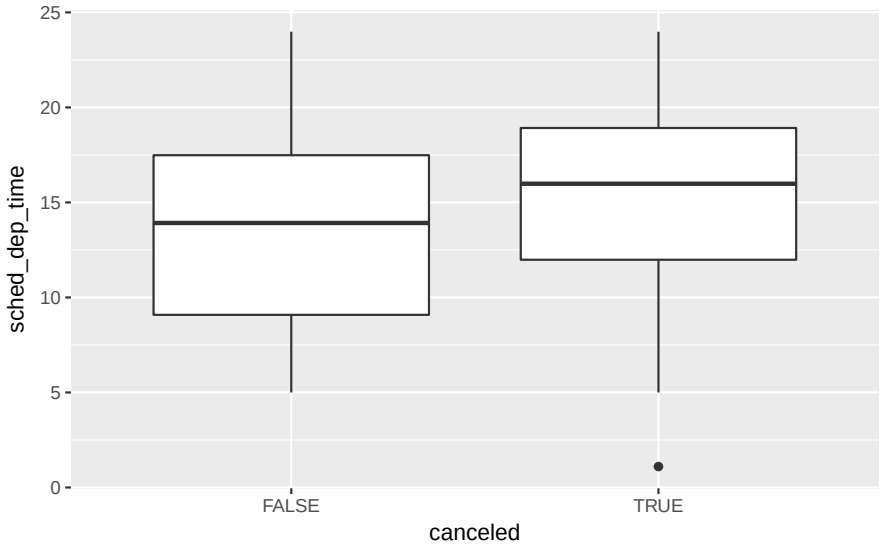
7.5.1.1 Exercise 1

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?

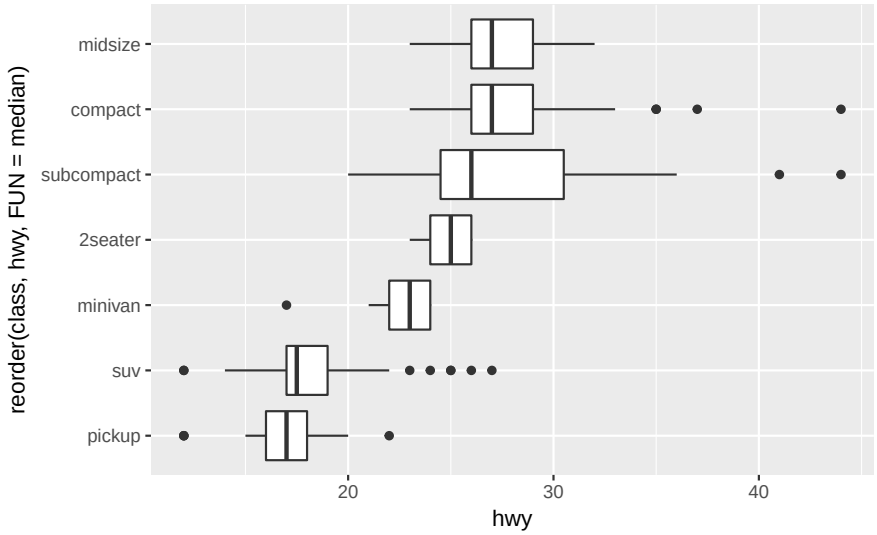
TODO I'm unsure what this question is asking conditional on using only the tools introduced in the book thus far.

7.5.1.3 Exercise 3

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

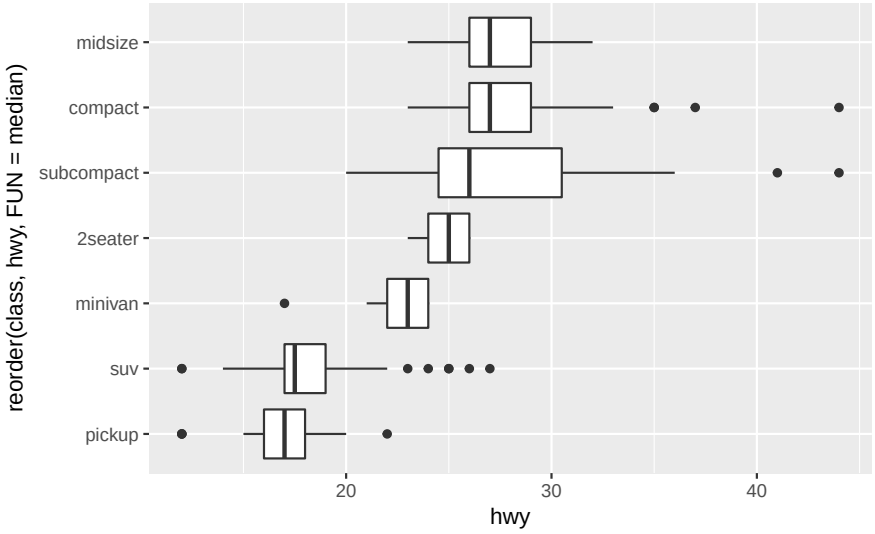
Earlier we created a horizontal box plot of the distribution `hwy` by `class`, using `geom_boxplot` and `coord_flip`:

```
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?

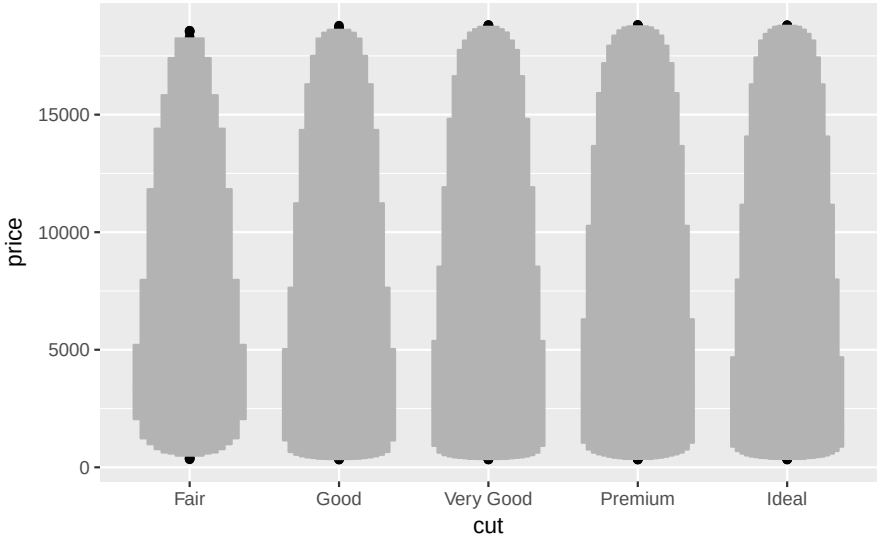
The boxes of the letter-value plot correspond to many more quantiles. They are useful for larger datasets because

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()
```

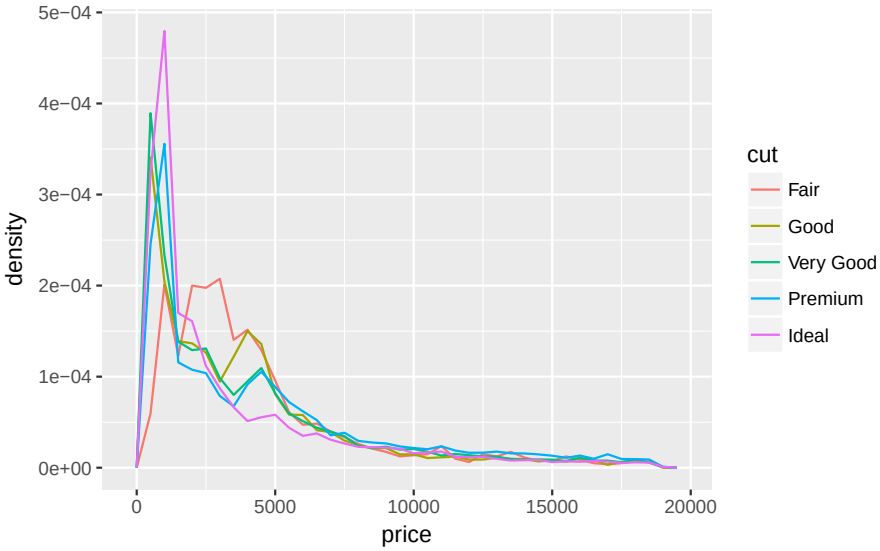


7.5.1.5 Exercise 5

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

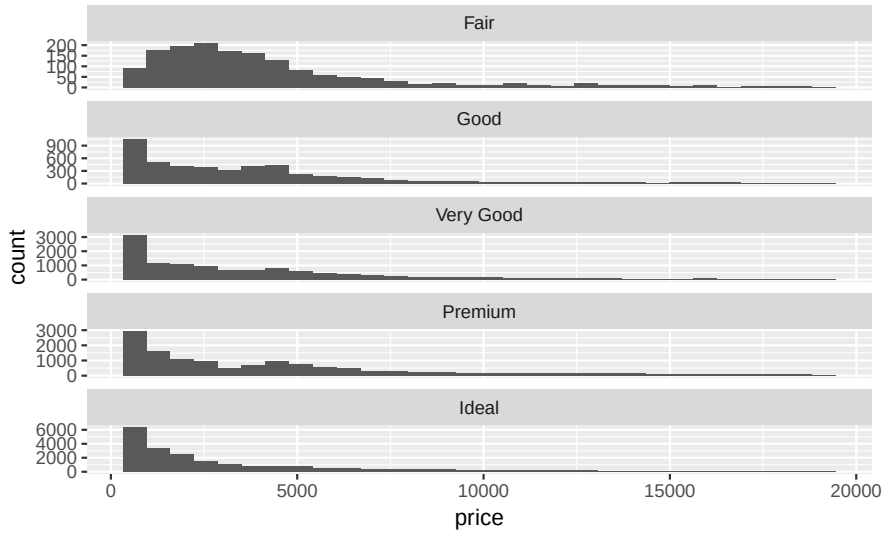
I produce plots for these three methods below. The `geom_freqpoly` is better for look-up: meaning that given a price, it is easy to tell which `cut` has the highest density. However, the overlapping lines makes it difficult to distinguish how the overall distributions relate to each other. The `geom_violin` 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)
```

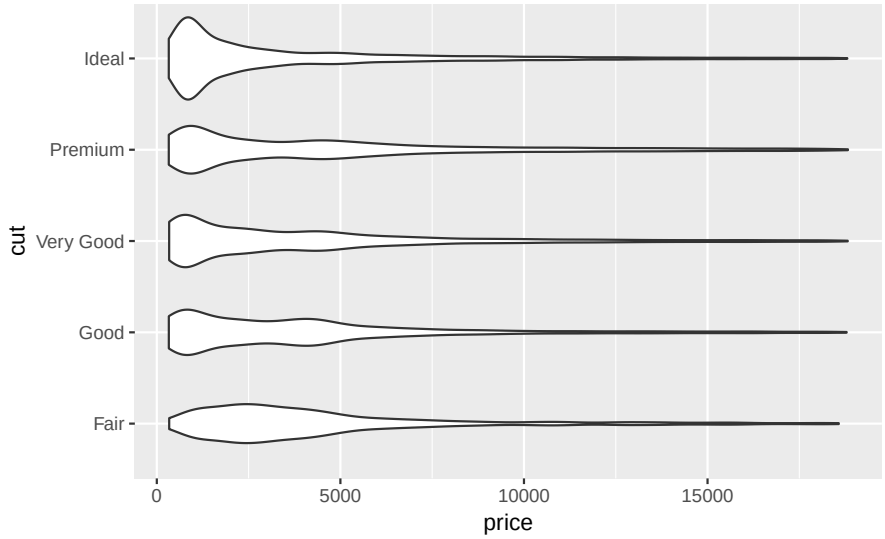



```
ggplot(data = diamonds, mapping = aes(x = price)) +
  geom_histogram() +
```

```
  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

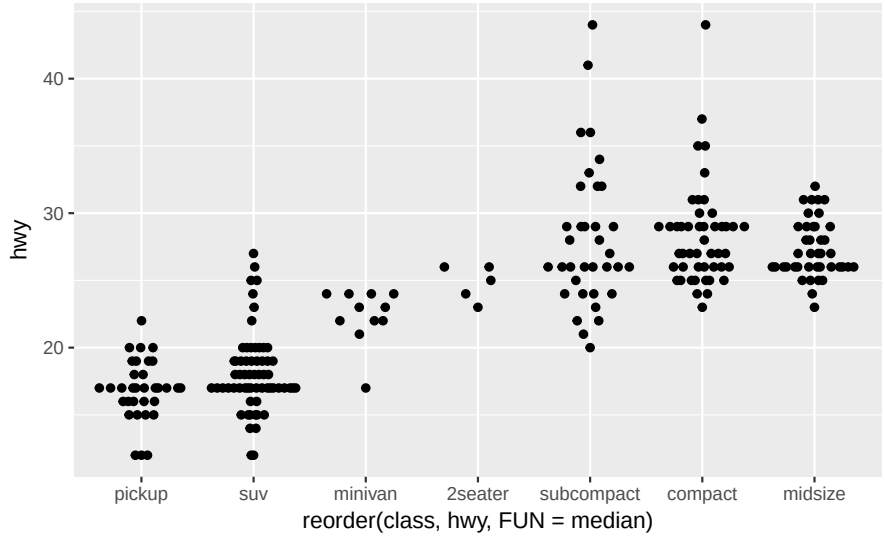
If you have a small dataset, it's sometimes useful to use `geom_jitter()` to see the relationship between a continuous and categorical variable. The **ggbeeswarm** package provides a number of methods similar to `geom_jitter()`. List them and briefly describe what each one does.

There are two methods:

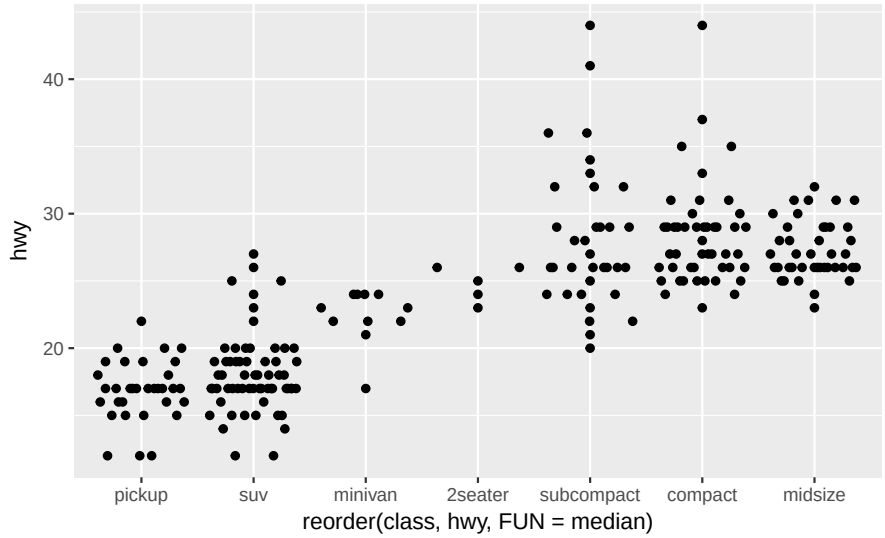
- `geom_quasirandom` that produces plots that resemble something between jitter and violin. There are several different methods that determine exactly how the random location of the points is generated.
- `geom_beeswarm` creates a shape similar to a violin plot, but by offsetting the points.

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

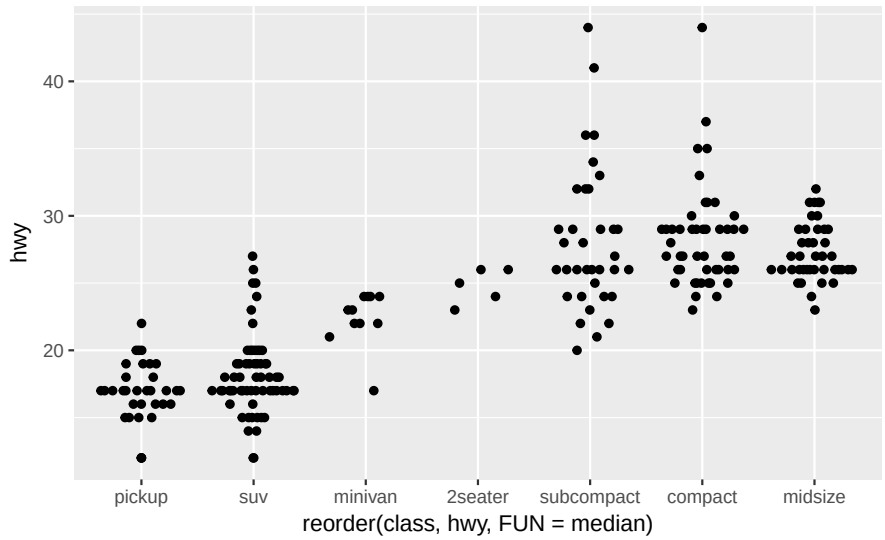
```
library("ggbeeswarm")
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy))
```



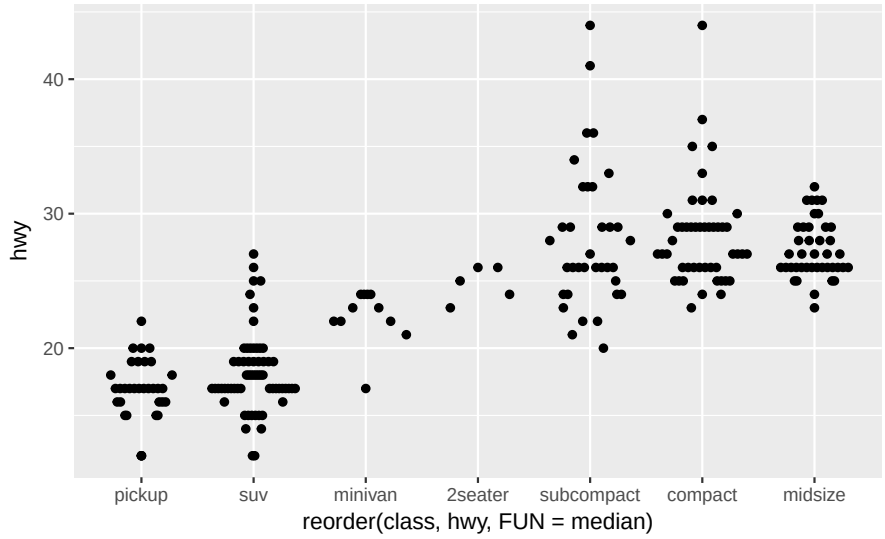

```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy),
                   method = "tukey")
```



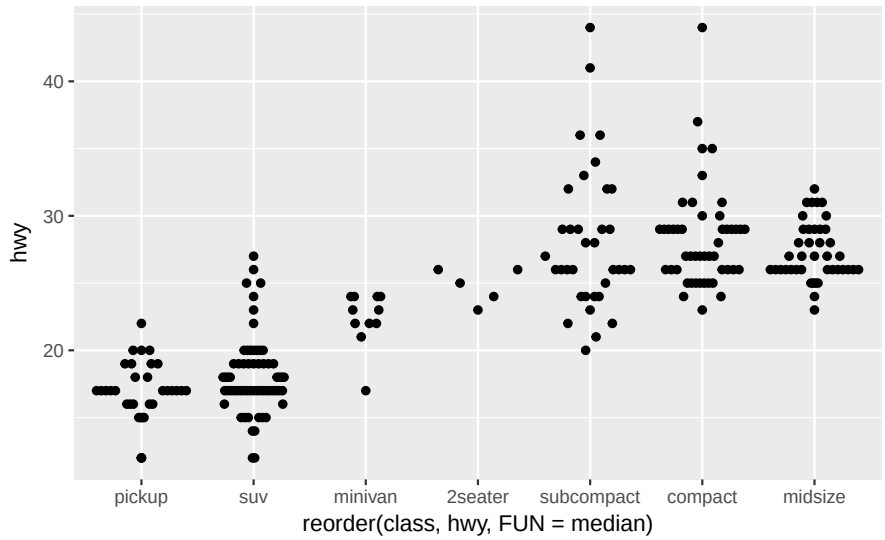

```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy),
                   method = "tukeyDense")
```



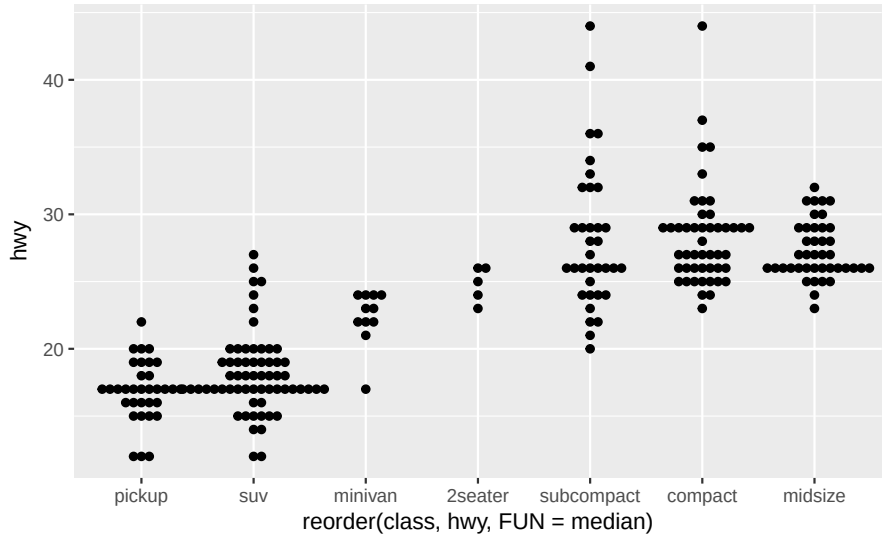

```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                 y = hwy),
                   method = "frowney")
```




```
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

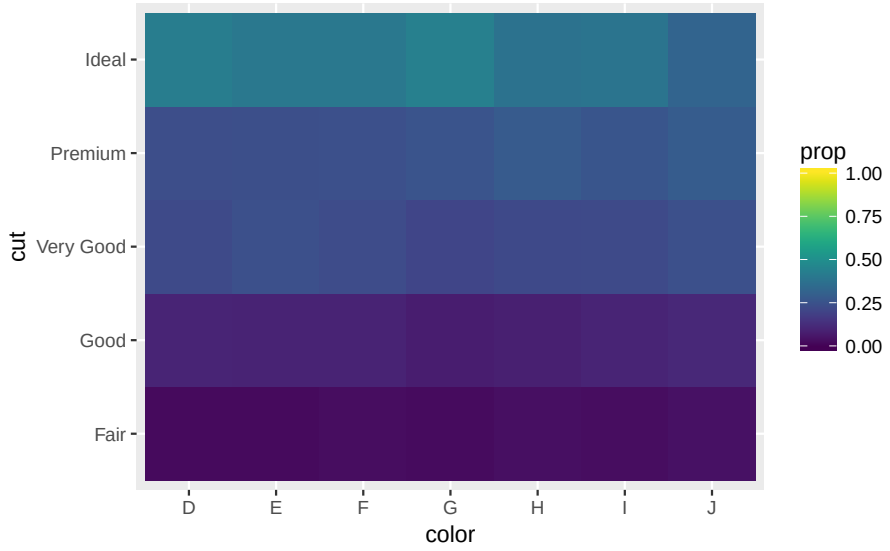
7.5.2.1 Exercise 1

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.

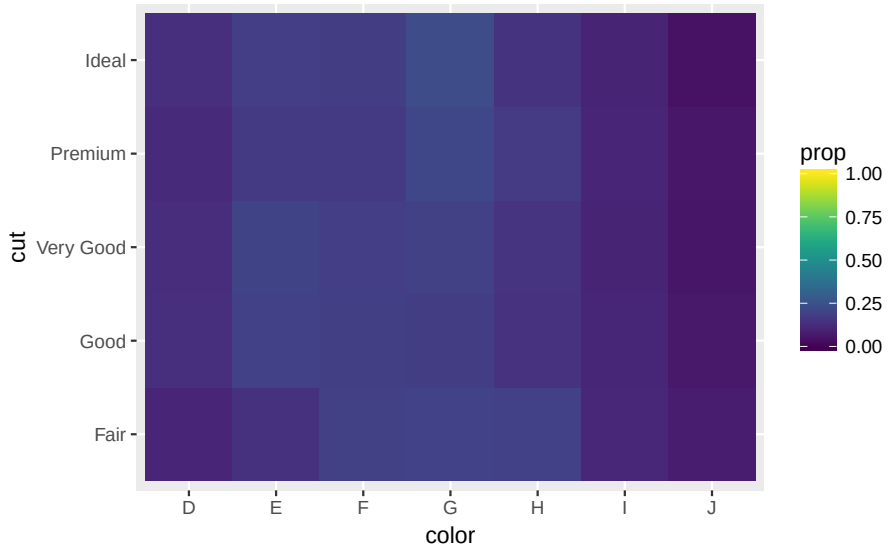
```
diamonds %>%
  count(color, cut) %>%
  group_by(color) %>%
  mutate(prop = n / sum(n)) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
```

```
  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))
```

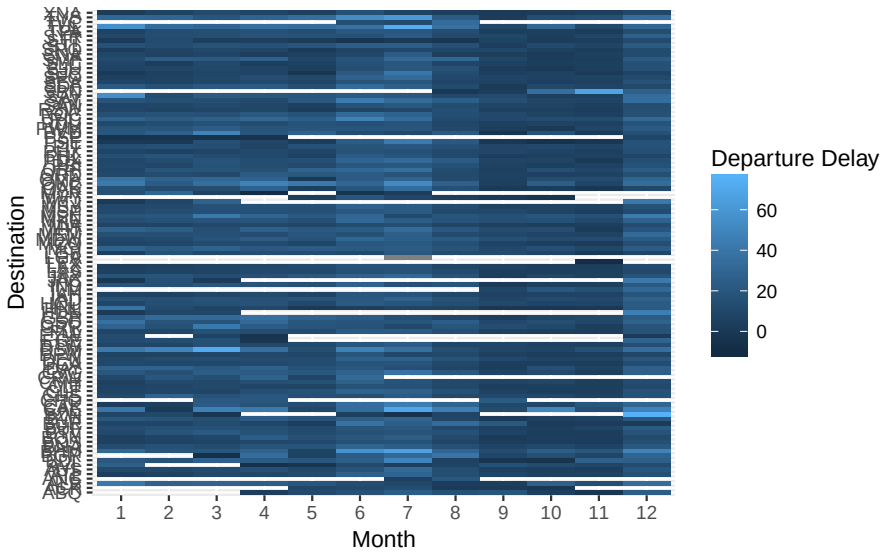


I add `limit = c(0, 1)` to put the color scale between (0, 1). These are the logical boundaries of proportions. This makes it possible to compare each cell to its actual value, and would improve comparisons across multiple plots. However, it ends up limiting the colors and makes it harder to compare 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")
```

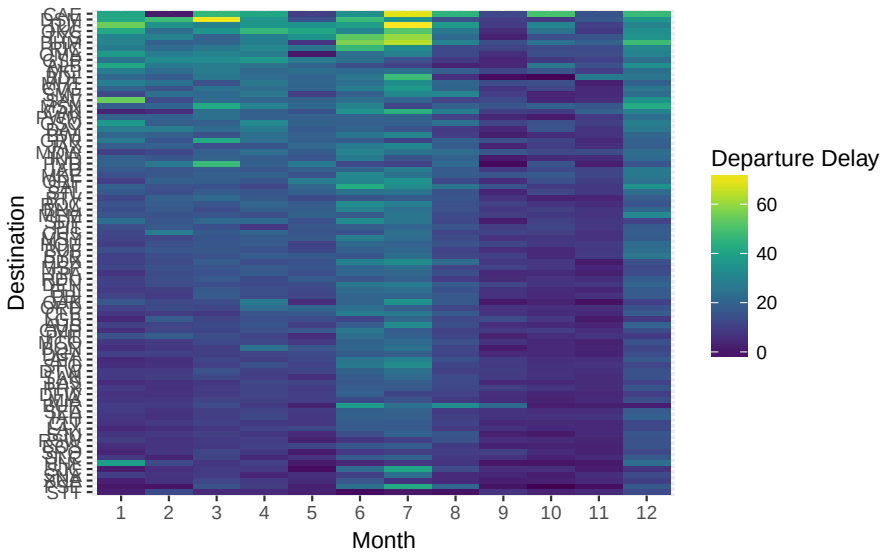


There are several things that could be done to improve it,

- 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")
```



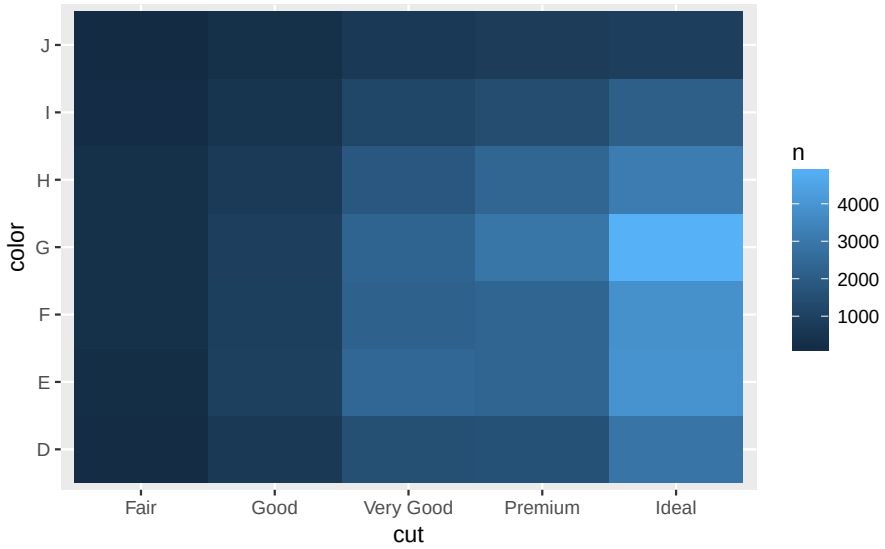
7.5.2.3 Exercise 3

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

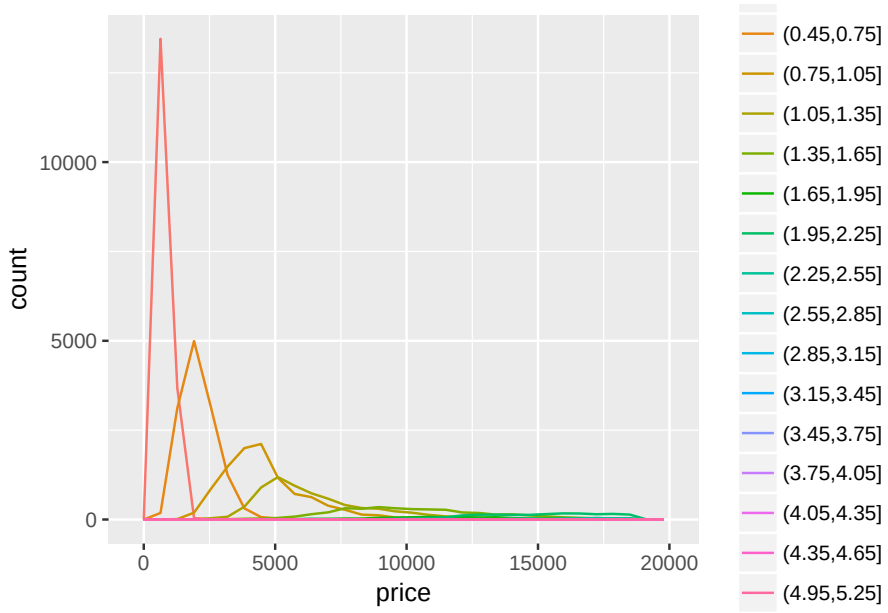
7.5.3.1 Exercise 1

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`?

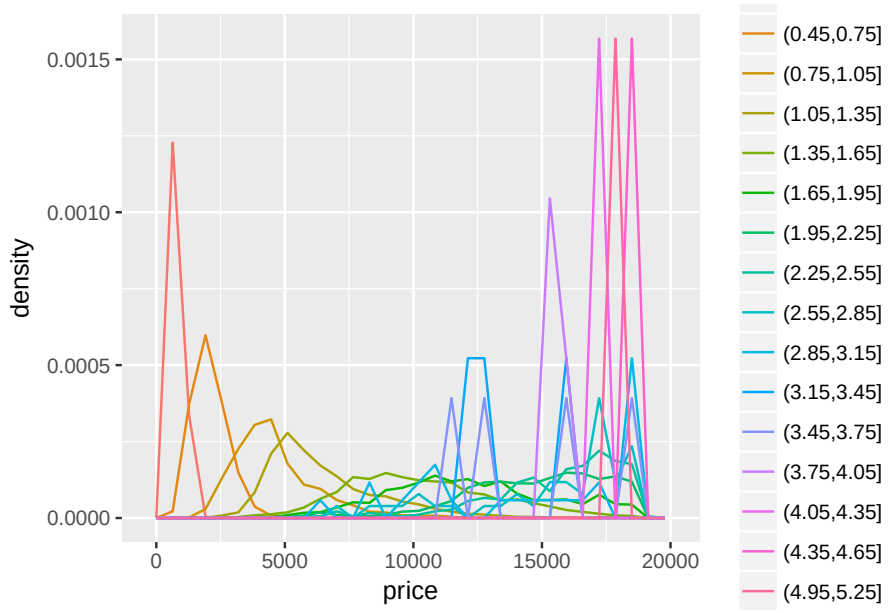
When using `cut_width` the number in each bin may be unequal. The distribution of `carat` is right skewed so there are few diamonds in those bins.

```
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`.
```



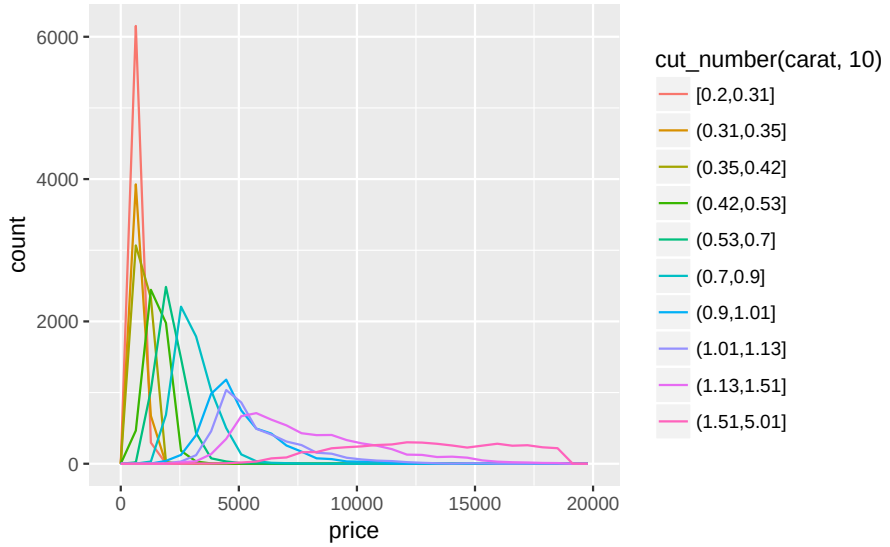
Plotting the density instead of counts will make the distributions comparable, although the bins with few observations will still be hard to interpret.

```
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`.
```



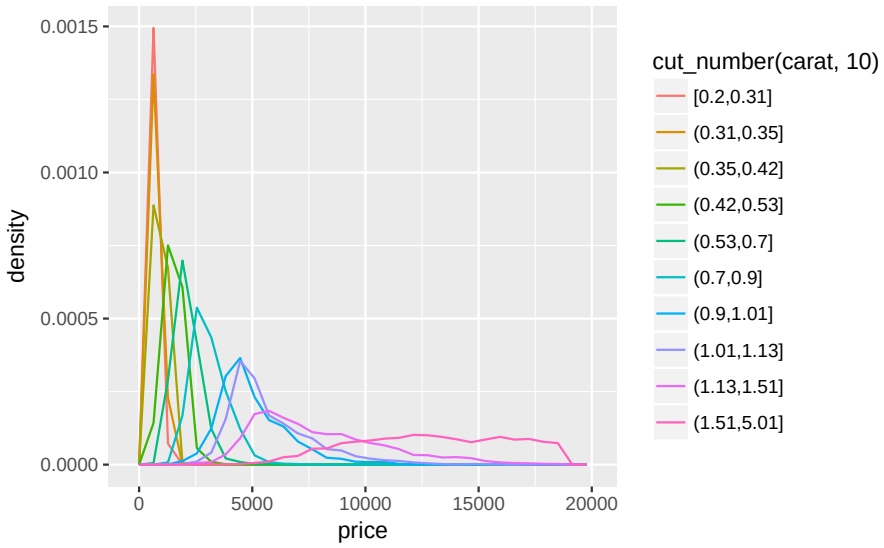
Plotting the density instead of counts will make the distributions comparable, although the bins with few observations will still be hard to interpret.

```
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`.
```

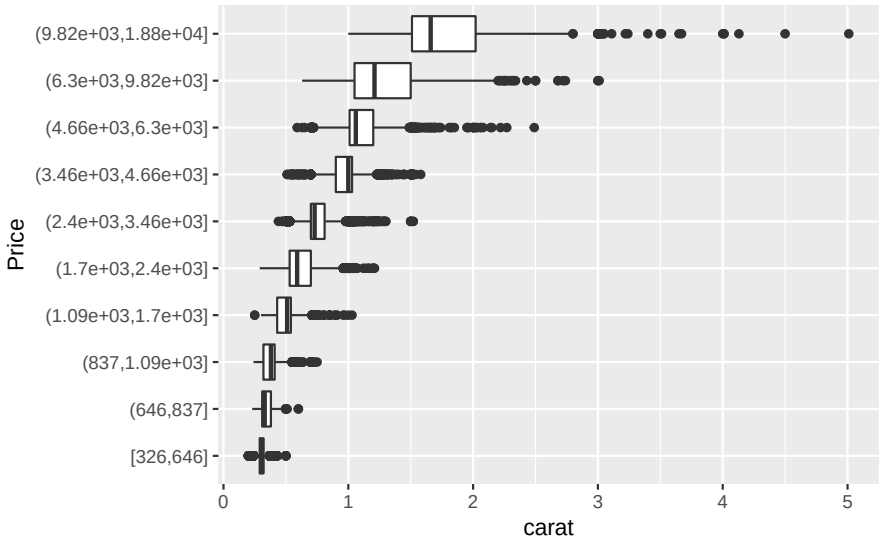


7.5.3.2 Exercise 2

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

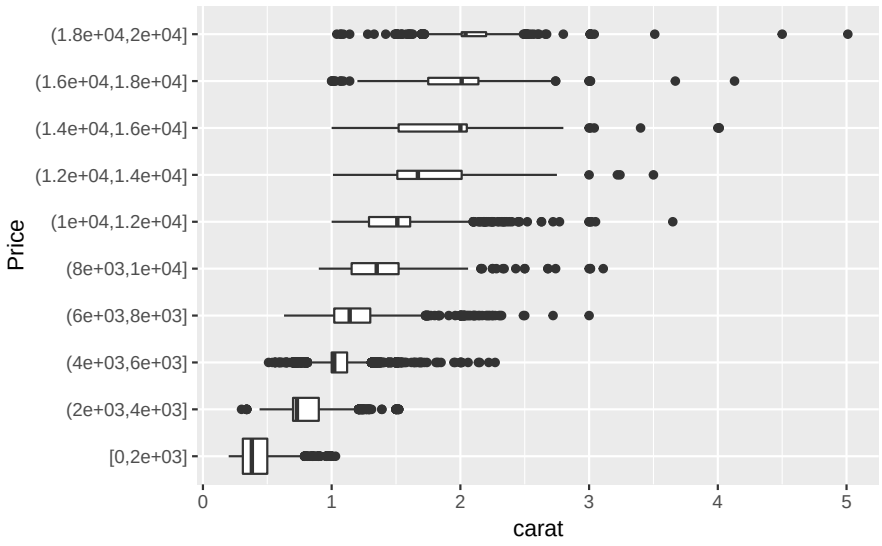
With a box plot, partitioning into an 10 bins with the same number of observations:

```
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?

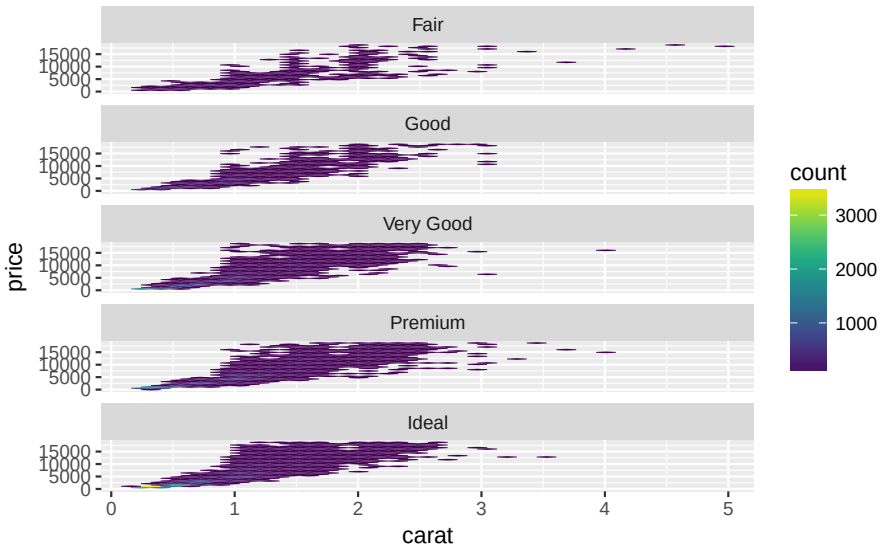
The distribution of very large diamonds is more variable. I'm not surprised, since I had a very weak prior about diamond prices. Ex post, I would reason that above a certain size other factors such as cut, clarity, color play more of a role in the price.

7.5.3.4 Exercise 4

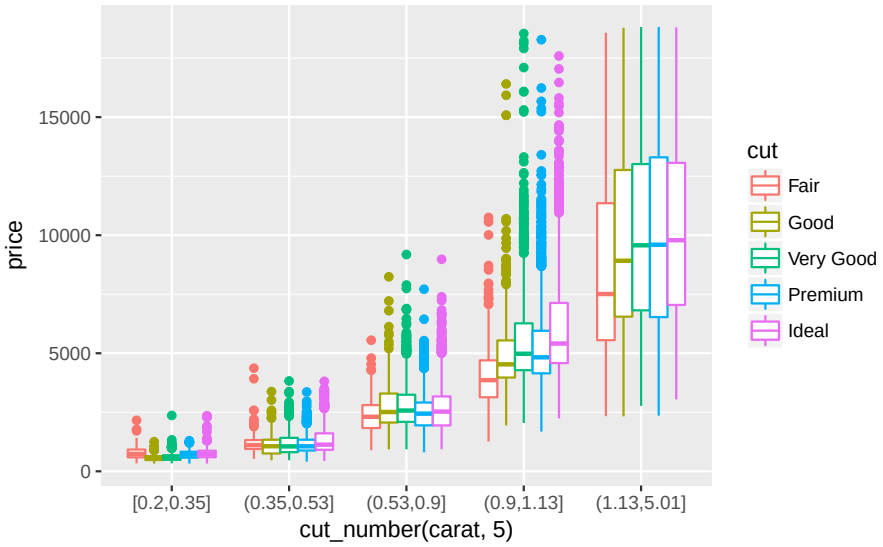
Combine two of the techniques you've learned to visualize the combined distribution of cut, carat, and price.

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

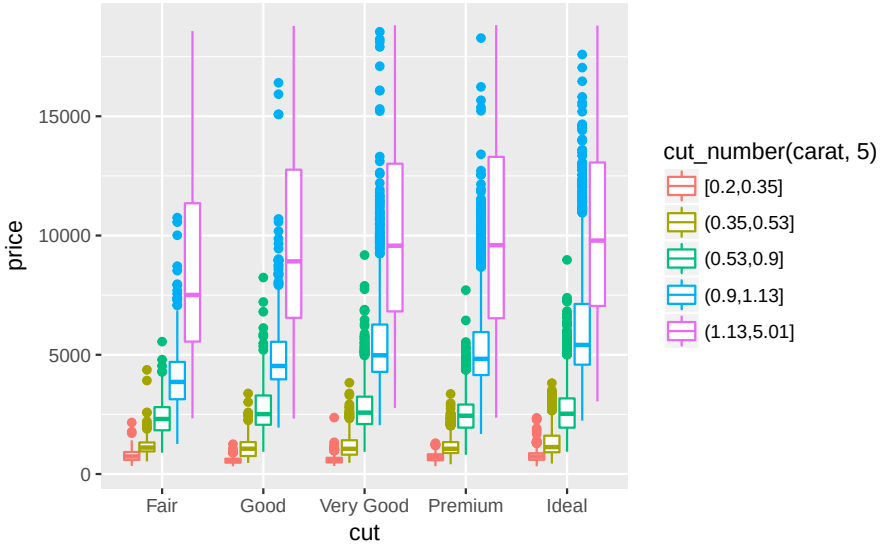
```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_hex() +
  facet_wrap(~ cut, ncol = 1) +
  scale_fill_viridis()
#> Loading required package: methods
```




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



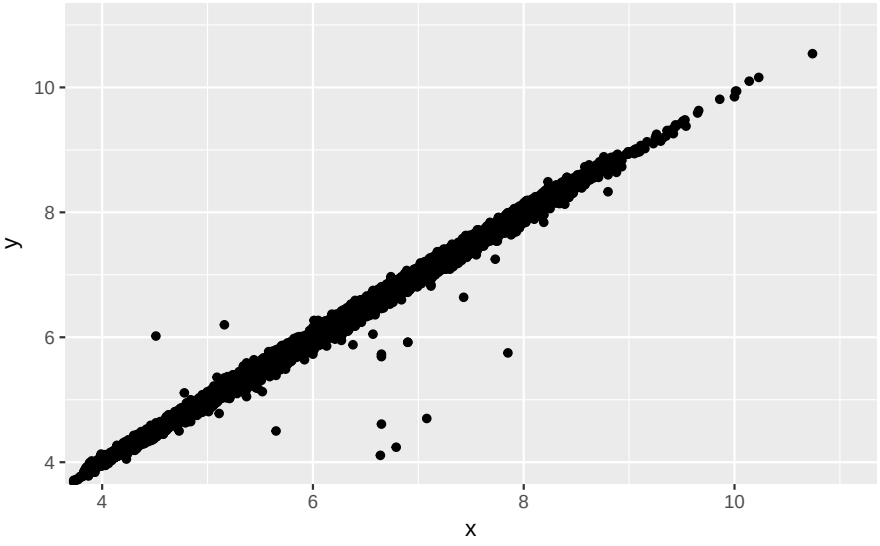

```
ggplot(diamonds, aes(color = cut_number(carat, 5), y = price, x = cut)) +
  geom_boxplot()
```



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. 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:

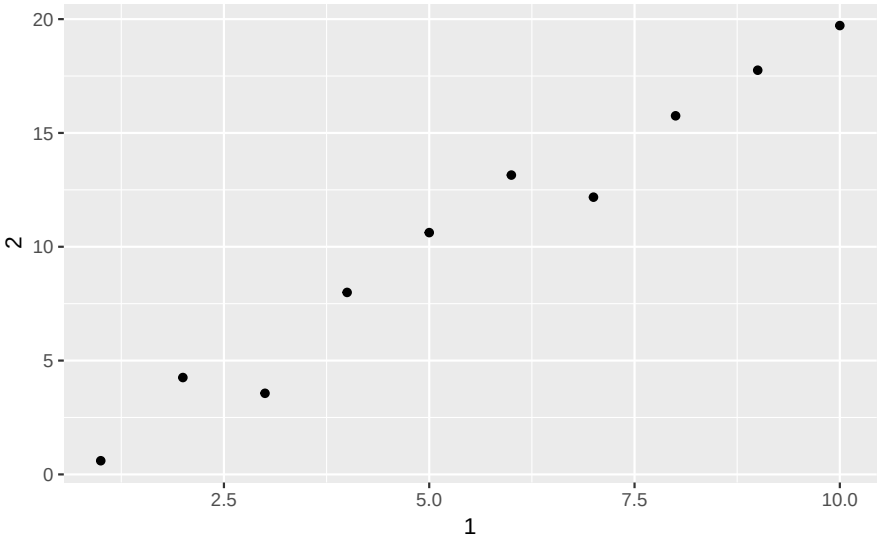
```
annoying[["1"]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

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.
```

```
#> 2 b         2.
#> 3 c         3.
```

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.
#> 2 Brazil      0.000219      1.
#> 3 China       0.000167      1.
```

or

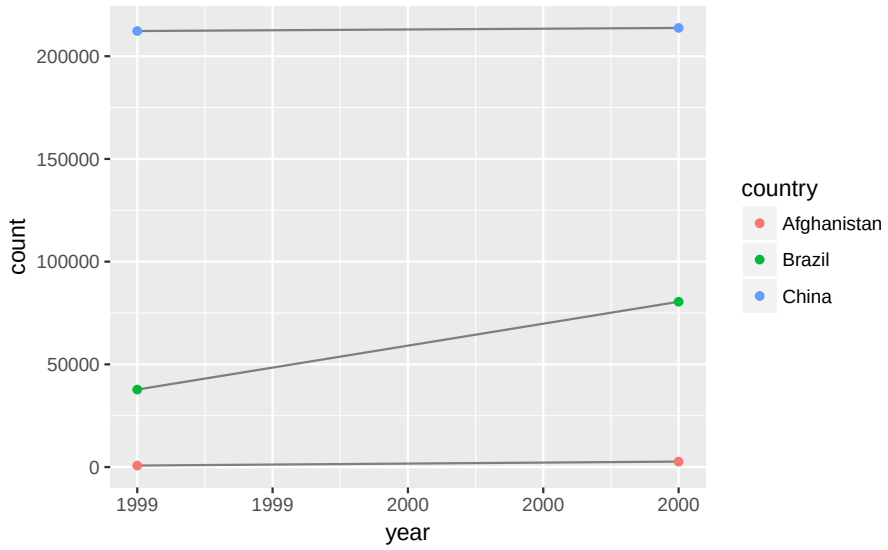
```
tibble(country = rep(table4a[["country"]], 2),
       year = rep(c(1999, 2000), each = nrow(table4a)),
       `rate` = c(table4a[["1999"]] / table4b[["1999"]],
                  table4b[["2000"]] / table4b[["2000"]]))
#> # A tibble: 6 x 3
#>   country      year      rate
#>   <chr>       <dbl>     <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. 2015   1.88
#> 2    2. 2015   0.590
#> 3    1. 2016   0.920
#> 4    2. 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.  2015  1.88
#> 2    2.  2015  0.590
#> 3    1.  2016  0.920
#> 4    2.  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.   37.   156.
#> 2 Phillip Woods      1.   45.   186.
#> 3 Phillip Woods      2.   50.    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 TRUE        NA  FALSE
#> 2 FALSE       20. FALSE
#> 3 TRUE        10. TRUE
#> 4 FALSE       12. TRUE
```

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 FALSE
#> 2 Afghanistan  1998 FALSE
#> 3 Afghanistan  1999 FALSE
#> 4 Afghanistan  2000 FALSE
#> 5 Afghanistan  2001 FALSE
#> 6 Afghanistan  2002 FALSE
#> # ... 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

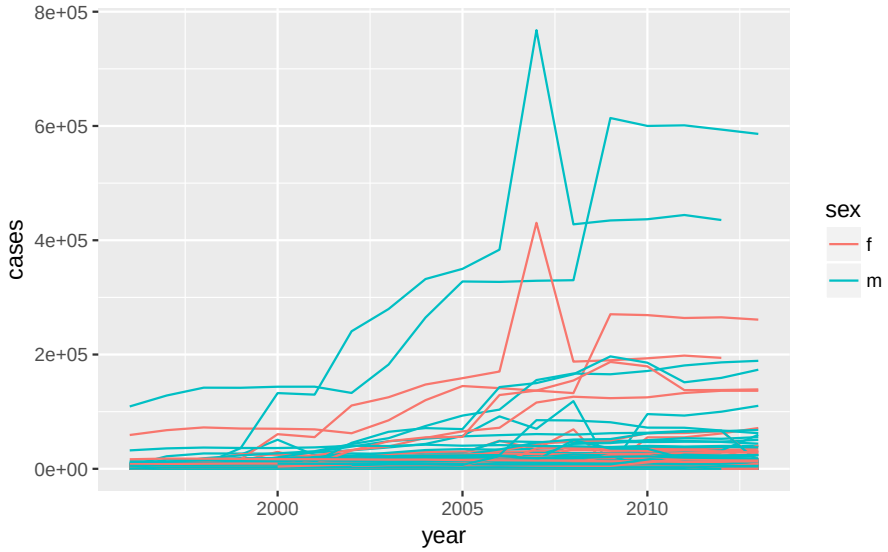
I claimed that `iso2` and `iso3` were redundant with country. Confirm this claim.

```
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.

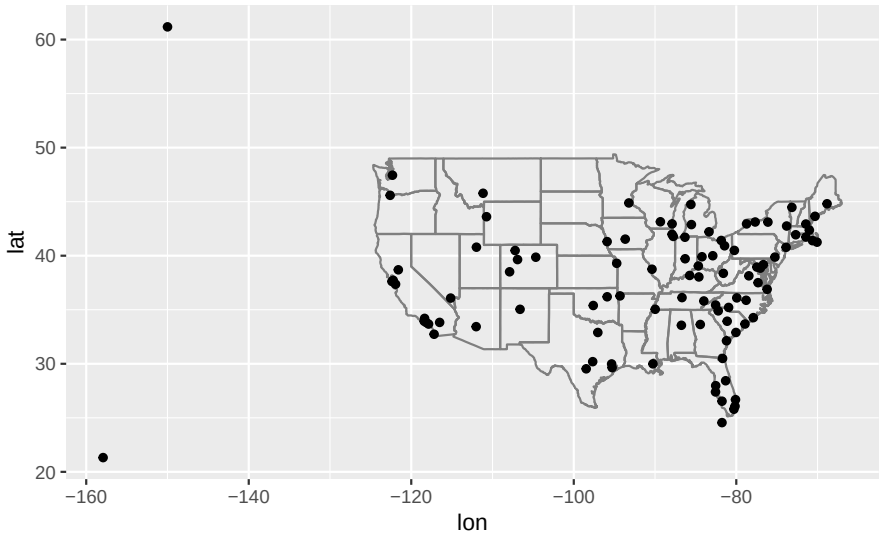
12.4 Mutating Joins

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

12.4.1 Exercise 1

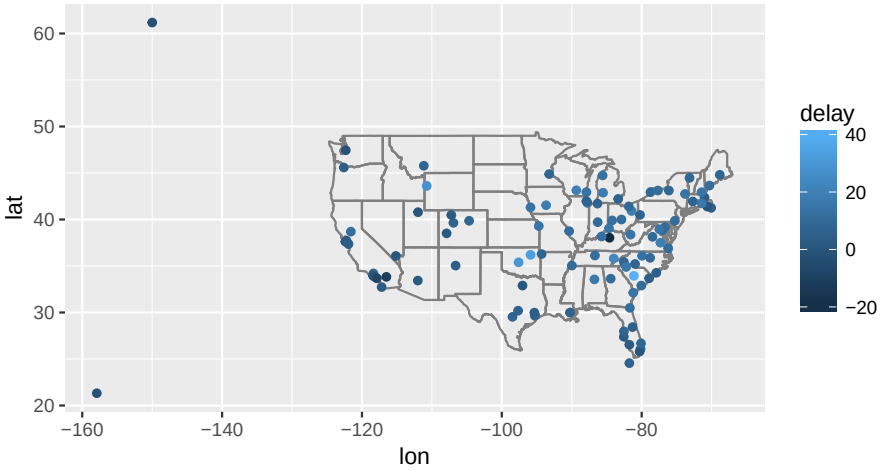
Compute the average delay by destination, then join on the `airports` data frame so you can show the spatial distribution of delays. Here's an easy way to draw a map of the United States:

```
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.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      542            540        2.      923
#> 4  2013     1     1      544            545       -1.     1004
#> 5  2013     1     1      554            600       -6.      812
#> 6  2013     1     1      554            558       -4.      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>
```

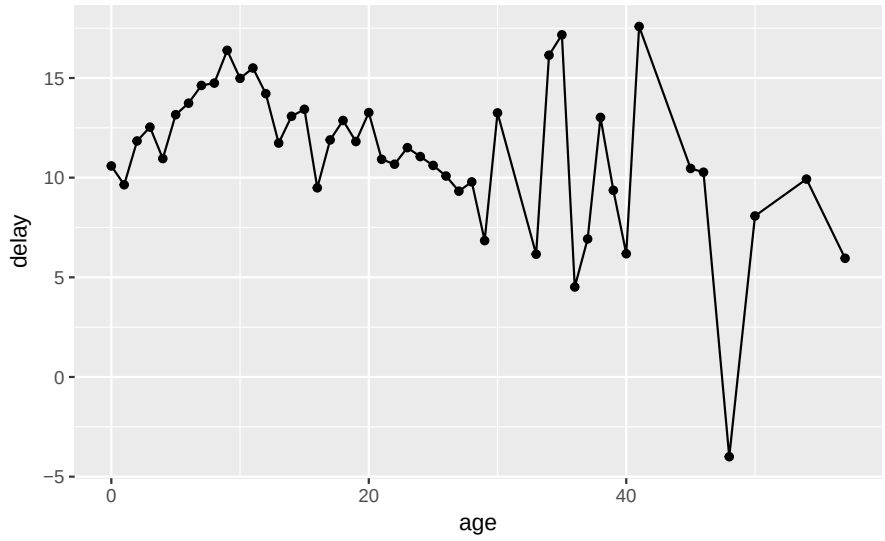
12.4.3 Exercise 3

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).
```

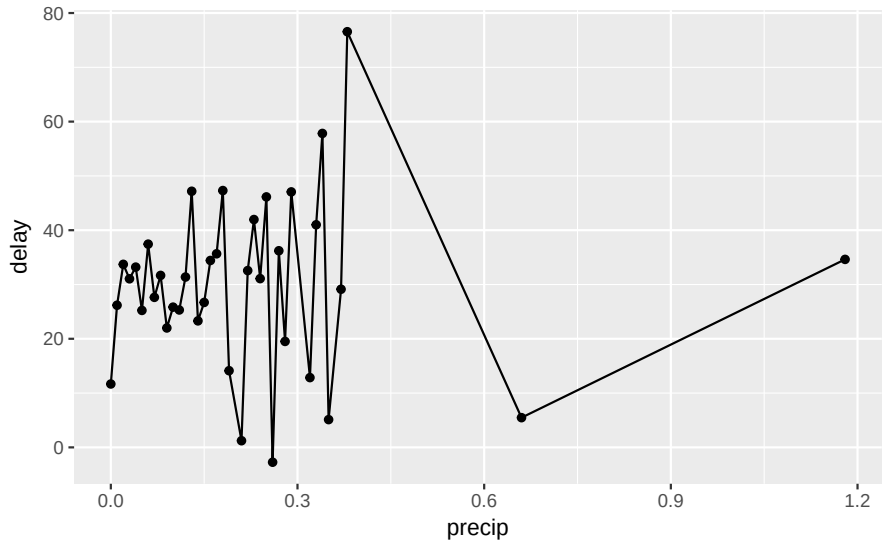


12.4.4 Exercise 4

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()
```



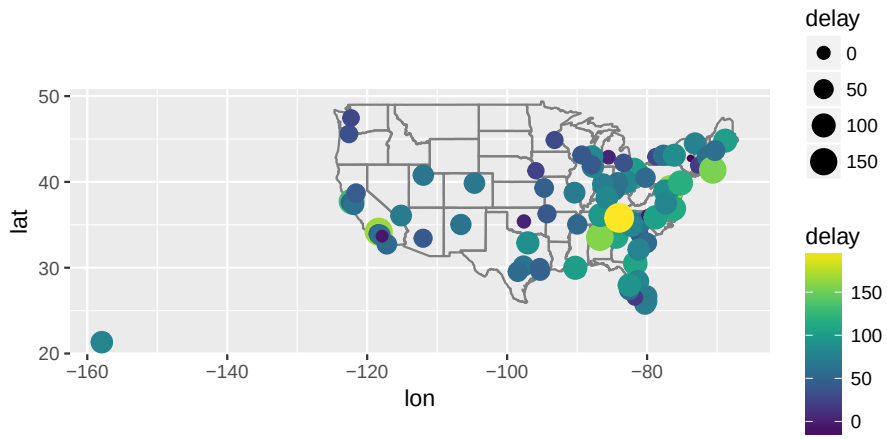
12.4.5 Exercise 5

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.      830
#> 2  2013     1     1      533            529        4.      850
#> 3  2013     1     1      544            545       -1.     1004
#> 4  2013     1     1      554            558       -4.      740
#> 5  2013     1     1      555            600       -5.      913
#> 6  2013     1     1      557            600       -3.      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("\"'\\", "\"'\\\\")
```

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")
```

```
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"), "^\\$\\^\\$$")
```

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)
```

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

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

A simpler way, shown later is

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

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

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)
```

Words that contain only consonants

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

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)
```

Words ending in `ing` or `ise`:

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

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)
```

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

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 au`"?

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

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

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")
```

or

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

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}")
```

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", ".{,}")
```

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")`
- `"\\{.+\\}"`: `c("{a}", "{abc}")`
- `\d{4}-\d{2}-\d{2}`: `2018-01-11`
- `"\\\\{4}"`: `"\\\\\\\\"` (Backslashes in an R character vector need to be escaped.)

13.3.4.3 Exercise 3

Create regular expressions to find all words that:

1. Start with three consonants.
2. Have three or more vowels in a row.
3. Have two or more vowel-consonant pairs in a row.

A regex to find all words starting with three consonants

```
str_view(words, "^[^aeiou]{3}")
```

A regex to find three or more vowels in a row:

```
str_view(words, "[aeiou]{3,}")
```

Two or more vowel-consonant pairs in a row.

```
str_view(words, "([aeiou][^aeiou]){2,}")
```

13.3.4.4 Exercise 4

Solve the beginner regexp crosswords at https://regexcrossword.com/challenges/

Exercise left to reader. That site validates its solutions, so they aren't repeated here.

13.3.5 Grouping and backreferences

13.3.5.1 Exercise 1

Describe, in words, what these expressions will match:

1. `(.)\1\1` :
2. `"(.)(.)\\2\\1"`:
3. `(..)\1`: Any two characters repeated. E.g. `"a1a1"`.
4. `"(.).\\1.\\1"`:
5. `"(.)(.)(.).*\\3\\2\\1"`

1. `(.)\1\1` : The same character appearing three times in a row. E.g. `"aaa"`
2. `"(.)(.)\\2\\1"`: A pair of characters followed by the same pair of characters in reversed order. E.g. `"abba"`.
3. `(..)\1`: Any two characters repeated. E.g. `"a1a1"`.
4. `"(.).\\1.\\1"`: A character followed by any character, the original character, any other character, the original character again. E.g. `"abaca"`, `"b8b.b"`.
5. `"(.)(.)(.).*\\3\\2\\1"` Three characters followed by zero or more characters of any kind followed by the same three characters but in reverse order. E.g. `"abcsgasgddsadgsdgcba"` or `"abccba"` or `"abc1cba"`.

13.3.5.2 Exercise 2

Construct regular expressions to match words that:

1. Start and end with the same character. Assuming the word is more than one character and all strings are considered words, `^(.).*\1$`
2. Contain a repeated pair of letters (e.g. `church'' containsch`" repeated twice.)
3. Contain one letter repeated in at least three places (e.g. `eleven'' contains threee`"s.)

Start and end with the same character. Assuming the word is more than one character and all strings are considered words, `^(.).*\1$`:

```
str_view(words, "^(.).*\\1$")
```

Contain a repeated pair of letters (e.g. "church" contains "ch" repeated twice.):

I'll consider several patterns with varying levels of generality. This pattern checks for any pair of repeated characters:

```
str_view(words, "(..).*\\1")
```

This pattern is checks for any pair of repeated "letters":

```
str_view(words, "([A-Za-z][A-Za-z]).*\\1")
```

Contain one letter repeated in at least three places (e.g. `eleven'' contains threee`"s.)

```
str_subset(str_to_lower(words), "([a-z]).*\\1.*\\1")
#>  [1] "appropriate" "available"   "believe"     "between"     "business"
#>  [6] "degree"      "difference"  "discuss"     "eleven"      "environment"
#> [11] "evidence"    "exercise"    "expense"     "experience"  "individual"
#> [16] "paragraph"   "receive"     "remember"    "represent"   "telephone"
#> [21] "therefore"   "tomorrow"
```

The `\\1` is used to refer back to the first group (`(.)`) so that whatever letter is matched by `[A-Za-z]` is again matched.

13.4 Tools

13.4.1 Detect matches

No exercises

13.4.2 Exercises

13.4.2.1 Exercise 1

For each of the following challenges, try solving it by using both a single regular expression, and a combination of multiple `str_detect()` calls.

1. Find all words that start or end with x.
2. Find all words that start with a vowel and end with a consonant.
3. Are there any words that contain at least one of each different vowel?

Words that start or end with `x`?

```


# one regex
```

```
words[str_detect(words, "^x|x$")]
#> [1] "box" "sex" "six" "tax"
# split regex into parts
start_with_x <- str_detect(words, "^x")
end_with_x <- str_detect(words, "x$")
words[start_with_x | end_with_x]
#> [1] "box" "sex" "six" "tax"
```

Find all words starting with vowel and ending with consonant.

```
str_subset(words, "^[aeiou].*[^aeiou]$") %>% head()
#> [1] "about"   "accept"  "account" "across"  "act"     "actual"
start_with_vowel <- str_detect(words, "^[aeiou]")
end_with_consonant <- str_detect(words, "[^aeiou]$")
words[start_with_vowel & end_with_consonant] %>% head()
#> [1] "about"   "accept"  "account" "across"  "act"     "actual"
```

Words that contain at least one of each vowel. I can't think of a good way of doing this without doing a regex of the permutations:

```
pattern <-
  cross_n(rerun(5, c("a", "e", "i", "o", "u")),
        .filter = function(...) {
          x <- as.character(unlist(list(...)))
          length(x) != length(unique(x))
        }) %>%
  map_chr(~ str_c(unlist(.x), collapse = ".*")) %>%
  str_c(collapse = "|")
#> Warning: `cross_n()` is deprecated; please use `cross()` instead.

str_subset(words, pattern)
#> character(0)

words[str_detect(words, "a") &
        str_detect(words, "e") &
        str_detect(words, "i") &
        str_detect(words, "o") &
        str_detect(words, "u")]
#> character(0)
```

There appear to be none. To check that it works,

```
str_subset("aseiouds", pattern)
#> [1] "aseiouds"
```

13.4.2.2 Exercise 2

What word has the highest number of vowels? What word has the highest proportion of vowels? (Hint: what is the denominator?)

```
prop_vowels <- str_count(words, "[aeiou]") / str_length(words)
words[which(prop_vowels == max(prop_vowels))]
#> [1] "a"
```

13.4.3 Extract Matches

13.4.3.1 Exercise 1

In the previous example, you might have noticed that the regular expression matched "flickered", which is not a color. Modify the regex to fix the problem.

This was the original color match pattern:

```
colours <- c("red", "orange", "yellow", "green", "blue", "purple")
colour_match <- str_c(colours, collapse = "|")
```

It matches "flickered" because it matches "red". The problem is that the previous pattern will match any word with the name of a color inside it. We want to only match colors in which the entire word is the name of the color. We can do this by adding a `\b` (to indicate a word boundary) before and after the pattern:

```
colour_match2 <- str_c("\\b(", str_c(colours, collapse = "|"), ")\\b")
colour_match2
#> [1] "\\b(red|orange|yellow|green|blue|purple)\\b"
```

```
more2 <- sentences[str_count(sentences, colour_match) > 1]
```

```
str_view_all(more2, colour_match2, match = TRUE)
```

13.4.3.2 Exercise 2

From the Harvard sentences data, extract:

1. The first word from each sentence.
2. All words ending in `ing`.
3. All plurals.

The first word in each sentence requires defining what a word is. I'll consider a word any contiguous

```
str_extract(sentences, "[a-zA-X]+") %>% head()
#> [1] "The"   "Glue"  "It"    "These" "Rice"  "The"
```

All words ending in `ing`:

```
pattern <- "\\b[A-Za-z]+ing\\b"
sentences_with_ing <- str_detect(sentences, pattern)
unique(unlist(str_extract_all(sentences[sentences_with_ing], pattern))) %>%
  head()
#> [1] "spring"  "evening" "morning" "winding" "living"  "king"
```

All plurals. To do this correct requires linguistic information. But if we just want to say any word ending in an "s" is plural (and with more than 3 characters to remove as, is, gas, etc.)

```
unique(unlist(str_extract_all(sentences, "\\b[A-Za-z]{3,}s\\b"))) %>%
  head()
#> [1] "planks" "days"   "bowls"  "lemons" "makes"  "hogs"
```

13.4.4 Grouped Matches

13.4.4.1 Exercise 1

Find all words that come after a "number" like "one", "two", "three" etc. Pull out both the number and the word.

I'll use the same following "word" pattern as used above

```
numword <- "(one|two|three|four|five|six|seven|eight|nine|ten) +(\\S+)"
sentences[str_detect(sentences, numword)] %>%
  str_extract(numword)
#>  [1] "ten served"    "one over"      "seven books"   "two met"
#>  [5] "two factors"   "one and"       "three lists"   "seven is"
#>  [9] "two when"      "one floor."    "ten inches."   "one with"
#> [13] "one war"       "one button"    "six minutes."  "ten years"
#> [17] "one in"        "ten chased"    "one like"      "two shares"
#> [21] "two distinct"  "one costs"     "ten two"       "five robins."
#> [25] "four kinds"    "one rang"      "ten him."      "three story"
#> [29] "ten by"        "one wall."     "three inches"  "ten your"
#> [33] "six comes"     "one before"    "three batches" "two leaves."
```

13.4.4.2 Exercise 2

Find all contractions. Separate out the pieces before and after the apostrophe.

```
contraction <- "([A-Za-z]+)'([A-Za-z]+)"
sentences %>%
  `[`(str_detect(sentences, contraction)) %>%
  str_extract(contraction)
#>  [1] "It's"       "man's"      "don't"      "store's"    "workmen's"
#>  [6] "Let's"      "sun's"      "child's"    "king's"     "It's"
#> [11] "don't"      "queen's"    "don't"      "pirate's"   "neighbor's"
```

13.4.5 Replacing Matches

13.4.5.1 Exercise 1

Replace all forward slashes in a string with backslashes.

```
backslashed <- str_replace_all("past/present/future", "\\/", "\\\\")
writeLines(backslashed)
#> past\present\future
```

13.4.5.2 Exercise 2

Implement a simple version of `str_to_lower()` using `replace_all()`.

```
lower <- str_replace_all(words, c("A"="a", "B"="b", "C"="c", "D"="d", "E"="e", "F"="f", "G"="g", "H"="h
```

13.4.5.3 Exercise 3

Switch the first and last letters in `words`. Which of those strings are still words?

First, make a vector of all the words with first and last letters swapped,

```
swapped <- str_replace_all(words, "^([A-Za-z])(.*)([a-z])$", "\\3\\2\\1")
```

Next, find what of "swapped" is also in the original list using the function `intersect`,

```
intersect(swapped,words)
#>  [1] "a"          "america"    "area"       "dad"        "dead"
#>  [6] "lead"       "read"       "depend"     "god"        "educate"
#> [11] "else"       "encourage"  "engine"     "europe"     "evidence"
#> [16] "example"    "excuse"     "exercise"   "expense"    "experience"
#> [21] "eye"        "dog"        "health"     "high"       "knock"
#> [26] "deal"       "level"      "local"      "nation"     "on"
#> [31] "non"        "no"         "rather"     "dear"       "refer"
#> [36] "remember"   "serious"    "stairs"     "test"       "tonight"
#> [41] "transport"  "treat"      "trust"      "window"     "yesterday"
```

13.4.6 Splitting

13.4.6.1 Exercise 1

Split up a string like `"apples, pears, and bananas"` into individual components.

```
x <- c("apples, pears, and bananas")
str_split(x, ", +(and +)?")[[1]]
#> [1] "apples"  "pears"   "bananas"
```

13.4.6.2 Exercise 2

Why is it better to split up by `boundary("word")` than `" "`?

Splitting by `boundary("word")` splits on punctuation and not just whitespace.

13.4.6.3 Exercise 3

What does splitting with an empty string (`""`) do? Experiment, and then read the documentation.

```
str_split("ab. cd|agt", "")[[1]]
#>  [1] "a" "b" "." " " "c" "d" "|" "a" "g" "t"
```

It splits the string into individual characters.

13.4.7 Find matches

No exercises

13.5 Other types of patterns

13.5.1 Exercise 1

How would you find all strings containing `\` with `regex()` vs. with `fixed()`?

```
str_subset(c("a\\b", "ab"), "\\\\")
#> [1] "a\\b"
str_subset(c("a\\b", "ab"), fixed("\\"))
#> [1] "a\\b"
```

13.5.2 Exercise 2

What are the five most common words in sentences?

```
str_extract_all(sentences, boundary("word")) %>%
  unlist() %>%
  str_to_lower() %>%
  tibble() %>%
  set_names("word") %>%
  group_by(word) %>%
  count(sort = TRUE) %>%
  head(5)
#> # A tibble: 5 x 2
#> # Groups:   word [5]
#>   word      n
#>   <chr> <int>
#> 1 the     751
#> 2 a       202
#> 3 of      132
#> 4 to      123
#> 5 and     118
```

13.6 Other uses of regular expressions

No exercises

13.7 stringi

13.7.1 Exercise 1

Find the **stringi** functions that:

1. Count the number of words. `stri_count_words`
2. Find duplicated strings. `stri_duplicated`
3. Generate random text. There are several functions beginning with `stri_rand_`. `stri_rand_lipsum` generates lorem ipsum text, `stri_rand_strings` generates random strings, `stri_rand_shuffle` randomly shuffles the code points in the text.

1. Count the number of words. `stri_count_words`
2. Find duplicated strings. `stri_duplicated`
3. Generate random text. There are several functions beginning with `stri_rand_`. `stri_rand_lipsum` generates lorem ipsum text, `stri_rand_strings` generates random strings, `stri_rand_shuffle` randomly shuffles the code points in the text.

13.7.2 Exercise 2

How do you control the language that `stri_sort()` uses for sorting?

Use the `locale` argument to the `opts_collator` argument.

Chapter 14

Factors

14.1 Introduction

Functions and packages:

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

14.2 Creating Factors

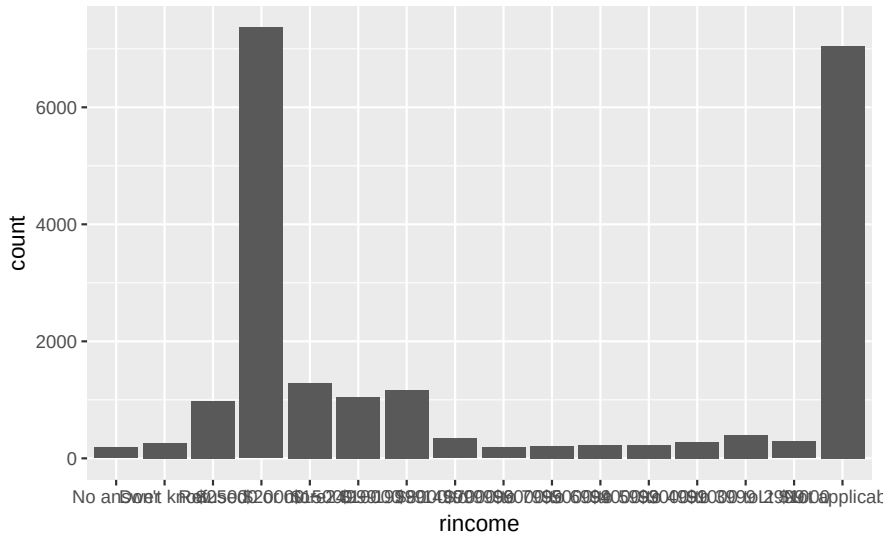
No exercises

14.3 General Social Survey

14.3.1 Exercise 1

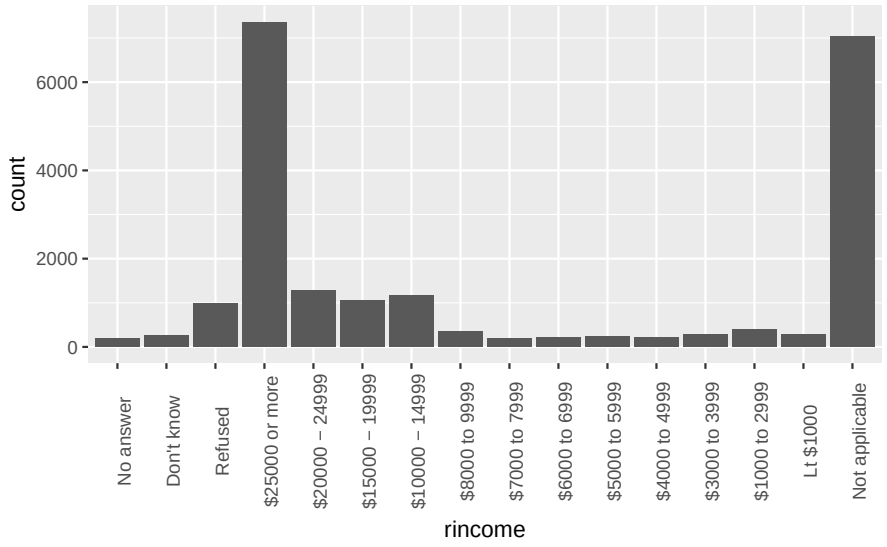
Explore the distribution of `rincome` (reported income). What makes the default bar chart hard to understand? How could you improve the plot?

```
rincome_plot <-
  gss_cat %>%
  ggplot(aes(rincome)) +
  geom_bar()
rincome_plot
```



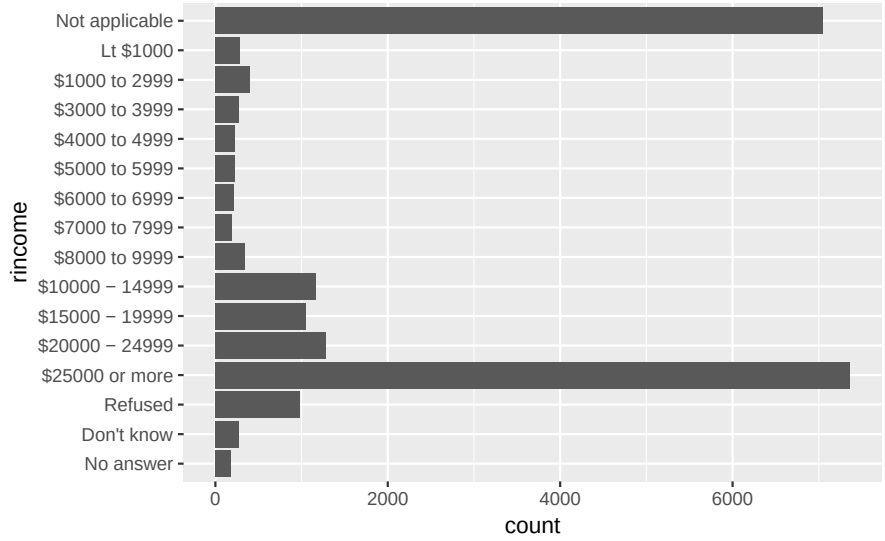
The default bar chart labels are too squished to read. One solution is to change the angle of the labels,

```
rincome_plot +
  theme(axis.text.x = element_text(angle = 90))
```



But that's not natural either, because text is vertical, and we read horizontally. So with long labels, it is better to flip it.

```
rincome_plot +
  coord_flip()
```



This is better, but it unintuitively goes from low to high. It would help if the scale is reversed. Also, if all the missing factors were differentiated.

14.3.2 Exercise 2

What is the most common `relig` in this survey? What's the most common `partyid`?

The most common `relig` is "Protestant"

```
gss_cat %>%
  count(relig) %>%
  arrange(-n) %>%
  head(1)
#> # A tibble: 1 x 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant 10846
```

The most common `partyid` is "Independent"

```
gss_cat %>%
  count(partyid) %>%
  arrange(-n) %>%
  head(1)
#> # A tibble: 1 x 2
#>   partyid         n
#>   <fct>       <int>
#> 1 Independent  4119
```

14.3.3 Exercise 4

Which `relig` does `denom` (denomination) apply to? How can you find out with a table? How can you find out with a visualization?

```
levels(gss_cat$denom)
#>  [1] "No answer"            "Don't know"           "No denomination"
#>  [4] "Other"                "Episcopal"            "Presbyterian-dk wh"
```

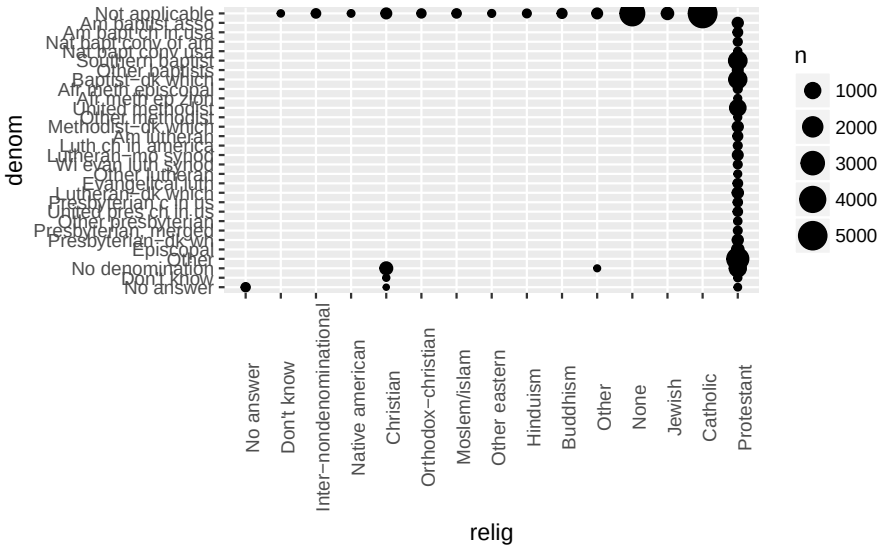
```
#>  [7] "Presbyterian, merged" "Other presbyterian"   "United pres ch in us"
#> [10] "Presbyterian c in us" "Lutheran-dk which"    "Evangelical luth"
#> [13] "Other lutheran"       "Wi evan luth synod"   "Lutheran-mo synod"
#> [16] "Luth ch in america"   "Am lutheran"          "Methodist-dk which"
#> [19] "Other methodist"      "United methodist"     "Afr meth ep zion"
#> [22] "Afr meth episcopal"   "Baptist-dk which"     "Other baptists"
#> [25] "Southern baptist"     "Nat bapt conv usa"    "Nat bapt conv of am"
#> [28] "Am bapt ch in usa"    "Am baptist asso"      "Not applicable"
```

From the context it is clear that `denom` refers to "Protestant" (and unsurprising given that it is the largest category in `freq`). Let's filter out the non-responses, no answers, others, not-applicable, or no denomination, to leave only answers to denominations. After doing that, the only remaining responses are "Protestant".

```
gss_cat %>%
  filter(!denom %in% c("No answer", "Other", "Don't know", "Not applicable",
                       "No denomination")) %>%
  count(relig)
#> # A tibble: 1 x 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant  7025
```

This is also clear in a scatter plot of `relig` vs. `denom` where the points are proportional to the size of the number of answers (since otherwise there would be overplotting).

```
gss_cat %>%
  count(relig, denom) %>%
  ggplot(aes(x = relig, y = denom, size = n)) +
  geom_point() +
  theme(axis.text.x = element_text(angle = 90))
```



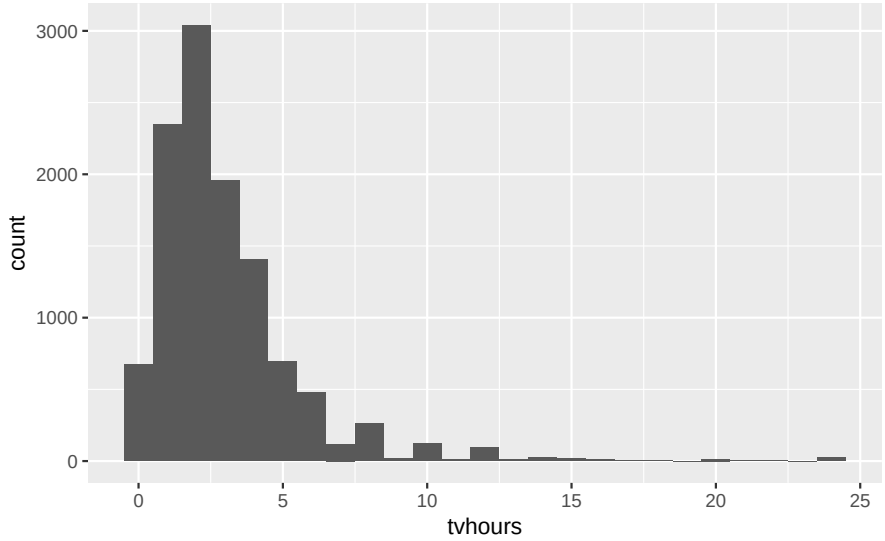
14.4 Modifying factor order

14.4.1 Exercise 1

There are some suspiciously high numbers in `tvhours`. Is the `mean` a good summary?

```
summary(gss_cat[["tvhours"]])
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#>       0       1       2       3       4      24   10146
```

```
gss_cat %>%
  filter(!is.na(tvhours)) %>%
  ggplot(aes(x = tvhours)) +
  geom_histogram(binwidth = 1)
```



Whether the mean is the best summary depends on what you are using it for :-), i.e. your objective. But probably the median would be what most people prefer. And the hours of TV doesn't look that surprising to me.

14.4.2 Exercise 2

For each factor in `gss_cat` identify whether the order of the levels is arbitrary or principled.

The following piece of code uses functions introduced in Ch 21, to print out the names of only the factors.

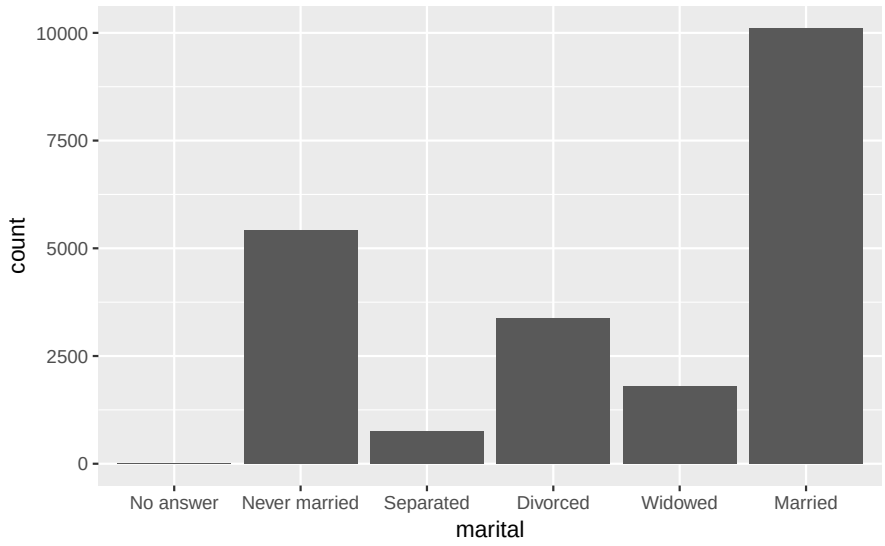
```
keep(gss_cat, is.factor) %>% names()
#> [1] "marital" "race"    "rincome" "partyid" "relig"   "denom"
```

There are five six categorical variables: `marital`, `race`, `rincome`, `partyid`, `relig`, `denom`.

The ordering of marital is "somewhat principled". There is some sort of logic in that the levels are grouped "never married", married at some point (separated, divorced, widowed), and "married"; though it would seem that "Never Married", "Divorced", "Widowed", "Separated", "Married" might be more natural. I find that the question of ordering can be determined by the level of aggregation in a categorical variable, and there can be more "partially ordered" factors than one would expect.

```
levels(gss_cat[["marital"]])
#> [1] "No answer"     "Never married" "Separated"     "Divorced"
#> [5] "Widowed"       "Married"
```

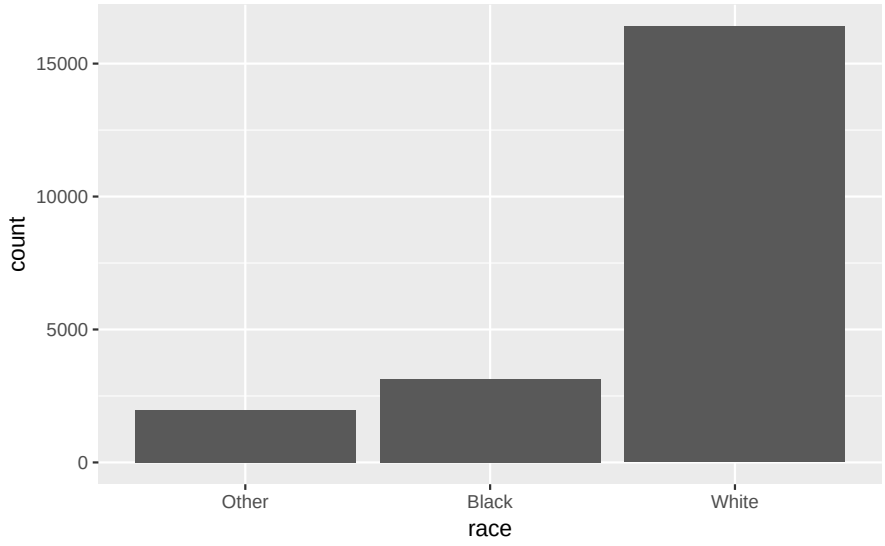
```
gss_cat %>%
  ggplot(aes(x = marital)) +
  geom_bar()
```



The ordering of race is principled in that the categories are ordered by count of observations in the data.

```
levels(gss_cat$race)
#> [1] "Other"          "Black"          "White"          "Not applicable"
```

```
gss_cat %>%
  ggplot(aes(race)) +
  geom_bar(drop = FALSE)
#> Warning: Ignoring unknown parameters: drop
```



The levels of `rincome` are ordered in decreasing order of the income; however the placement of "No answer", "Don't know", and "Refused" before, and "Not applicable" after the income levels is arbitrary. It would be better to place all the missing income level categories either before or after all the known values.

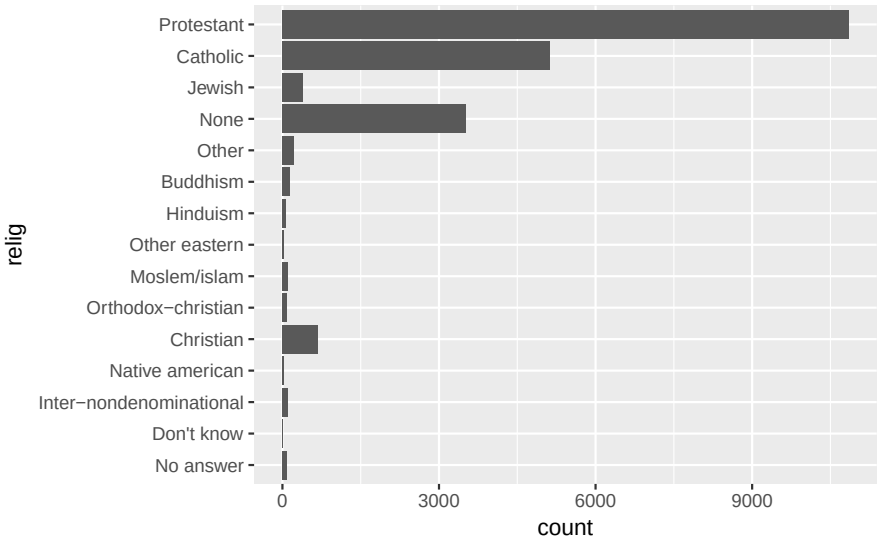
```
levels(gss_cat$rincome)
#>  [1] "No answer"      "Don't know"     "Refused"        "$25000 or more"
#>  [5] "$20000 - 24999" "$15000 - 19999" "$10000 - 14999" "$8000 to 9999"
#>  [9] "$7000 to 7999"  "$6000 to 6999"  "$5000 to 5999"  "$4000 to 4999"
#> [13] "$3000 to 3999"  "$1000 to 2999"  "Lt $1000"       "Not applicable"
```

The levels of `relig` is arbitrary: there is no natural ordering, and they don't appear to be ordered by stats

within the dataset.

```
levels(gss_cat$relig)
#>  [1] "No answer"               "Don't know"
#>  [3] "Inter-nondenominational" "Native american"
#>  [5] "Christian"               "Orthodox-christian"
#>  [7] "Moslem/islam"            "Other eastern"
#>  [9] "Hinduism"                "Buddhism"
#> [11] "Other"                   "None"
#> [13] "Jewish"                  "Catholic"
#> [15] "Protestant"              "Not applicable"
```

```
gss_cat %>%
  ggplot(aes(relig)) +
  geom_bar() +
  coord_flip()
```



The same goes for `denom`.

```
levels(gss_cat$denom)
#>  [1] "No answer"            "Don't know"           "No denomination"
#>  [4] "Other"                "Episcopal"            "Presbyterian-dk wh"
#>  [7] "Presbyterian, merged" "Other presbyterian"   "United pres ch in us"
#> [10] "Presbyterian c in us" "Lutheran-dk which"    "Evangelical luth"
#> [13] "Other lutheran"       "Wi evan luth synod"   "Lutheran-mo synod"
#> [16] "Luth ch in america"   "Am lutheran"          "Methodist-dk which"
#> [19] "Other methodist"      "United methodist"     "Afr meth ep zion"
#> [22] "Afr meth episcopal"   "Baptist-dk which"     "Other baptists"
#> [25] "Southern baptist"     "Nat bapt conv usa"    "Nat bapt conv of am"
#> [28] "Am bapt ch in usa"    "Am baptist asso"      "Not applicable"
```

Ignoring "No answer", "Don't know", and "Other party", the levels of `partyid` are ordered from "Strong Republican"" to "Strong Democrat".

```
levels(gss_cat$partyid)
#>  [1] "No answer"          "Don't know"         "Other party"
#>  [4] "Strong republican"  "Not str republican" "Ind,near rep"
#>  [7] "Independent"        "Ind,near dem"       "Not str democrat"
#> [10] "Strong democrat"
```

14.4.3 Exercise 3

Why did moving "Not applicable" to the front of the levels move it to the bottom of the plot?

Because that gives the level "Not applicable" an integer value of 1.

14.5 Modifying factor levels

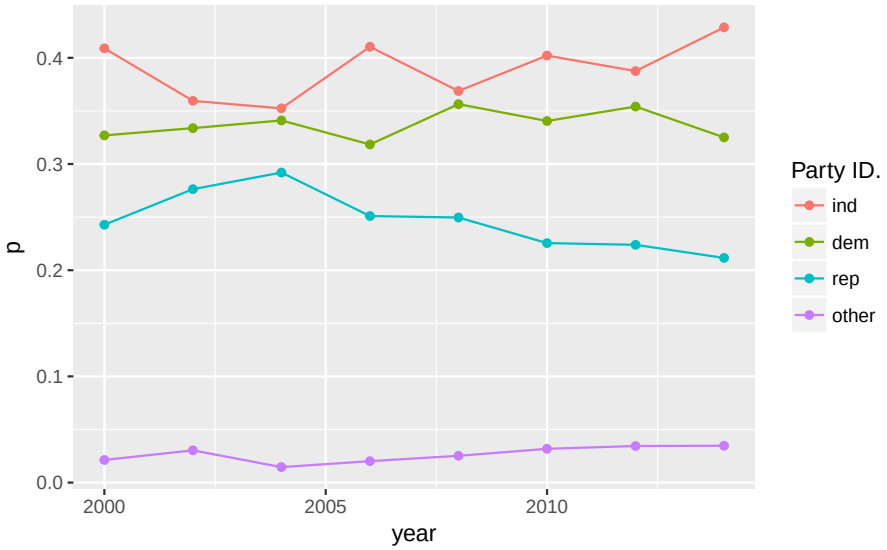
14.5.1 Exercise 1

How have the proportions of people identifying as Democrat, Republican, and Independent changed over time?

To answer that, we need to combine the multiple levels into Democrat, Republican, and Independent

```
levels(gss_cat$partyid)
#>  [1] "No answer"          "Don't know"         "Other party"
#>  [4] "Strong republican"  "Not str republican" "Ind,near rep"
#>  [7] "Independent"        "Ind,near dem"       "Not str democrat"
#> [10] "Strong democrat"
```

```
gss_cat %>%
  mutate(partyid =
           fct_collapse(partyid,
                        other = c("No answer", "Don't know", "Other party"),
                        rep = c("Strong republican", "Not str republican"),
                        ind = c("Ind,near rep", "Independent", "Ind,near dem"),
                        dem = c("Not str democrat", "Strong democrat"))) %>%
  count(year, partyid)  %>%
  group_by(year) %>%
  mutate(p = n / sum(n)) %>%
  ggplot(aes(x = year, y = p,
             colour = fct_reorder2(partyid, year, p))) +
  geom_point() +
  geom_line() +
  labs(colour = "Party ID.")
```



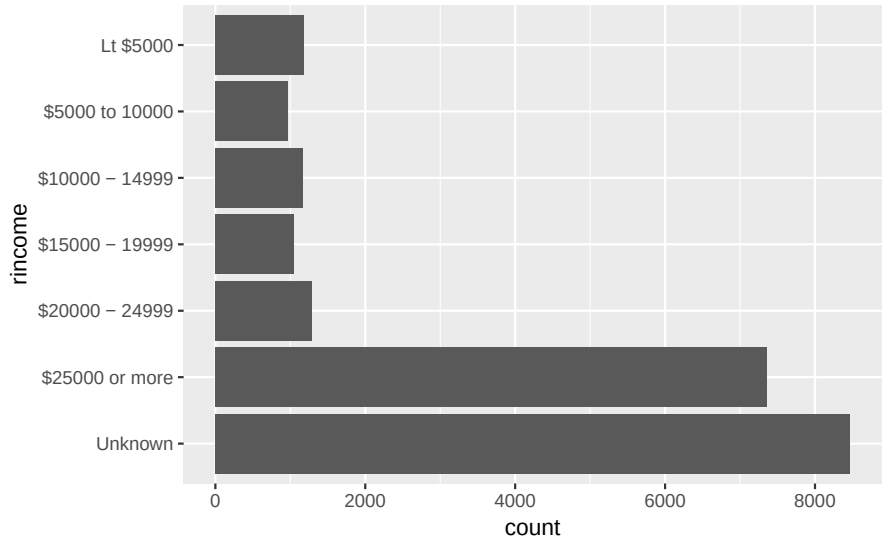
14.5.2 Exercise 2

How could you collapse `rincome` into a small set of categories?

Group all the non-responses into one category, and then group other categories into a smaller number. Since there is a clear ordering, we wouldn't want to use something like `fct_lump`.

```
levels(gss_cat$rincome)
#>  [1] "No answer"      "Don't know"     "Refused"        "$25000 or more"
#>  [5] "$20000 - 24999" "$15000 - 19999" "$10000 - 14999" "$8000 to 9999"
#>  [9] "$7000 to 7999"  "$6000 to 6999"  "$5000 to 5999"  "$4000 to 4999"
#> [13] "$3000 to 3999"  "$1000 to 2999"  "Lt $1000"       "Not applicable"
```

```
library("stringr")
gss_cat %>%
  mutate(rincome =
           fct_collapse(
             rincome,
             `Unknown` = c("No answer", "Don't know", "Refused", "Not applicable"),
             `Lt $5000` = c("Lt $1000", str_c("$", c("1000", "3000", "4000"),
                                              " to ", c("2999", "3999", "4999"))),
             `$5000 to 10000` = str_c("$", c("5000", "6000", "7000", "8000"),
                                      " to ", c("5999", "6999", "7999", "9999"))
           )) %>%
  ggplot(aes(x = rincome)) +
  geom_bar() +
  coord_flip()
```



Chapter 15

Dates and Times

15.1 Introduction

```
library(tidyverse)
library(lubridate)
library(nycflights13)
```

15.2 Creating date/times

This code is needed by exercises.

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
    sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)
  ) %>%
  select(origin, dest, ends_with("delay"), ends_with("time"))
```

15.2.1 Exercise 1

What happens if you parse a string that contains invalid dates?

```
ret <- ymd(c("2010-10-10", "bananas"))
#> Warning: 1 failed to parse.
print(class(ret))
#> [1] "Date"
ret
#> [1] "2010-10-10" NA
```

It produces an `NA` and an warning message.

15.2.2 Exercise 2

What does the `tzone` argument to `today()` do? Why is it important?

It determines the time-zone of the date. Since different time-zones can have different dates, the value of `today()` can vary depending on the time-zone specified.

15.2.3 Exercise 3

Use the appropriate **lubridate** function to parse each of the following dates:

```
d1 <- "January 1, 2010"
mdy(d1)
#> [1] "2010-01-01"
d2 <- "2015-Mar-07"
ymd(d2)
#> [1] "2015-03-07"
d3 <- "06-Jun-2017"
dmy(d3)
#> [1] "2017-06-06"
d4 <- c("August 19 (2015)", "July 1 (2015)")
mdy(d4)
#> [1] "2015-08-19" "2015-07-01"
d5 <- "12/30/14" # Dec 30, 2014
mdy(d5)
#> [1] "2014-12-30"
```

15.3 Date-Time Components

The following code from the chapter is used

```
sched_dep <- flights_dt %>%
  mutate(minute = minute(sched_dep_time)) %>%
  group_by(minute) %>%
  summarise(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n())
```

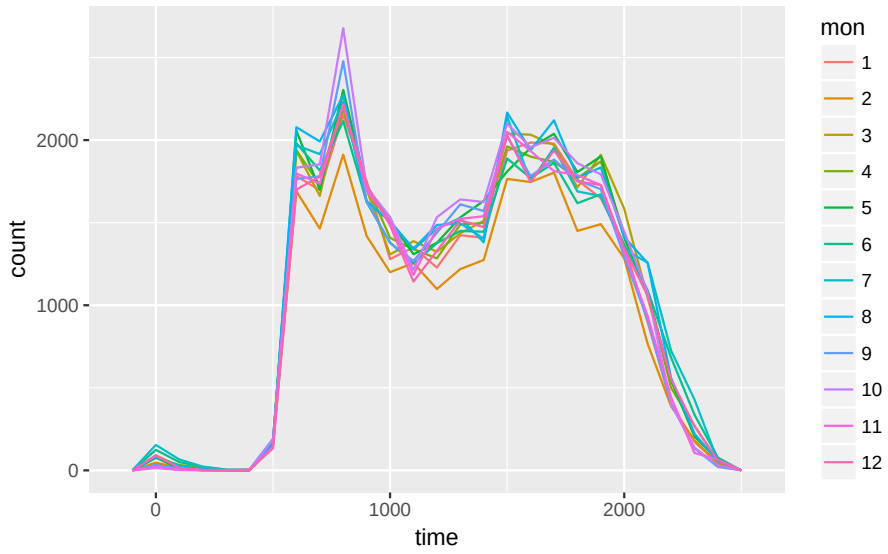
In the previous code, the difference between rounded and un-rounded dates provides the within-period time.

15.3.1 Exercise 1

How does the distribution of flight times within a day change over the course of the year?

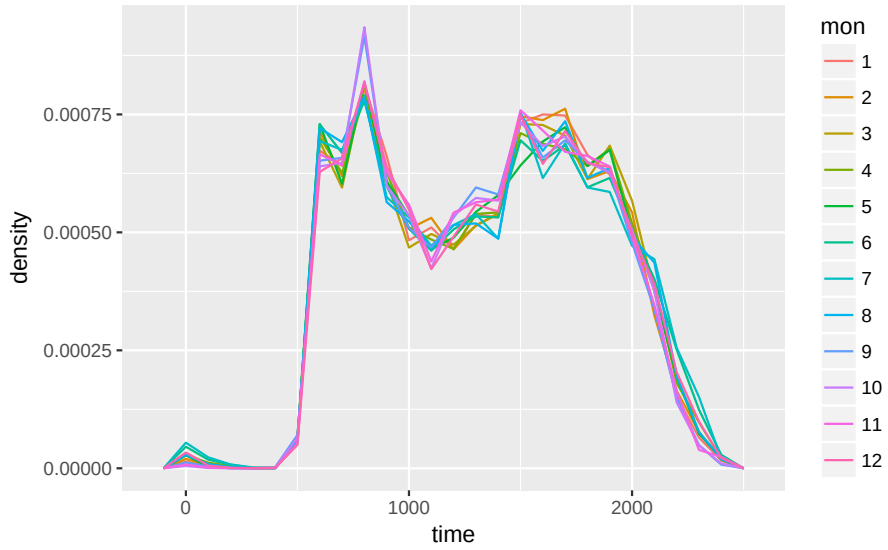
Let's try plotting this by month:

```
flights_dt %>%
  mutate(time = hour(dep_time) * 100 + minute(dep_time),
         mon = as.factor(month
                         (dep_time))) %>%
  ggplot(aes(x = time, group = mon, color = mon)) +
  geom_freqpoly(binwidth = 100)
```



This will look better if everything is normalized within groups. The reason that February is lower is that there are fewer days and thus fewer flights.

```
flights_dt %>%
  mutate(time = hour(dep_time) * 100 + minute(dep_time),
         mon = as.factor(month
                         (dep_time))) %>%
  ggplot(aes(x = time, y = ..density.., group = mon, color = mon)) +
  geom_freqpoly(binwidth = 100)
```



At least to me there doesn't appear to much difference in within-day distribution over the year, but I maybe thinking about it incorrectly.

15.3.2 Exercise 2

Compare `dep_time`, `sched_dep_time` and `dep_delay`. Are they consistent? Explain your findings.

If they are consistent, then `dep_time = sched_dep_time + dep_delay`.

```
flights_dt %>%
  mutate(dep_time_ = sched_dep_time + dep_delay * 60) %>%
  filter(dep_time_ != dep_time) %>%
  select(dep_time_, dep_time, sched_dep_time, dep_delay)
#> # A tibble: 1,205 x 4
#>   dep_time_           dep_time            sched_dep_time      dep_delay
#>   <dttm>              <dttm>              <dttm>                  <dbl>
#> 1 2013-01-02 08:48:00 2013-01-01 08:48:00 2013-01-01 18:35:00      853.
#> 2 2013-01-03 00:42:00 2013-01-02 00:42:00 2013-01-02 23:59:00       43.
#> 3 2013-01-03 01:26:00 2013-01-02 01:26:00 2013-01-02 22:50:00      156.
#> 4 2013-01-04 00:32:00 2013-01-03 00:32:00 2013-01-03 23:59:00       33.
#> 5 2013-01-04 00:50:00 2013-01-03 00:50:00 2013-01-03 21:45:00      185.
#> 6 2013-01-04 02:35:00 2013-01-03 02:35:00 2013-01-03 23:59:00      156.
#> # ... with 1,199 more rows
```

There exist discrepancies. It looks like there are mistakes in the dates. These are flights in which the actual departure time is on the *next* day relative to the scheduled departure time. We forgot to account for this when creating the date-times. The code would have had to check if the departure time is less than the scheduled departure time. Alternatively, simply adding the delay time is more robust because it will automatically account for crossing into the next day.

15.3.3 Exercise 3

Compare `air_time` with the duration between the departure and arrival. Explain your findings.

```
flights_dt %>%
  mutate(flight_duration = as.numeric(arr_time - dep_time),
         air_time_mins = air_time,
         diff = flight_duration - air_time_mins) %>%
  select(origin, dest, flight_duration, air_time_mins, diff)
#> # A tibble: 328,063 x 5
#>   origin dest  flight_duration air_time_mins  diff
#>   <chr>  <chr>           <dbl>         <dbl> <dbl>
#> 1 EWR    IAH               193.          227.  -34.
#> 2 LGA    IAH               197.          227.  -30.
#> 3 JFK    MIA               221.          160.   61.
#> 4 JFK    BQN               260.          183.   77.
#> 5 LGA    ATL               138.          116.   22.
#> 6 EWR    ORD               106.          150.  -44.
#> # ... with 3.281e+05 more rows
```

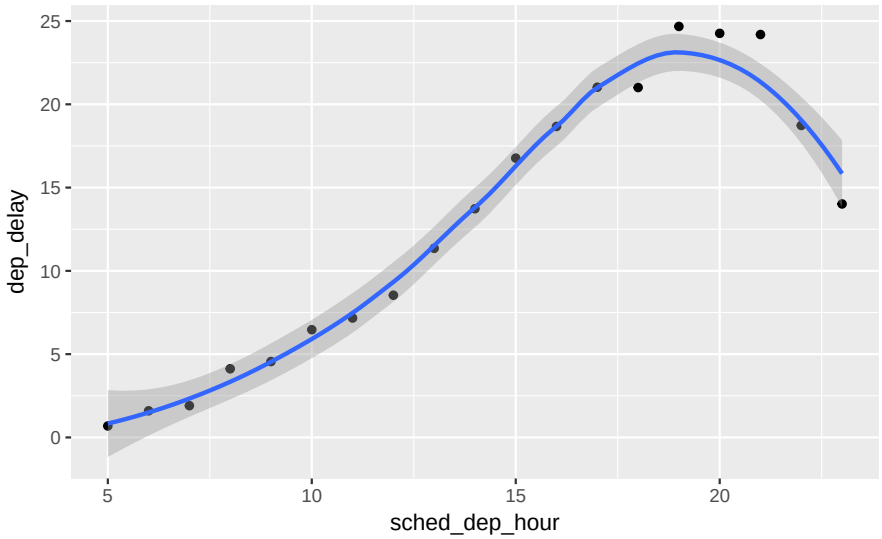
15.3.4 Exercise 4

How does the average delay time change over the course of a day? Should you use `dep_time` or `sched_dep_time`? Why?

Use `sched_dep_time` because that is the relevant metric for someone scheduling a flight. Also, using `dep_time` will always bias delays to later in the day since delays will push flights later.

```
flights_dt %>%
  mutate(sched_dep_hour = hour(sched_dep_time)) %>%
  group_by(sched_dep_hour) %>%
  summarise(dep_delay = mean(dep_delay)) %>%
  ggplot(aes(y = dep_delay, x = sched_dep_hour)) +
```

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



15.3.5 Exercise 5

On what day of the week should you leave if you want to minimize the chance of a delay?

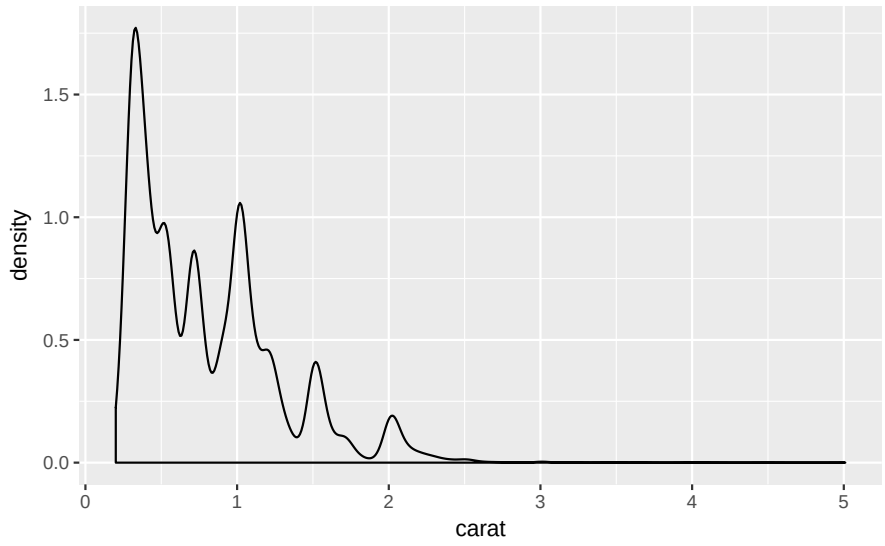
Sunday has the lowest average departure delay time and the lowest average arrival delay time.

```
flights_dt %>%
  mutate(dow = wday(sched_dep_time)) %>%
  group_by(dow) %>%
  summarise(dep_delay = mean(dep_delay),
            arr_delay = mean(arr_delay, na.rm = TRUE))
#> # A tibble: 7 x 3
#>     dow dep_delay arr_delay
#>   <dbl>     <dbl>     <dbl>
#> 1    1.      11.5      4.82
#> 2    2.      14.7      9.65
#> 3    3.      10.6      5.39
#> 4    4.      11.7      7.05
#> 5    5.      16.1     11.7
#> 6    6.      14.7      9.07
#> # ... with 1 more row
```

15.3.6 Exercise 6

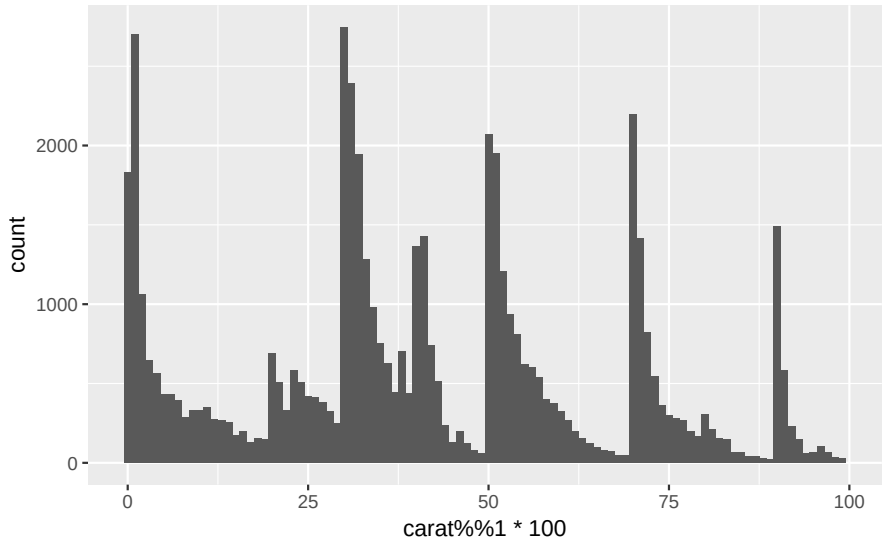
What makes the distribution of `diamonds$carat` and `flights$sched_dep_time` similar?

```
ggplot(diamonds, aes(x = carat)) +
  geom_density()
```



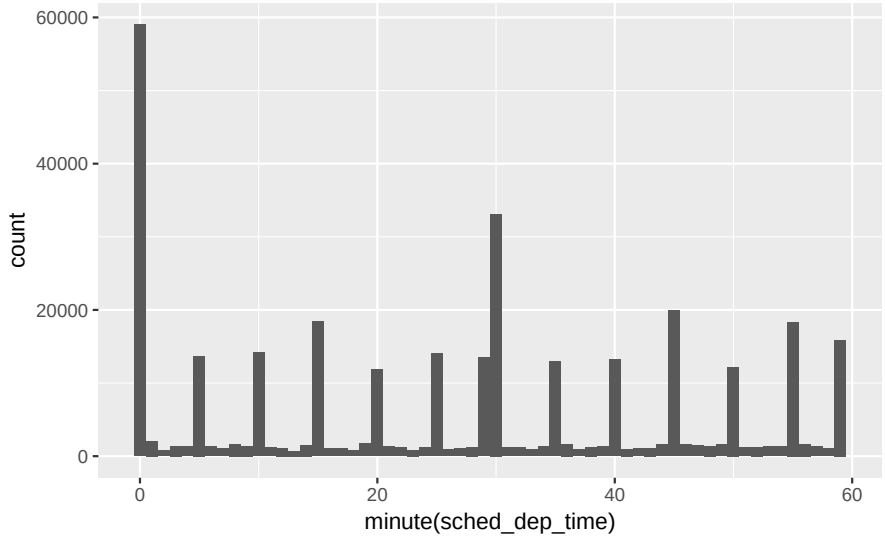
In both `carat` and `sched_dep_time` there are abnormally large numbers of values are at nice "human" numbers. In `sched_dep_time` it is at 00 and 30 minutes. In carats, it is at 0, 1/3, 1/2, 2/3,

```
ggplot(diamonds, aes(x = carat %% 1 * 100)) +
  geom_histogram(binwidth = 1)
```



In scheduled departure times it is 00 and 30 minutes, and minutes ending in 0 and 5.

```
ggplot(flights_dt, aes(x = minute(sched_dep_time))) +
  geom_histogram(binwidth = 1)
```

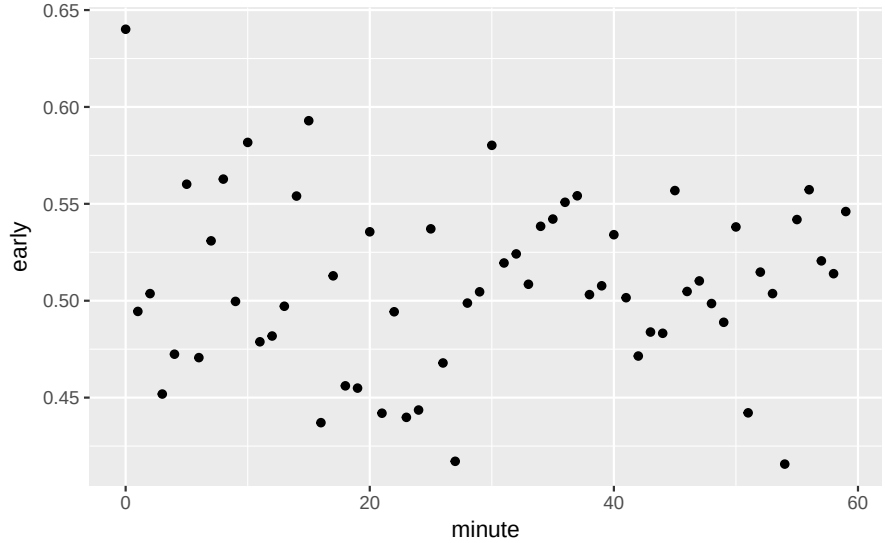


15.3.7 Exercise 7

Confirm my hypothesis that the early departures of flights in minutes 20-30 and 50-60 are caused by scheduled flights that leave early. Hint: create a binary variable that tells you whether or not a flight was delayed.

At the minute level, there doesn't appear to be anything:

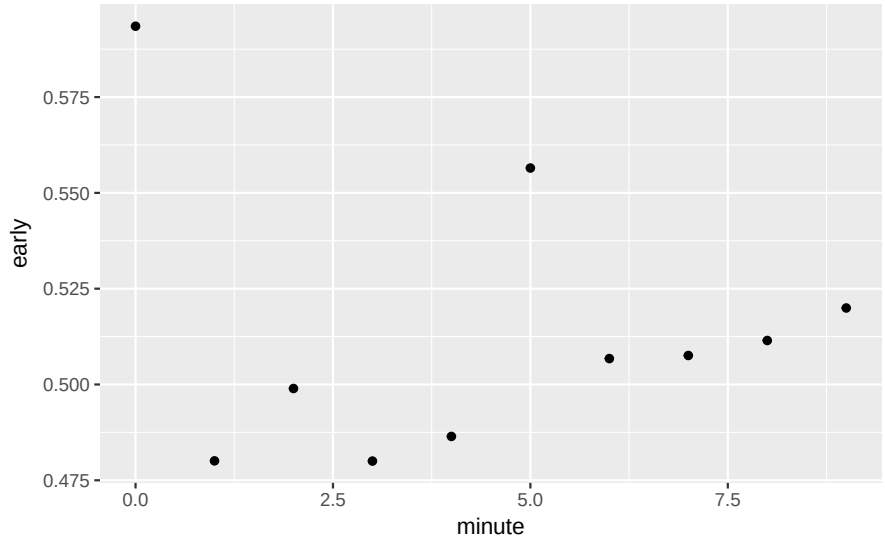
```
flights_dt %>%
  mutate(early = dep_delay < 0,
         minute = minute(sched_dep_time)) %>%
  group_by(minute) %>%
  summarise(early = mean(early)) %>%
  ggplot(aes(x = minute, y = early)) +
  geom_point()
```



But if grouped in 10 minute intervals, there is a higher proportion of early flights during those minutes.

```
flights_dt %>%
  mutate(early = dep_delay < 0,
         minute = minute(sched_dep_time) %% 10) %>%
```

```
  group_by(minute) %>%
  summarise(early = mean(early)) %>%
  ggplot(aes(x = minute, y = early)) +
  geom_point()
```



15.4 Time Spans

15.4.1 Exercise 1

Why is there `months()` but no `dmonths()`?

There is no direct unambiguous value of months in seconds since months have differing numbers of days.

- 31 days: January, March, May, July, August, October
- 30 days: April, Jun, September, November, December
- 28 or 29 days: February

Though in the past, in the pre-computer era, for arithmetic convenience, bankers adopted a 360 day year with 30 day months.

15.4.2 Exercise 2

Explain `days(overnight * 1)` to someone who has just started learning R. How does it work?

The variable `overnight` is equal to `TRUE` or `FALSE`. If it is an overnight flight, this becomes 1 day, and if not, then overnight = 0, and no days are added to the date.

15.4.3 Exercise 3

Create a vector of dates giving the first day of every month in 2015. Create a vector of dates giving the first day of every month in the current year.

A vector of the first day of the month for every month in 2015:

```
ymd("2015-01-01") + months(0:11)
#>  [1] "2015-01-01" "2015-02-01" "2015-03-01" "2015-04-01" "2015-05-01"
```

```
#>  [6] "2015-06-01" "2015-07-01" "2015-08-01" "2015-09-01" "2015-10-01"
#> [11] "2015-11-01" "2015-12-01"
```

To get the vector of the first day of the month for *this* year, we first need to figure out what this year is, and get January 1st of it. I can do that by taking `today()` and truncating it to the year using `floor_date`:

```
floor_date(today(), unit = "year") + months(0:11)
#>  [1] "2018-01-01" "2018-02-01" "2018-03-01" "2018-04-01" "2018-05-01"
#>  [6] "2018-06-01" "2018-07-01" "2018-08-01" "2018-09-01" "2018-10-01"
#> [11] "2018-11-01" "2018-12-01"
```

15.4.4 Exercise 4

Write a function that given your birthday (as a date), returns how old you are in years.

```
age <- function(bday) {
  (bday %--% today()) %/% years(1)
}
age(ymd("1990-10-12"))
#> Note: method with signature 'Timespan#Timespan' chosen for function '%/%',
#>  target signature 'Interval#Period'.
#>  "Interval#ANY", "ANY#Period" would also be valid
#> [1] 27
```

15.4.5 Exercise 5

Why can't `(today() %--% (today() + years(1)) / months(1)` work?

It appears to work. Today is a date. Today + 1 year is a valid endpoint for an interval. And months is period that is defined in this period.

```
(today() %--% (today() + years(1))) %/% months(1)
#> [1] 12
(today() %--% (today() + years(1))) / months(1)
#> [1] 12
```

15.5 Time Zones

No exercises.

Part III

Program

Chapter 16

Introduction

Chapter 17

Pipes

No exercises in this chapter.

Chapter 18

Functions

18.1 Introduction

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

18.2 When should you write a function?

18.2.1 Exercise 1

Why is TRUE not a parameter to `rescale01()`? What would happen if `x` contained a single missing value, and `na.rm` was `FALSE`?

First, note that by a a single missing value, this means that the vector `x` has at least one element equal to `NA`.

If there were any `NA` values, and `na.rm = FALSE`, then the function would return `NA`.

I can confirm this by testing a function that allows for `na.rm` as an argument

```
rescale01_alt <- function(x, finite = TRUE) {
  rng <- range(x, na.rm = finite, finite = finite)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01_alt(c(NA, 1:5), finite = FALSE)
#> [1] NA NA NA NA NA NA
rescale01_alt(c(NA, 1:5), finite = TRUE)
#> [1]   NA 0.00 0.25 0.50 0.75 1.00
```

18.2.2 Exercise 2

In the second variant of `rescale01()`, infinite values are left unchanged. Rewrite `rescale01()` so that `-Inf` is mapped to 0, and `Inf` is mapped to 1.

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  y <- (x - rng[1]) / (rng[2] - rng[1])
  y[y == -Inf] <- 0
```

```
  y[y == Inf] <- 1
  y
}

rescale01(c(Inf, -Inf, 0:5, NA))
#> [1] 1.0 0.0 0.0 0.2 0.4 0.6 0.8 1.0  NA
```

18.2.3 Exercise 3

Practice turning the following code snippets into functions. Think about what each function does. What would you call it? How many arguments does it need? Can you rewrite it to be more expressive or less duplicative?

```
mean(is.na(x))

x / sum(x, na.rm = TRUE)

sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
```

This function calculates the proportion of `NA` values in a vector

```
prop_na <- function(x) {
  mean(is.na(x))
}
prop_na(c(NA, 0, NA, 0, NA))
#> [1] 0.6
```

This function standardizes a function to its weight. If all elements of `x` are non-negative, this will ensure the vector sums to 1.

```
weights <- function(x) {
  x / sum(x, na.rm = TRUE)
}
y <- weights(0:5)
y
#> [1] 0.0000 0.0667 0.1333 0.2000 0.2667 0.3333
sum(y)
#> [1] 1
```

This function calculates the coefficient of variation (assuming that `x` can only take non-negative values). The coefficient of variation is the standard deviation divided by the mean

```
coef_variation <- function(x) {
  sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
}
coef_variation(runif(10))
#> [1] 0.672
```

18.2.4 Exercise 4

Follow http://nicercode.github.io/intro/writing-functions.html to write your own functions to compute the variance and skew of a numeric vector.

Note The math in https://nicercode.github.io/intro/writing-functions.html seems not to be rendering, but I'll write functions for the variance and skewness.

The sample variance is defined as

$$Var(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where the sample mean is $\bar{x} = (\sum x_i)/n$.

```
variance <- function(x) {
  # remove missing values
  x <- x[!is.na(x)]
  n <- length(x)
  m <- mean(x)
  sq_err <- (x - m) ^ 2
  sum(sq_err) / (n - 1)
}
var(1:10)
#> [1] 9.17
variance(1:10)
#> [1] 9.17
```

There are multiple definitions of skewness, but I'll use the method of moments estimator of the population skewness,

$$b_1 = \frac{m_3}{s^3} = \frac{\frac{1}{n}\sum(x_i - \bar{x})^3}{\left(\frac{1}{n-1}\sum(x_i - \bar{x})^2\right)^{\frac{3}{2}}}$$

```
skewness <- function(x) {
  x <- x[!is.na(x)]
  n <- length(x)
  m <- mean(x)
  m3 <- sum((x - m) ^ 3) / n
  s3 <- sqrt(sum((x - m) ^ 2) / (n - 1))
  m3 / s3
}
skewness(rgamma(10, 1, 1))
#> [1] 1.56
```

5. Write `both_na()`, a function that takes two vectors of the same length and returns the number of positions that have an `NA` in both vectors.

```
both_na <- function(x, y) {
  sum(is.na(x) & is.na(y))
}
both_na(c(NA, NA,  1, 2),
        c(NA,  1, NA, 2))
#> [1] 1
both_na(c(NA, NA,  1, 2, NA, NA, 1),
        c(NA,  1, NA, 2, NA, NA, 1))
#> [1] 3
```

18.2.5 Exercise 6

What do the following functions do? Why are they useful even though they are so short?

```
is_directory <- function(x) file.info(x)$isdir
is_readable <- function(x) file.access(x, 4) == 0
```

The function `is_directory` checks whether the path in `x` is a directory. The function `is_readable` checks whether the path in `x` is readable, meaning that the file exists and the user has permission to open it. These functions are useful even though they are short because their names make it much clearer what the code is doing.

18.2.6 Exercise 7

Read the complete lyrics to "Little Bunny Foo Foo". There's a lot of duplication in this song. Extend the initial piping example to recreate the complete song, and use functions to reduce the duplication.

The lyrics of one of the most common versions of this song are

Little bunny Foo Foo Hopping through the forest Scooping up the field mice And bopping them on the head

Down came the Good Fairy, and she said "Little bunny Foo Foo I don't want to see you Scooping up the field mice

And bopping them on the head. I'll give you three chances, And if you don't stop, I'll turn you into a GOON!" And the next day…

The verses repeat with one chance fewer each time. When there are no chances left, the Good Fairy says

> "I gave you three chances, and you didn't stop; so…." POOF. She turned him into a GOON! And the moral of this story is: *hare today, goon tomorrow.*

Here's one way of writing this

```
threat <- function(chances) {
  give_chances(from = Good_Fairy,
               to = foo_foo,
               number = chances,
               condition = "Don't behave",
               consequence = turn_into_goon)
}

lyric <- function() {
  foo_foo %>%
    hop(through = forest) %>%
    scoop(up = field_mouse) %>%
    bop(on = head)

  down_came(Good_Fairy)
  said(Good_Fairy,
       c("Little bunny Foo Foo",
         "I don't want to see you",
         "Scooping up the field mice",
         "And bopping them on the head.")
}

lyric()
threat(3)
lyric()
threat(2)
lyric()
threat(1)
lyric()
turn_into_goon(Good_Fairy, foo_foo)
```

18.3 Functions are for humans and computers

18.3.1 Exercise 1

Read the source code for each of the following three functions, puzzle out what they do, and then brainstorm better names.

```
f1 <- function(string, prefix) {
  substr(string, 1, nchar(prefix)) == prefix
}

f2 <- function(x) {
  if (length(x) <= 1) return(NULL)
  x[-length(x)]
}

f3 <- function(x, y) {
  rep(y, length.out = length(x))
}
```

The function `f1` returns whether a function has a common prefix.

```
f1(c("str_c", "str_foo", "abc"), "str_")
#> [1]  TRUE  TRUE FALSE
```

A better name for `f1` is `has_prefix()`

The function `f2` drops the last element

```
f2(1:3)
#> [1] 1 2
f2(1:2)
#> [1] 1
f2(1)
#> NULL
```

A better name for `f2` is `drop_last()`.

The function `f3` repeats `y` once for each element of `x`.

```
f3(1:3, 4)
#> [1] 4 4 4
```

This is a harder one to name. I would say something like `recycle` (R's name for this behavior), or `epxand`.

18.3.2 Exercise 2

Take a function that you've written recently and spend 5 minutes brainstorming a better name for it and its arguments.

Answer left to the reader.

18.3.3 Exercise 3

Compare and contrast `rnorm()` and `MASS::mvrnorm()`. How could you make them more consistent?

You can ignore

`rnorm` samples from the univariate normal distribution, while `MASS::mvrnorm` samples from the multivariate normal distribution. The main arguments in `rnorm` are `n`, `mean`, `sd`. The main arguments is `MASS::mvrnorm` are `n`, `mu`, `Sigma`. To be consistent they should have the same names. However, this is difficult. In general, it is better to be consistent with more widely used functions, e.g. `rmvnorm` should follow the conventions of `rnorm`. However, while `mean` is correct in the multivariate case, `sd` does not make sense in the multivariate case. Both functions an internally consistent though; it would be bad to have `mu` and `sd` or `mean` and `Sigma`.

18.3.4 Exercise 4

Make a case for why `norm_r()`, `norm_d()` etc would be better than `rnorm()`, `dnorm()`. Make a case for the opposite.

If named `norm_r` and `norm_d`, it groups the family of functions related to the normal distribution. If named `rnorm`, and `dnorm`, functions related to are grouped into families by the action they perform. `r*` functions always sample from distributions: `rnorm`, `rbinom`, `runif`, `rexp`. `d*` functions calculate the probability density or mass of a distribution: `dnorm`, `dbinom`, `dunif`, `dexp`.

18.4 Conditional execution

18.4.1 Exercise 1

What's the difference between `if` and `ifelse()`? > Carefully read the help and construct three examples that illustrate the key differences.

The keyword `if` tests a single condition, while `ifelse` tests each element.

18.4.2 Exercise 2

Write a greeting function that says "good morning", "good afternoon", or "good evening", depending on the time of day. (Hint: use a time argument that defaults to `lubridate::now()`. That will make it easier to test your function.)

```
greet <- function(time = lubridate::now()) {
  hr <- hour(time)
  # I don't know what to do about times after midnight,
  # are they evening or morning?
  if (hr < 12) {
    print("good morning")
  } else if (hr < 17) {
    print("good afternoon")
  } else {
    print("good evening")
  }
}
greet()
#> [1] "good evening"
greet(ymd_h("2017-01-08:05"))
#> [1] "good morning"
greet(ymd_h("2017-01-08:13"))
#> [1] "good afternoon"
greet(ymd_h("2017-01-08:20"))
#> [1] "good evening"
```

18.4.3 Exercise 3

Implement a `fizzbuzz` function. It takes a single number as input. If the number is divisible by three, it returns "fizz". If it's divisible by five it returns "buzz". If it's divisible by three and five, it returns "fizzbuzz". Otherwise, it returns the number. Make sure you first write working code before you create the function.

```
fizzbuzz <- function(x) {
  stopifnot(length(x) == 1)
  stopifnot(is.numeric(x))
  # this could be made more efficient by minimizing the
  # number of tests
  if (!(x %% 3) & !(x %% 5)) {
    print("fizzbuzz")
  } else if (!(x %% 3)) {
    print("fizz")
  } else if (!(x %% 5)) {
    print("buzz")
  }
}
fizzbuzz(6)
#> [1] "fizz"
fizzbuzz(10)
#> [1] "buzz"
fizzbuzz(15)
#> [1] "fizzbuzz"
fizzbuzz(2)
```

18.4.4 Exercise 4

How could you use `cut()` to simplify this set of nested if-else statements?

```
if (temp <= 0) {
  "freezing"
} else if (temp <= 10) {
  "cold"
} else if (temp <= 20) {
  "cool"
} else if (temp <= 30) {
  "warm"
} else {
  "hot"
}
```

How would you change the call to `cut()` if I'd used `<` instead of `<=`? What is the other chief advantage of cut() for this problem? (Hint: what happens if you have many values in temp?)

```
temp <- seq(-10, 50, by = 5)
cut(temp, c(-Inf, 0, 10, 20, 30, Inf), right = TRUE,
    labels = c("freezing", "cold", "cool", "warm", "hot"))
#>  [1] freezing freezing freezing cold     cold     cool     cool
#>  [8] warm     warm     hot      hot      hot      hot
#> Levels: freezing cold cool warm hot
```

To have intervals open on the left (using `<`), I change the argument to `right = FALSE`

```
temp <- seq(-10, 50, by = 5)
cut(temp, c(-Inf, 0, 10, 20, 30, Inf), right = FALSE,
    labels = c("freezing", "cold", "cool", "warm", "hot"))
#>  [1] freezing freezing cold     cold     cool     cool     warm
#>  [8] warm     hot      hot      hot      hot      hot
#> Levels: freezing cold cool warm hot
```

Two advantages of using `cut` is that it works on vectors, whereas `if` only works on a single value (I already demonstrated this above), and that to change comparisons I only needed to change the argument to `right`, but I would have had to change four operators in the `if` expression.

18.4.5 Exercise 5

What happens if you use `switch()` with numeric values?

It selects that number argument from ....

```
switch(2, "one", "two", "three")
#> [1] "two"
```

18.4.6 Exercise 6

What does this `switch()` call do? What happens if `x` is `"e"`?

It will return the `"ab"` for `a` or `b`, `"cd"` for `c` or `d`, an `NULL` for `e`. It returns the first non-missing value for the first name it matches.

```
x <- "e"
switch(x,
  a = ,
  b = "ab",
  c = ,
  d = "cd"
)
```

Experiment, then carefully read the documentation.

```
switcheroo <- function(x) {
  switch(x,
  a = ,
  b = "ab",
  c = ,
  d = "cd"
  )
}
switcheroo("a")
#> [1] "ab"
switcheroo("b")
#> [1] "ab"
switcheroo("c")
#> [1] "cd"
switcheroo("d")
#> [1] "cd"
switcheroo("e")
```

18.5 Function arguments

18.5.1 Exercise 1

What does `commas(letters, collapse = "-")` do? Why?

The `commas` function in the chapter is defined as

```
commas <- function(...) {
  stringr::str_c(..., collapse = ", ")
}
```

When `commas()` is given a collapse argument, it throws an error.

```
commas(letters, collapse = "-")
#> Error in stringr::str_c(..., collapse = ", "): formal argument "collapse" matched by multiple actual
```

This is because when the argument `collapse` is given to `commas`, it is passed to `str_c` as part of `...`. In other words, the previous code is equivalent to

```
str_c(letters, collapse = "-", collapse = ", ")
```

However, it is an error to give the same named argument to a function twice.

One way to allow the user to override the separator in `commas` is to add a `collapse` argument to the function.

```
commas <- function(..., collapse = ", ") {
  stringr::str_c(..., collapse = collapse)
}
```

18.5.2 Exercise 2

It'd be nice if you could supply multiple characters to the pad argument, e.g. `rule("Title", pad = "-+")`. Why doesn't this currently work? How could you fix it?

```
rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
```

```
rule("Important output")
#> Important output ------------------------------------------------------
rule("Important output", pad = "-+")
#> Important output -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

It does not work because it duplicates pad by the width minus the length of the string. This is implicitly assuming that pad is only one character. I could adjust the code to calculate the length of pad. The trickiest part is handling what to do if width is not a multiple of the number of characters of `pad`.

```
rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  padchar <- nchar(pad)
  cat(title, " ",
      stringr::str_dup(pad, width %/% padchar),
      # if not multiple, fill in the remaining characters
      stringr::str_sub(pad, 1, width %% padchar),
      "\n", sep = "")
```

```
}
rule("Important output")
#> Important output ----------------------------------------------------
rule("Important output", pad = "-+")
#> Important output -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
rule("Important output", pad = "-+-")
#> Important output -+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+-
```

18.5.3 Exercise 3

What does the `trim` argument to `mean()` do? When might you use it?

The `trim` arguments trims a fraction of observations from each end of the vector (meaning the range) before calculating the mean. This is useful for calculating a measure of central tendency that is robust to outliers.

18.5.4 Exercise 4

The default value for the `method` argument to `cor()` is `c("pearson", "kendall", "spearman")`. What does that mean? What value is used by default?

It means that the `method` argument can take one of those three values. The first value, `"pearson"`, is used by default.

18.6 Environment

No Exercises

Chapter 19

Vectors

19.1 Introduction

```
library("tidyverse")
#> -- Attaching packages ------------------------------------------------- tidyverse 1.2.1 --
#> ✓ ggplot2 2.2.1     ✓ purrr   0.2.4
#> ✓ tibble  1.4.2     ✓ dplyr   0.7.4
#> ✓ tidyr   0.8.0     ✓ stringr 1.3.0
#> ✓ readr   1.1.1     ✓ forcats 0.3.0
#> -- Conflicts ---------------------------------------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

19.2 Vector Basics

No exercises

19.3 Important Types of Atomic Vector

19.3.1 Exercise 1

Describe the difference between `is.finite(x)` and `!is.infinite(x)`.

To find out, try the functions on a numeric vector that includes a number and the five special values (`NA`, `NaN`, `Inf`, `-Inf`).

```
x <- c(0, NA, NaN, Inf, -Inf)
is.finite(x)
#> [1]  TRUE FALSE FALSE FALSE FALSE
!is.infinite(x)
#> [1]  TRUE  TRUE  TRUE FALSE FALSE
```

`is.finite` considers only a number to be finite, and considers missing (`NA`), not a number (`NaN`), and positive and negative infinity to be not finite. However, since `is.infinite` only considers `Inf` and `-Inf` to be infinite, `!is.infinite` considers `0` as well as missing and not-a-number to be not infinite.

So `NA` and `NaN` are neither finite or infinite. Mind blown.

19.3.2 Exercise 2

Read the source code for `dplyr::near()` (Hint: to see the source code, drop the ()). How does it work?

The source for `dplyr::near` is:

```
dplyr::near
#> function (x, y, tol = .Machine$double.eps^0.5)
#> {
#>     abs(x - y) < tol
#> }
#> <environment: namespace:dplyr>
```

Instead of checking for exact equality, it checks that two numbers are within a certain tolerance, `tol`. By default the tolerance is set to the square root of `.Machine$double.eps`, which is the smallest floating point number that the computer can represent.

19.3.3 Exercise 3

A logical vector can take 3 possible values. How many possible values can an integer vector take? How many possible values can a double take? Use Google to do some research.

The help for `.Machine` describes some of this:

As all current implementations of R use 32-bit integers and uses IEC 60559 floating-point (double precision) arithmetic,

The IEC 60559 or IEEE 754 format uses a 64 bit vector, but

19.3.4 Exercise 4

Brainstorm at least four functions that allow you to convert a double to an integer. How do they differ? Be precise.

Broadly, could convert a double to an integer by truncating or rounding to the nearest integer. For truncating or for handling ties (doubles ending in 0.5), there are multiple methods for determining which integer value to go to.

methods	0.5	-0.5	1.5	-1.5
towards zero:	0	0	1	1
away from zero	1	-1	2	-2
largest towards $+\infty$)	1	0	2	-1
smallest (towards $-\infty$)	0	-1	1	-2
even	0	0	2	-2
odd	1	-1	1	-1

See the Wikipedia article IEEE floating point for rounding rules.

For rounding, R and many programming languages use the IEEE standard. This is "round to nearest, ties to even". This is not the same as what you See the value of looking at the value of `.Machine$double.rounding` and its documentation.

```
x <- seq(-10, 10, by = 0.5)

round2 <- function(x, to_even = TRUE) {
  q <- x %/% 1
```

```
  r <- x %% 1
  q + (r >= 0.5)
}
x <- c(-12.5, -11.5, 11.5, 12.5)
round(x)
#> [1] -12 -12  12  12
round2(x, to_even = FALSE)
#> [1] -12 -11  12  13
```

The problem with the always rounding 0.5 up rule is that it is biased upwards. Rounding to nearest with ties towards even is not. Consider the sequence $-100.5, -99.5, \ldots, 0, \ldots, 99.5, 100.5$. Its sum is 0. It would be nice if rounding preserved that sum. Using the "ties towards even", the sum is still zero. However, the "ties towards $+\infty$" produces a non-zero number.

```
x <- seq(-100.5, 100.5, by = 1)
sum(x)
#> [1] 0
sum(round(x))
#> [1] 0
sum(round2(x))
#> [1] 101
```

Here's a real-world non-engineering example of rounding going terribly wrong. In 1983, the Vancouver stock exchange adjusted its index from 524.811 to 1098.892 to correct for accumulated error due to rounding to three decimal points (see Vancouver Stock Exchange).

Here's a list of a few more.

19.3.5 Exercise 5

What functions from the **readr** package allow you to turn a string into logical, integer, and double vector?

The functions `parse_logical`, `parse_integer`, and `parse_number`.

```
parse_logical(c("TRUE", "FALSE", "1", "0", "true", "t", "NA"))
#> [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE    NA
```

```
parse_integer(c("1235", "0134", "NA"))
#> [1] 1235  134   NA
```

```
parse_number(c("1.0", "3.5", "1,000", "NA"))
#> [1]    1.0    3.5 1000.0     NA
```

Read the documentation of `read_number`. In order to ignore things like currency symbols and comma separators in number strings it ignores them using a heuristic.

19.4 Using atomic vectors

19.4.1 Exercise 1

What does `mean(is.na(x))` tell you about a vector `x`? What about `sum(!is.finite(x))`?

The expression `mean(is.na(x))` calculates the proportion of missing values in a vector

```
x <- c(1:10, NA, NaN, Inf, -Inf)
mean(is.na(x))
#> [1] 0.143
```

The expression `mean(!is.finite(x))` calculates the proportion of values that are `NA`, `NaN`, or infinite.

```
mean(!is.finite(x))
#> [1] 0.286
```

19.4.2 Exercise 2

Carefully read the documentation of `is.vector()`. What does it actually test for? Why does `is.atomic()` not agree with the definition of atomic vectors above?

The function `is.vector` only checks whether the object has no attributes other than names. Thus a `list` is a vector:

```
is.vector(list(a = 1, b = 2))
#> [1] TRUE
```

But any object that has an attribute (other than names) is not:

```
x <- 1:10
attr(x, "something") <- TRUE
is.vector(x)
#> [1] FALSE
```

The idea behind this is that object oriented classes will include attributes, including, but not limited to `"class"`.

The function `is.atomic` explicitly checks whether an object is one of the atomic types ("logical", "integer", "numeric", "complex", "character", and "raw") or NULL.

```
is.atomic(1:10)
#> [1] TRUE
is.atomic(list(a = 1))
#> [1] FALSE
```

The function `is.atomic` will consider objects to be atomic even if they have extra attributes.

```
is.atomic(x)
#> [1] TRUE
```

19.4.3 Exercise 3

Compare and contrast `setNames()` with `purrr::set_names()`.

These are simple functions, so we can simply print out their source code:

```
setNames
#> function (object = nm, nm)
#> {
#>     names(object) <- nm
#>     object
#> }
#> <bytecode: 0x7fe7646b9478>
#> <environment: namespace:stats>
```

```
purrr::set_names
#> function (x, nm = x, ...)
#> {
#>     set_names_impl(x, x, nm, ...)
#> }
#> <bytecode: 0x7fe7659540a8>
#> <environment: namespace:rlang>
```

From the code we can see that `set_names` adds a few sanity checks: `x` has to be a vector, and the lengths of the object and the names have to be the same.

19.4.4 Exercise 4

Create functions that take a vector as input and returns:

1. The last value. Should you use [or [[? 2 The elements at even numbered positions.
2. Every element except the last value.
3. Only even numbers (and no missing values).

```
last_value <- function(x) {
  # check for case with no length
  if (length(x)) {
    # Use [[ as suggested because it returns one element
    x[[length(x)]]
  } else {
    x
  }
}
last_value(numeric())
#> numeric(0)
last_value(1)
#> [1] 1
last_value(1:10)
#> [1] 10
```

```
even_indices <- function(x) {
  if (length(x)) {
    x[seq_along(x) %% 2 == 0]
  } else {
    x
  }
}
even_indices(numeric())
#> numeric(0)
even_indices(1)
#> numeric(0)
even_indices(1:10)
#> [1]  2  4  6  8 10
# test using case to ensure that values not indices
# are being returned
even_indices(letters)
#>  [1] "b" "d" "f" "h" "j" "l" "n" "p" "r" "t" "v" "x" "z"
```

```
not_last <- function(x) {
  if (length(x)) {
```

```
    x[-length(x)]
  } else {
    x
  }
}
not_last(1:5)
#> [1] 1 2 3 4
```

```
even_numbers <- function(x) {
  x[!is.na(x) & (x %% 2 == 0)]
}
even_numbers(-10:10)
#>  [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

19.4.5 Exercise 5

Why is `x[-which(x > 0)]` not the same as `x[x <= 0]`?

They will treat missing values differently.

```
x <- c(-5:5, Inf, -Inf, NaN, NA)
x[-which(x > 0)]
#> [1]   -5   -4   -3   -2   -1    0 -Inf  NaN   NA
-which(x > 0)
#> [1]  -7  -8  -9 -10 -11 -12
x[x <= 0]
#> [1]   -5   -4   -3   -2   -1    0 -Inf   NA   NA
x <= 0
#>  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
#> [12] FALSE  TRUE    NA    NA
```

`-which(x > 0)` which calculates the indexes for any value that is `TRUE` and ignores `NA`. Thus is keeps `NA` and `NaN` because the comparison is not `TRUE`. `x <= 0` works slightly differently. If `x <= 0` returns `TRUE` or `FALSE` it works the same way. However, if the comparison generates a `NA`, then it will always keep that entry, but set it to `NA`. This is why the last two values of `x[x <= 0]` are `NA` rather than `c(NaN, NA)`.

19.4.6 Exercise 6

What happens when you subset with a positive integer that's bigger than the length of the vector? What happens when you subset with a name that doesn't exist?

When you subset with positive integers that are larger than the length of the vector, `NA` values are returned for those integers larger than the length of the vector.

```
(1:10)[11:12]
#> [1] NA NA
```

When a vector is subset with a name that doesn't exist, an error is generated.

```
c(a = 1, 2)[["b"]]
#> Error in c(a = 1, 2)[["b"]]: subscript out of bounds
```

19.5 Recursive Vectors (lists)

19.5.1 Exercise 1

Draw the following lists as nested sets:

1. `list(a, b, list(c, d), list(e, f))`
2. `list(list(list(list(list(list(a)))))))`

TODO

19.5.2 Exercise 2

What happens if you subset a `tibble` as if you're subsetting a list? What are the key differences between a list and a `tibble`?

Subsetting a `tibble` works the same way as a list; a data frame can be thought of as a list of columns. The key different between a list and a `tibble` is that a tibble (data frame) has the restriction that all its elements (columns) must have the same length.

```
x <- tibble(a = 1:2, b = 3:4)
x[["a"]]
#> [1] 1 2
x["a"]
#> # A tibble: 2 x 1
#>       a
#>   <int>
#> 1     1
#> 2     2
x[1]
#> # A tibble: 2 x 1
#>       a
#>   <int>
#> 1     1
#> 2     2
x[1, ]
#> # A tibble: 1 x 2
#>       a     b
#>   <int> <int>
#> 1     1     3
```

19.6 Attributes

No exercises

19.7 Augmented Vectors

19.7.1 Exercise 1

What does `hms::hms(3600)` return? How does it print? What primitive type is the augmented vector built on top of? What attributes does it use?

```
x <- hms::hms(3600)
class(x)
#> [1] "hms"      "difftime"
x
#> 01:00:00
```

`hms::hms` returns an object of class, and prints the time in "%H:%M:%S" format.

The primitive type is a double

```
typeof(x)
#> [1] "double"
```

The attributes is uses are `"units"` and `"class"`.

```
attributes(x)
#> $units
#> [1] "secs"
#>
#> $class
#> [1] "hms"      "difftime"
```

19.7.2 Exercise 2

Try and make a tibble that has columns with different lengths. What happens?

If I try to create at tibble with a scalar and column of a different length there are no issues, and the scalar is repeated to the length of the longer vector.

```
tibble(x = 1, y = 1:5)
#> # A tibble: 5 x 2
#>       x     y
#>   <dbl> <int>
#> 1    1.     1
#> 2    1.     2
#> 3    1.     3
#> 4    1.     4
#> 5    1.     5
```

However, if I try to create a tibble with two vectors of different lengths (other than one), the `tibble` function throws an error.

```
tibble(x = 1:3, y = 1:4)
#> Error: Column `x` must be length 1 or 4, not 3
```

19.7.3 Exercise 3

Based on the definition above, is it OK to have a list as a column of a tibble?

If I didn't already know the answer, what I would do is try it out. From the above, the error message was about vectors having different lengths. But there is nothing that prevents a tibble from having vectors of different types: doubles, character, integers, logical, factor, date. The later are still atomic, but they have additional attributes. So, maybe there won't be an issue with a list vector as long as it is the same length.

```
tibble(x = 1:3, y = list("a", 1, list(1:3)))
#> # A tibble: 3 x 2
#>       x y
```

```
#>   <int> <list>
#> 1     1 <chr [1]>
#> 2     2 <dbl [1]>
#> 3     3 <list [1]>
```

It works! I even used a list with heterogeneous types and there wasn't an issue. In following chapters we'll see that list vectors can be very useful: for example, when processing many different models.

Chapter 20

Iteration

20.1 Introduction

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

The package **microbenchmark** is used for timing code

```
library("microbenchmark")
```

20.2 For Loops

20.2.1 Exercise 1

Write for loops to:

1. Compute the mean of every column in `mtcars`.
2. Determine the type of each column in `nycflights13::flights`.
3. Compute the number of unique values in each column of `iris`.
4. Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.

   Think about the output, sequence, and body before you start writing the loop.

To compute the mean of every column in `mtcars`.

```
output <- vector("double", ncol(mtcars))
names(output) <- names(mtcars)
for (i in names(mtcars)) {
  output[i] <- mean(mtcars[[i]])
}
output
#>     mpg     cyl    disp      hp    drat      wt    qsec      vs      am
#>  20.091   6.188 230.722 146.688   3.597   3.217  17.849   0.438   0.406
#>    gear    carb
#>   3.688   2.812
```

Determine the type of each column in `nycflights13::flights`. Note that we need to use a `list`, not a character vector, since the class can have multiple values.

```r
data("flights", package = "nycflights13")
output <- vector("list", ncol(flights))
names(output) <- names(flights)
for (i in names(flights)) {
  output[[i]] <- class(flights[[i]])
}
output
#> $year
#> [1] "integer"
#>
#> $month
#> [1] "integer"
#>
#> $day
#> [1] "integer"
#>
#> $dep_time
#> [1] "integer"
#>
#> $sched_dep_time
#> [1] "integer"
#>
#> $dep_delay
#> [1] "numeric"
#>
#> $arr_time
#> [1] "integer"
#>
#> $sched_arr_time
#> [1] "integer"
#>
#> $arr_delay
#> [1] "numeric"
#>
#> $carrier
#> [1] "character"
#>
#> $flight
#> [1] "integer"
#>
#> $tailnum
#> [1] "character"
#>
#> $origin
#> [1] "character"
#>
#> $dest
#> [1] "character"
#>
#> $air_time
#> [1] "numeric"
#>
#> $distance
```

```
#> [1] "numeric"
#>
#> $hour
#> [1] "numeric"
#>
#> $minute
#> [1] "numeric"
#>
#> $time_hour
#> [1] "POSIXct" "POSIXt"
```

```
data(iris)
iris_uniq <- vector("double", ncol(iris))
names(iris_uniq) <- names(iris)
for (i in names(iris)) {
  iris_uniq[i] <- length(unique(iris[[i]]))
}
iris_uniq
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
#>           35           23           43           22            3
```

```
# number to draw
n <- 10
# values of the mean
mu <- c(-10, 0, 10, 100)
normals <- vector("list", length(mu))
for (i in seq_along(normals)) {
  normals[[i]] <- rnorm(n, mean = mu[i])
}
normals
#> [[1]]
#>  [1] -11.40  -9.74 -12.44 -10.01  -9.38  -8.85 -11.82 -10.25 -10.24 -10.28
#>
#> [[2]]
#>  [1] -0.5537  0.6290  2.0650 -1.6310  0.5124 -1.8630 -0.5220 -0.0526
#>  [9]  0.5430 -0.9141
#>
#> [[3]]
#>  [1] 10.47 10.36  8.70 10.74 11.89  9.90  9.06  9.98  9.17  8.49
#>
#> [[4]]
#>  [1] 100.9 100.2 100.2 101.6 100.1  99.9  98.1  99.7  99.7 101.1
```

However, we don't need a `for` loop for this since `rnorm` recycles means.

```
matrix(rnorm(n * length(mu), mean = mu), ncol = n)
#>        [,1]   [,2]  [,3]     [,4]   [,5]   [,6]    [,7]   [,8]    [,9]
#> [1,] -9.930  -9.56 -9.88 -10.2061 -12.27 -8.926 -11.178  -9.51  -8.663
#> [2,] -0.639   2.76 -1.91   0.0192   2.68 -0.665  -0.976  -1.70   0.237
#> [3,]  9.950  10.05 10.86  10.0296   9.64 11.114  11.065   8.53  11.318
#> [4,] 99.749 100.58 99.76 100.5498 100.21 99.754 100.132 100.28 100.524
#>       [,10]
#> [1,] -9.39
#> [2,] -0.11
```

```
#> [3,] 10.17
#> [4,] 99.91
```

20.2.2 Exercise 2

Eliminate the for loop in each of the following examples by taking advantage of an existing function that works with vectors:

```
out <- ""
for (x in letters) {
  out <- stringr::str_c(out, x)
}
out
#> [1] "abcdefghijklmnopqrstuvwxyz"
```

`str_c` already works with vectors, so simply use `str_c` with the `collapse` argument to return a single string.

```
stringr::str_c(letters, collapse = "")
#> [1] "abcdefghijklmnopqrstuvwxyz"
```

For this I'm going to rename the variable `sd` to something different because `sd` is the name of the function we want to use.

```
x <- sample(100)
sd. <- 0
for (i in seq_along(x)) {
  sd. <- sd. + (x[i] - mean(x)) ^ 2
}
sd. <- sqrt(sd. / (length(x) - 1))
sd.
#> [1] 29
```

We could simply use the `sd` function.

```
sd(x)
#> [1] 29
```

Or if there was a need to use the equation (e.g. for pedagogical reasons), then the functions `mean` and `sum` already work with vectors:

```
sqrt(sum((x - mean(x)) ^ 2) / (length(x) - 1))
#> [1] 29
```

```
x <- runif(100)
out <- vector("numeric", length(x))
out[1] <- x[1]
for (i in 2:length(x)) {
  out[i] <- out[i - 1] + x[i]
}
out
#>   [1]  0.126  1.064  1.865  2.623  3.156  3.703  3.799  4.187  4.359  5.050
#>  [11]  5.725  6.672  6.868  7.836  8.224  8.874  9.688  9.759 10.286 11.050
#>  [21] 11.485 12.038 12.242 12.273 13.242 13.421 14.199 15.085 15.921 16.527
#>  [31] 17.434 17.470 17.601 17.695 18.392 18.797 18.863 18.989 19.927 20.143
#>  [41] 20.809 21.013 21.562 22.389 22.517 22.778 23.066 23.081 23.935 24.349
#>  [51] 25.100 25.819 26.334 27.309 27.670 27.840 28.623 28.654 29.444 29.610
#>  [61] 29.639 30.425 31.250 32.216 32.594 32.769 33.372 34.178 34.215 34.947
```

```
#>  [71] 35.163 35.179 35.307 35.993 36.635 36.963 37.350 38.058 38.755 39.681
#>  [81] 40.140 40.736 40.901 41.468 42.366 42.960 43.792 44.386 45.165 45.562
#>  [91] 46.412 47.154 47.472 47.583 47.685 48.485 48.865 48.917 49.904 50.508
```

The code above is calculating a cumulative sum. Use the function `cumsum`

```
all.equal(cumsum(x),out)
#> [1] TRUE
```

20.2.3 Exercise 3

Combine your function writing and for loop skills:

1. Write a for loop that `prints()` the lyrics to the children's song "Alice the camel".
2. Convert the nursery rhyme "ten in the bed" to a function. Generalise it to any number of people in any sleeping structure.
3. Convert the song "99 bottles of beer on the wall" to a function. Generalise to any number of any vessel containing any liquid on surface.

The lyrics for "Alice the Camel" can be found at http://www.kididdles.com/lyrics/a012.html.

We'll look from five to no humps, and print out a different last line if there are no humps. This uses `cat` instead of `print`, so it looks nicer.

```
humps <- c("five", "four", "three", "two", "one", "no")
for (i in humps) {
  cat(str_c("Alice the camel has ", rep(i, 3), " humps.",
            collapse = "\n"), "\n")
  if (i == "no") {
    cat("Now Alice is a horse.\n")
  } else {
    cat("So go, Alice, go.\n")
  }
  cat("\n")
}
#> Alice the camel has five humps.
#> Alice the camel has five humps.
#> Alice the camel has five humps.
#> So go, Alice, go.
#>
#> Alice the camel has four humps.
#> Alice the camel has four humps.
#> Alice the camel has four humps.
#> So go, Alice, go.
#>
#> Alice the camel has three humps.
#> Alice the camel has three humps.
#> Alice the camel has three humps.
#> So go, Alice, go.
#>
#> Alice the camel has two humps.
#> Alice the camel has two humps.
#> Alice the camel has two humps.
#> So go, Alice, go.
#>
#> Alice the camel has one humps.
```

```
#> Alice the camel has one humps.
#> Alice the camel has one humps.
#> So go, Alice, go.
#>
#> Alice the camel has no humps.
#> Alice the camel has no humps.
#> Alice the camel has no humps.
#> Now Alice is a horse.
```

The lyrics for Ten in the Bed:

```
numbers <- c("ten", "nine", "eight", "seven", "six", "five",
             "four", "three", "two", "one")
for (i in numbers) {
  cat(str_c("There were ", i, " in the bed\n"))
  cat("and the little one said\n")
  if (i == "one") {
    cat("I'm lonely...")
  } else {
    cat("Roll over, roll over\n")
    cat("So they all rolled over and one fell out.\n")
  }
  cat("\n")
}
#> There were ten in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were nine in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were eight in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were seven in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were six in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were five in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
```

```
#> There were four in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were three in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were two in the bed
#> and the little one said
#> Roll over, roll over
#> So they all rolled over and one fell out.
#>
#> There were one in the bed
#> and the little one said
#> I'm lonely...
```

For the bottles of beer, I define a helper function to correctly print the number of bottles.

```
bottles <- function(i) {
  if (i > 2) {
   bottles <- str_c(i - 1, " bottles")
  } else if (i == 2) {
   bottles <- "1 bottle"
  } else {
   bottles <- "no more bottles"
  }
  bottles
}

beer_bottles <- function(n) {
  # should test whether n >= 1.
  for (i in seq(n, 1)) {
     cat(str_c(bottles(i), " of beer on the wall, ", bottles(i), " of beer.\n"))
     cat(str_c("Take one down and pass it around, ", bottles(i - 1),
               " of beer on the wall.\n\n"))
  }
  cat("No more bottles of beer on the wall, no more bottles of beer.\n")
  cat(str_c("Go to the store and buy some more, ", bottles(n), " of beer on the wall.\n"))
}
beer_bottles(3)
#> 2 bottles of beer on the wall, 2 bottles of beer.
#> Take one down and pass it around, 1 bottle of beer on the wall.
#>
#> 1 bottle of beer on the wall, 1 bottle of beer.
#> Take one down and pass it around, no more bottles of beer on the wall.
#>
#> no more bottles of beer on the wall, no more bottles of beer.
#> Take one down and pass it around, no more bottles of beer on the wall.
#>
#> No more bottles of beer on the wall, no more bottles of beer.
#> Go to the store and buy some more, 2 bottles of beer on the wall.
```

20.2.3.1 Exercise 4

It's common to see for loops that don't preallocate the output and instead increase the length of a vector at each step:

```
output <- vector("integer", 0)
for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
}
output
```

> I'll use the package **microbenchmark** to time this. The `microbenchmark` function will run an R expression a number of times and time it.

Define a function that appends to an integer vector.

```
add_to_vector <- function(n) {
  output <- vector("integer", 0)
  for (i in seq_len(n)) {
    output <- c(output, i)
  }
  output
}
microbenchmark(add_to_vector(10000), times = 3)
#> Unit: milliseconds
#>                   expr min  lq mean median  uq max neval
#>  add_to_vector(10000) 145 147  153    149 158 166     3
```

And one that pre-allocates it.

```
add_to_vector_2 <- function(n) {
  output <- vector("integer", n)
  for (i in seq_len(n)) {
    output[[i]] <- i
  }
  output
}
microbenchmark(add_to_vector_2(10000), times = 3)
#> Unit: microseconds
#>                     expr min  lq mean median   uq  max neval
#>  add_to_vector_2(10000) 653 657 1974    661 2634 4607     3
```

The pre-allocated vector is about **100** times faster! You may get different answers, but the longer the vector and the bigger the objects, the more that pre-allocation will outperform appending.

20.3 For loop variations

20.3.1 Exercise 1

Imagine you have a directory full of CSV files that you want to read in. You have their paths in a vector,

`files <- dir("data/", pattern = "\\.csv$", full.names = TRUE)`, and now want to read each one with `read_csv()`. Write the for loop that will load them into a single data frame.

I will pre-allocate a list, read each file as data frame into an element in that list. This creates a list of data frames. I then use `bind_rows` to create a single data frame from the list of data frames.

```
df <- vector("list", length(files))
for (fname in seq_along(files)) {
  df[[i]] <- read_csv(files[[i]])
}
df <- bind_rows(df)
```

20.3.2 Exercise 2

What happens if you use `for (nm in names(x))` and `x` has no names? What if only some of the elements are named? What if the names are not unique?

Let's try it out and see what happens.

When there are no names for the vector, it does not run the code in the loop (it runs zero iterations of the loop):

```
x <- 1:3
print(names(x))
#> NULL
for (nm in names(x)) {
  print(nm)
  print(x[[nm]])
}
```

Note that the length of `NULL` is zero:

```
length(NULL)
#> [1] 0
```

If there only some names, then we get an error if we try to access an element without a name. However, oddly, `nm == ""` when there is no name.

```
x <- c(a = 1, 2, c = 3)
names(x)
#> [1] "a" ""  "c"
```

```
for (nm in names(x)) {
  print(nm)
  print(x[[nm]])
}
#> [1] "a"
#> [1] 1
#> [1] ""
#> Error in x[[nm]]: subscript out of bounds
```

Finally, if there are duplicate names, then `x[[nm]]` will give the *first* element with that name. There is no way to access elements with duplicate names.

```
x <- c(a = 1, a = 2, c = 3)
names(x)
#> [1] "a" "a" "c"
```

```
for (nm in names(x)) {
  print(nm)
  print(x[[nm]])
}
#> [1] "a"
#> [1] 1
```

```
#> [1] "a"
#> [1] 1
#> [1] "c"
#> [1] 3
```

20.3.3 Exercise 3

Write a function that prints the mean of each numeric column in a data frame, along with its name. For example, `show_mean(iris)` would print:

```
show_mean(iris)
#> Sepal.Length: 5.84
#> Sepal.Width:  3.06
#> Petal.Length: 3.76
#> Petal.Width:  1.20
```

> (Extra challenge: what function did I use to make sure that the numbers lined up nicely, even though the variable names had different lengths?)

There may be other functions to do this, but I'll use `str_pad`, and `str_length` to ensure that the space given to the variable names is the same. I messed around with the options to `format` until I got two digits .

```
show_mean <- function(df, digits = 2) {
  # Get max length of any variable in the dataset
  maxstr <- max(str_length(names(df)))
  for (nm in names(df)) {
    if (is.numeric(df[[nm]])) {
      cat(str_c(str_pad(str_c(nm, ":"), maxstr + 1L, side = "right"),
                format(mean(df[[nm]]), digits = digits, nsmall = digits),
                sep = " "),
          "\n")
    }
  }
}
show_mean(iris)
#> Sepal.Length: 5.84
#> Sepal.Width:  3.06
#> Petal.Length: 3.76
#> Petal.Width:  1.20
```

20.3.4 Exercise 4

What does this code do? How does it work?

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) {
    factor(x, labels = c("auto", "manual"))
  }
)
```

```
for (var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

This code mutates the `disp` and `am` columns:

- `disp` is multiplied by 0.0163871
- `am` is replaced by a factor variable.

The code works by looping over a named list of functions. It calls the named function in the list on the column of `mtcars` with the same name, and replaces the values of that column.

E.g. this is a function:

```
trans[["disp"]]
```

This applies the function to the column of `mtcars` with the same name

```
trans[["disp"]](mtcars[["disp"]])
```

20.4 For loops vs. functionals

20.4.1 Exercise 1

Read the documentation for `apply()`. In the 2d case, what two for loops does it generalize.

It generalizes looping over the rows or columns of a matrix or data-frame.

20.4.2 Exercise 2

Adapt `col_summary()` so that it only applies to numeric columns. You might want to start with an `is_numeric()` function that returns a logical vector that has a `TRUE` corresponding to each numeric column.

The original `col_summary()` function is,

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
```

The adapted version is,

```
col_summary2 <- function(df, fun) {
  # test whether each colum is numeric
  numeric_cols <- vector("logical", length(df))
  for (i in seq_along(df)) {
    numeric_cols[[i]] <- is.numeric(df[[i]])
  }
  # indexes of numeric columns
  idxs <- seq_along(df)[numeric_cols]
  # number of numeric columns
  n <- sum(numeric_cols)
  out <- vector("double", n)
  for (i in idxs) {
    out[i] <- fun(df[[i]])
  }
```

```
  out
}
```

Let's test that it works,

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = letters[1:10],
  d = rnorm(10)
)
col_summary2(df, mean)
#> [1]  0.859  0.555  0.000 -0.451
```

20.5 The map functions

20.5.1 Exercise 1

Write code that uses one of the map functions to:

1. Compute the mean of every column in `mtcars`.
2. Determine the type of each column in `nycflights13::flights`.
3. Compute the number of unique values in each column of `iris`.
4. Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.

To calculate the mean of every column in `mtcars`:

```
map_dbl(mtcars, mean)
#>     mpg     cyl    disp      hp    drat      wt    qsec      vs      am
#>  20.091   6.188 230.722 146.688   3.597   3.217  17.849   0.438   0.406
#>    gear    carb
#>   3.688   2.812
```

To calculate the type of every column in `nycflights13::flights`.

```
map(nycflights13::flights, class)
#> $year
#> [1] "integer"
#>
#> $month
#> [1] "integer"
#>
#> $day
#> [1] "integer"
#>
#> $dep_time
#> [1] "integer"
#>
#> $sched_dep_time
#> [1] "integer"
#>
#> $dep_delay
#> [1] "numeric"
#>
#> $arr_time
```

```
#> [1] "integer"
#>
#> $sched_arr_time
#> [1] "integer"
#>
#> $arr_delay
#> [1] "numeric"
#>
#> $carrier
#> [1] "character"
#>
#> $flight
#> [1] "integer"
#>
#> $tailnum
#> [1] "character"
#>
#> $origin
#> [1] "character"
#>
#> $dest
#> [1] "character"
#>
#> $air_time
#> [1] "numeric"
#>
#> $distance
#> [1] "numeric"
#>
#> $hour
#> [1] "numeric"
#>
#> $minute
#> [1] "numeric"
#>
#> $time_hour
#> [1] "POSIXct" "POSIXt"
```

I had to use `map` rather than `map_chr` since the class Though if by type, `typeof` is meant:

```
map_chr(nycflights13::flights, typeof)
#>           year          month            day       dep_time sched_dep_time
#>      "integer"      "integer"      "integer"      "integer"      "integer"
#>      dep_delay       arr_time sched_arr_time      arr_delay        carrier
#>       "double"      "integer"      "integer"       "double"    "character"
#>         flight        tailnum         origin           dest       air_time
#>      "integer"    "character"    "character"    "character"       "double"
#>       distance           hour         minute      time_hour
#>       "double"       "double"       "double"       "double"
```

To calculate the number of unique values in each column of `iris`:

```
map_int(iris, ~ length(unique(.)))
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
#>           35           23           43           22            3
```

To generate 10 random normals for each of $\mu = -10$, 0, 10, and 100:

```
map(c(-10, 0, 10, 100), rnorm, n = 10)
#> [[1]]
#>  [1] -11.27  -9.46  -9.92  -9.44  -9.58 -11.45  -9.06 -10.34 -10.08  -9.96
#>
#> [[2]]
#>  [1]  0.124 -0.998  1.233  0.340 -0.473  0.709 -1.529  0.237 -1.313  0.747
#>
#> [[3]]
#>  [1]  8.44 10.07  9.36  9.15 10.68 11.15  8.31  9.10 11.32 11.10
#>
#> [[4]]
#>  [1] 101.2  98.6 101.4 100.0  99.9 100.4 100.1  99.2  99.5  98.8
```

20.5.2 Exercise 2

How can you create a single vector that for each column in a data frame indicates whether or not it's a factor?

Use `map_lgl` with the function `is.factor`,

```
map_lgl(mtcars, is.factor)
#>   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#> FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

20.5.3 Exercise 3

What happens when you use the map functions on vectors that aren't lists? What does `map(1:5, runif)` do? Why?

The function `map` applies the function to each element of the vector.

```
map(1:5, runif)
#> [[1]]
#> [1] 0.226
#>
#> [[2]]
#> [1] 0.133 0.927
#>
#> [[3]]
#> [1] 0.894 0.204 0.257
#>
#> [[4]]
#> [1] 0.614 0.441 0.316 0.101
#>
#> [[5]]
#> [1] 0.2726 0.6537 0.9279 0.0266 0.5595
```

20.5.4 Exercise 4

What does `map(-2:2, rnorm, n = 5)` do? Why? > What does `map_dbl(-2:2, rnorm, n = 5)` do? Why?

This takes samples of `n = 5` from normal distributions of means -2, -1, 0, 1, and 2, and returns a list with each element a numeric vectors of length 5.

```
map(-2:2, rnorm, n = 5)
#> [[1]]
#> [1] -0.945 -2.821 -2.638 -2.153 -3.416
#>
#> [[2]]
#> [1] -0.393 -0.912 -2.570 -0.687 -0.347
#>
#> [[3]]
#> [1] -0.00796  1.72703  2.08647 -0.35835 -1.44212
#>
#> [[4]]
#> [1] 1.38 1.09 1.16 1.36 0.64
#>
#> [[5]]
#> [1] 1.8914 3.8278 0.0381 2.9460 2.5490
```

However, if we use `map_dbl` it throws an error. `map_dbl` expects the function to return a numeric vector of length one.

```
map_dbl(-2:2, rnorm, n = 5)
#> Error: Result 1 is not a length 1 atomic vector
```

If we wanted a numeric vector, we could use `map` followed by `flatten_dbl`,

```
flatten_dbl(map(-2:2, rnorm, n = 5))
#>  [1] -1.402 -1.872 -3.717 -1.964 -0.993 -0.287 -2.110 -0.851 -1.386 -1.230
#> [11]  0.392  0.470  0.989 -0.714  1.270  1.709  2.047 -0.210  1.380  0.933
#> [21]  2.280  2.330  2.285  2.429  1.879
```

20.5.5 Exercise 5

Rewrite `map(x, function(df) lm(mpg ~ wt, data = df))` to eliminate the anonymous function.

```
map(list(mtcars), ~ lm(mpg ~ wt, data = .))
#> [[1]]
#>
#> Call:
#> lm(formula = mpg ~ wt, data = .)
#>
#> Coefficients:
#> (Intercept)           wt
#>       37.29        -5.34
```

20.6 Dealing with Failure

No exercises

20.7 Mapping over multiple arguments

No exercises

20.8 Walk

No exercises

20.9 Other patterns of for loops

20.9.1 Exercise 1

Implement your own version of `every()` using a for loop. Compare it with `purrr::every()`. What does purrr's version do that your version doesn't?

```
# Use ... to pass arguments to the function
every2 <- function(.x, .p, ...) {
  for (i in .x) {
    if (!.p(i, ...)) {
      # If any is FALSE we know not all of then were TRUE
      return(FALSE)
    }
  }
  # if nothing was FALSE, then it is TRUE
  TRUE
}

every2(1:3, function(x) {x > 1})
#> [1] FALSE
every2(1:3, function(x) {x > 0})
#> [1] TRUE
```

The function `purrr::every` does fancy things with `.p`, like taking a logical vector instead of a function, or being able to test part of a string if the elements of `.x` are lists.

20.9.2 Exercise 2

Create an enhanced `col_sum()` that applies a summary function to every numeric column in a data frame.

Note: this question has a typo. It is referring to `col_summary`.

I will use `map` to apply the function to all the columns, and `keep` to only select numeric columns.

```
col_sum2 <- function(df, f, ...) {
  map(keep(df, is.numeric), f, ...)
}
```

```
col_sum2(iris, mean)
#> $Sepal.Length
#> [1] 5.84
#>
#> $Sepal.Width
#> [1] 3.06
#>
#> $Petal.Length
#> [1] 3.76
#>
```

```
#> $Petal.Width
#> [1] 1.2
```

20.9.3 Exercise 3

Create possible base R equivalent of `col_sum()` is:

```
col_sum3 <- function(df, f) {
  is_num <- sapply(df, is.numeric)
  df_num <- df[, is_num]
  sapply(df_num, f)
}
```

But it has a number of bugs as illustrated with the following inputs:

```
df <- tibble(
  x = 1:3,
  y = 3:1,
  z = c("a", "b", "c")
)


# OK
col_sum3(df, mean)
# Has problems: don't always return numeric vector
col_sum3(df[1:2], mean)
col_sum3(df[1], mean)
col_sum3(df[0], mean)
```

What causes these bugs?

The problem is that `sapply` does not always return numeric vectors. If no columns are selected, instead of returning an empty numeric vector, it returns an empty list. This causes an error since we can't use a list with `[`.

```
sapply(df[0], is.numeric)
#> named list()
```

```
sapply(df[1], is.numeric)
#>    a
#> TRUE
```

```
sapply(df[1:2], is.numeric)
#>    a    b
#> TRUE TRUE
```

Part IV

Model

Chapter 21

Introduction

Chapter 22

Model Basics

22.1 Prerequisites

```
library("tidyverse")
library("modelr")
options(na.action = na.warn)
```

The option `na.action` determines how missing values are handled. It is a function. `na.warn` sets it so that there is a warning if there are any missing values (by default, R will just silently drop them).

22.2 A simple model

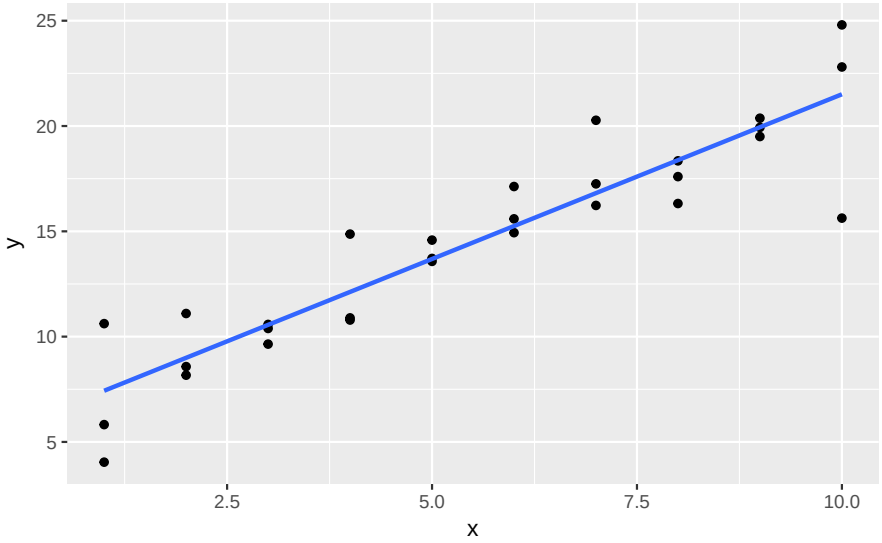
22.2.1 Exercise 1

One downside of the linear model is that it is sensitive to unusual values because the distance incorporates a squared term. Fit a linear model to the simulated data below, and visualize the results. Rerun a few times to generate different simulated datasets. What do you notice about the model?

```
sim1a <- tibble(
  x = rep(1:10, each = 3),
  y = x * 1.5 + 6 + rt(length(x), df = 2)
)
```

Let's run it once and plot the results:

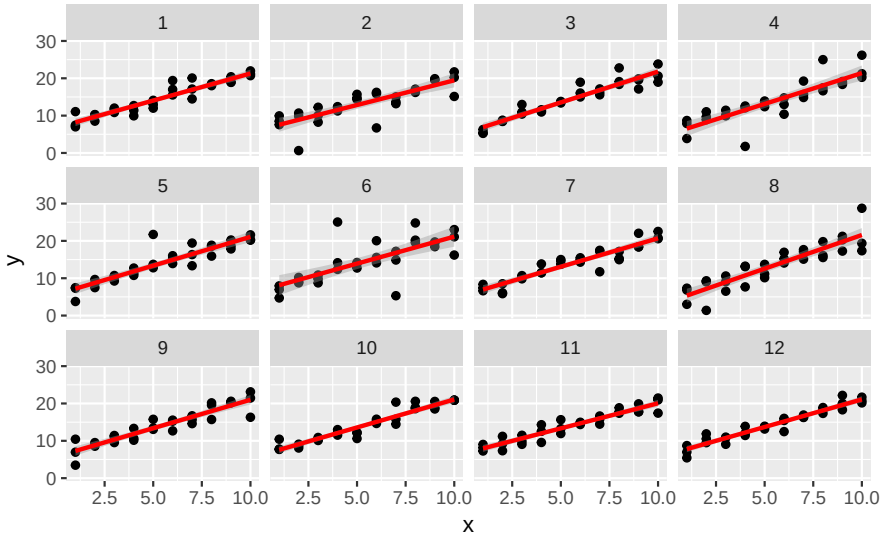
```
ggplot(sim1a, aes(x = x, y = y)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```



To re-run this a few times using `purrr`, and plot using code similar to that in the chapter: There appear to be a few outliers in this data. You can re-rerun this a couple times to see if this hold.

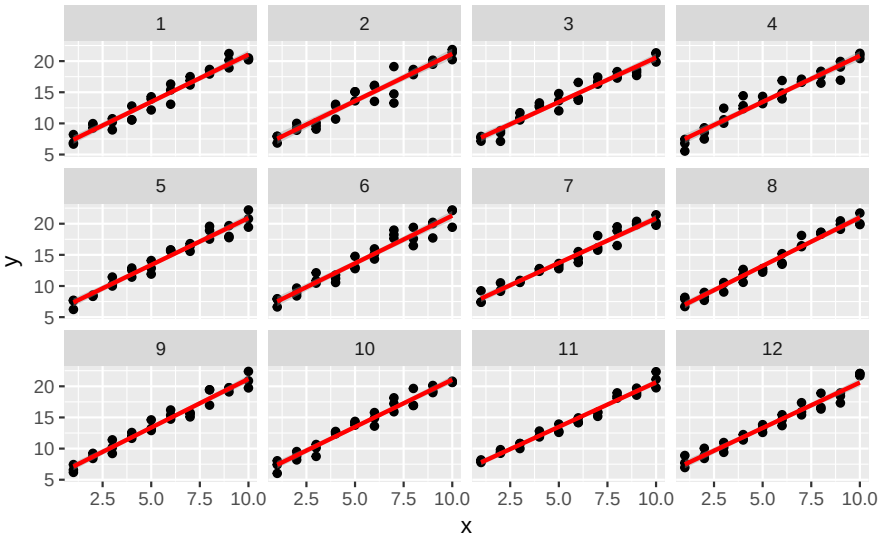
We can also do this slightly more systematically. We will simulate this several times using `purrr` and plot the line using `geom_smooth`:

```
simt <- function(i) {
  tibble(
    x = rep(1:10, each = 3),
    y = x * 1.5 + 6 + rt(length(x), df = 2),
    .id = i
  )
}

sims <- map_df(1:12, simt)

ggplot(sims, aes(x = x, y = y)) +
  geom_point() +
  geom_smooth(method = "lm", color = "red") +
  facet_wrap(~ .id, ncol = 4)
```



What if we did the same things with normal distributions?

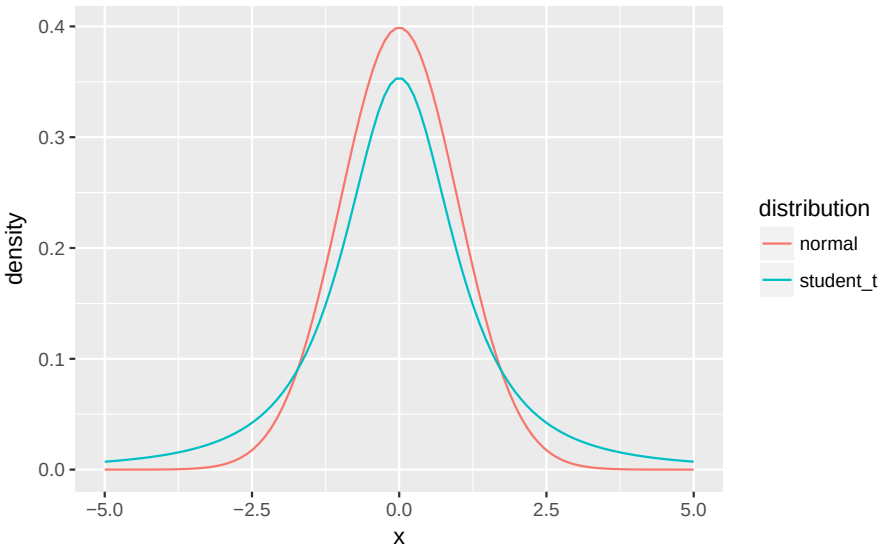
```
sim_norm <- function(i) {
  tibble(
    x = rep(1:10, each = 3),
    y = x * 1.5 + 6 + rnorm(length(x)),
    .id = i
  )
}

simdf_norm <- map_df(1:12, sim_norm)

ggplot(simdf_norm, aes(x = x, y = y)) +
  geom_point() +
  geom_smooth(method = "lm", color = "red") +
  facet_wrap(~ .id, ncol = 4)
```



There are not large outliers, and the slopes are more similar.

The reason for this is that the Student's t-distribution, from which we sample with `rt` has fatter tails than the normal distribution (`rnorm`), which means is assigns larger probability to values further from the center of the distribution.

```
tibble(
  x = seq(-5, 5, length.out = 100),
  normal = dnorm(x),
  student_t = dt(x, df = 2)
) %>%
  gather(distribution, density, -x) %>%
  ggplot(aes(x = x, y = density, colour = distribution)) +
  geom_line()
```



For a normal distribution with mean zero and standard deviation one, the probability of being greater than 2 is,

```
pnorm(2, lower.tail = FALSE)
#> [1] 0.0228
```

For a Student's t distribution with degrees of freedom $= 2$, it is more than 3 times higher,

```
pt(2, df = 2, lower.tail = FALSE)
#> [1] 0.0918
```

22.2.2 Exercise 2

One way to make linear models more robust is to use a different distance measure. For example, instead of root-mean-squared distance, you could use mean-absolute distance:

```
measure_distance <- function(mod, data) {
  diff <- data$y - make_prediction(mod, data)
  mean(abs(diff))
}
```

For the above function to work, we need to define a function `make_prediction` that takes a numeric vector of length two (the intercept and slope) and returns the predictions,

```
make_prediction <- function(mod, data) {
  mod[1] + mod[2] * data$x
}
```

Using the `sim1a` data, the best parameters of the least absolute deviation are:

```
best <- optim(c(0, 0), measure_distance, data = sim1a)
best$par
#> [1] 5.25 1.66
```

Using the `sim1a` data, while the parameters the minimize the least squares objective function are:

```
measure_distance_ls <- function(mod, data) {
  diff <- data$y - (mod[1] + mod[2] * data$x)
  sqrt(mean(diff ^ 2))
}
```

```
best <- optim(c(0, 0), measure_distance_ls, data = sim1a)
best$par
#> [1] 5.87 1.56
```

In practice, you would not use a `optim` to fit this model, you would you an existing implementation. See the `MASS` package's `rlm` and `lqs` functions for more information and functions to fit robust and resistant linear models.

22.2.3 Exercise 3

One challenge with performing numerical optimization is that it's only guaranteed to find a local optimum. What's the problem with optimizing a three parameter model like this?

```
model3 <- function(a, data) {
  a[1] + data$x * a[2] + a[3]
}
```

The problem is that you for any values `a[1] = a1` and `a[3] = a3`, any other values of `a[1]` and `a[3]` where `a[1] + a[3] == (a1 + a3)` will have the same fit.

```
measure_distance_3 <- function(a, data) {
  diff <- data$y - model3(a, data)
  sqrt(mean(diff ^ 2))
}
```

Depending on our starting points, we can find different optimal values:

```
best3a <- optim(c(0, 0, 0), measure_distance_3, data = sim1)
best3a$par
#> [1] 3.367 2.052 0.853
```

```
best3b <- optim(c(0, 0, 1), measure_distance_3, data = sim1)
best3b$par
#> [1] -3.47  2.05  7.69
```

```
best3c <- optim(c(0, 0, 5), measure_distance_3, data = sim1)
best3c$par
#> [1] -1.12  2.05  5.35
```

In fact there are an infinite number of optimal values for this model.

22.3 Visualizing Models

22.3.1 Exercise 1

Instead of using `lm()` to fit a straight line, you can use `loess()` to fit a smooth curve. Repeat the process of model fitting, grid generation, predictions, and visualization on `sim1` using `loess()` instead of `lm()`. How does the result compare to `geom_smooth()`?

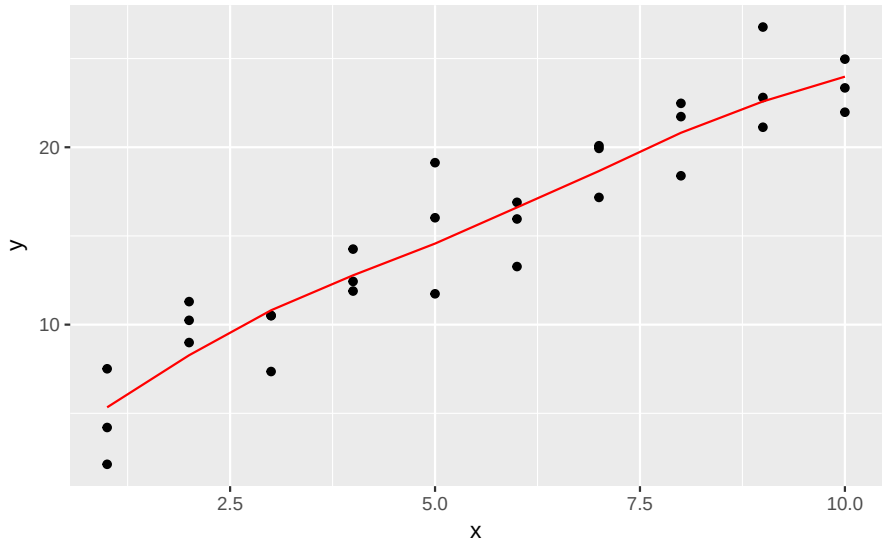
I'll use `add_predictions` and `add_residuals` to add the predictions and residuals from a loess regression to the `sim1` data.

```
sim1_loess <- loess(y ~ x, data = sim1)
sim1_lm <- lm(y ~ x, data = sim1)

grid_loess <- sim1 %>%
```

```
  add_predictions(sim1_loess)

sim1 <- sim1 %>%
  add_residuals(sim1_lm) %>%
  add_predictions(sim1_lm) %>%
  add_residuals(sim1_loess, var = "resid_loess") %>%
  add_predictions(sim1_loess, var = "pred_loess")
```

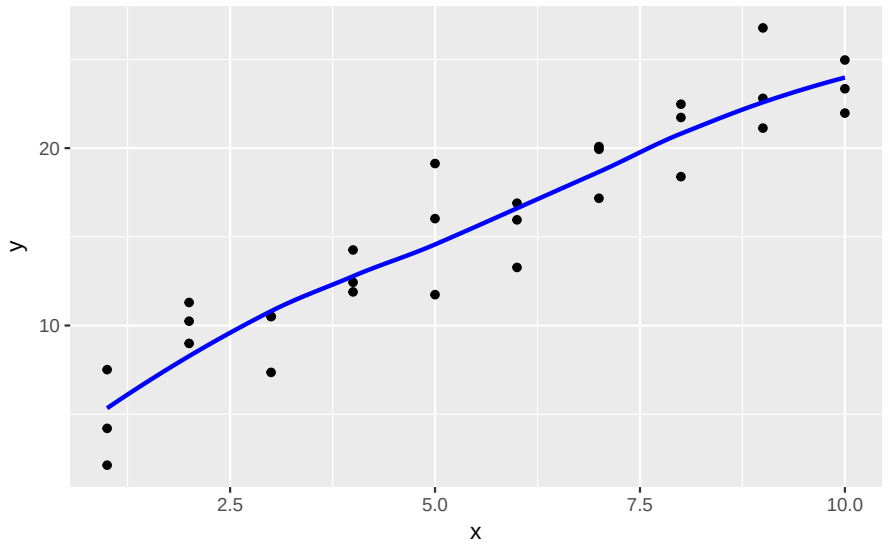
This plots the loess predictions. The loess produces a nonlinear, smooth line through the data.

```
plot_sim1_loess <-
  ggplot(sim1, aes(x = x, y = y)) +
  geom_point() +
  geom_line(aes(x = x, y = pred), data = grid_loess, colour = "red")
plot_sim1_loess
```



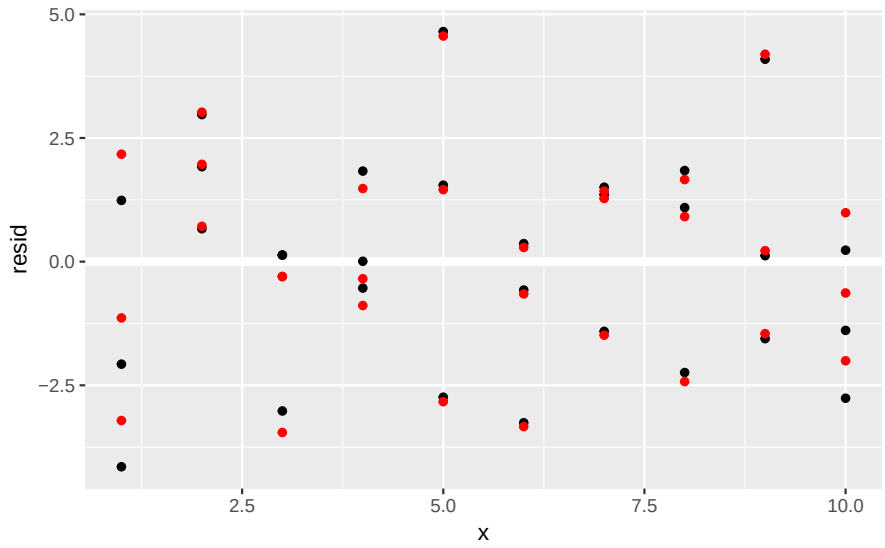
The predictions of loess are the same as the default method for `geom_smooth` because `geom_smooth()` uses `loess()` by default; the message even tells us that.

```
plot_sim1_loess +
  geom_smooth(method = "loess", colour = "blue", se = FALSE, alpha = 0.20)
```



We can plot the residuals (red), and compare them to the residuals from `lm` (black). In general, the loess model has smaller residuals within the sample (out of sample is a different issue, and we haven't considered the uncertainty of these estimates).

```
ggplot(sim1, aes(x = x)) +
  geom_ref_line(h = 0) +
  geom_point(aes(y = resid)) +
  geom_point(aes(y = resid_loess), colour = "red")
```



22.3.2 Exercise 2

`add_predictions()` is paired with `gather_predictions()` and `spread_predictions()`. How do these three functions differ?

The functions `gather_predictions` and `spread_predictions` allow for adding predictions from multiple models at once.

Taking the `sim1_mod` example,

```
sim1_mod <- lm(y ~ x, data = sim1)
grid <- sim1 %>%
  data_grid(x)
```

The function `add_predictions` adds only a single model at a time. To add two models:

```
grid %>%
  add_predictions(sim1_mod, var = "pred_lm") %>%
  add_predictions(sim1_loess, var = "pred_loess")
#> # A tibble: 10 x 3
#>       x pred_lm pred_loess
#>   <int>   <dbl>      <dbl>
#> 1     1    6.27       5.34
#> 2     2    8.32       8.27
#> 3     3   10.4       10.8
#> 4     4   12.4       12.8
#> 5     5   14.5       14.6
#> 6     6   16.5       16.6
#> # ... with 4 more rows
```

The function `gather_predictions` adds predictions from multiple models by stacking the results and adding a column with the model name,

```
grid %>%
  gather_predictions(sim1_mod, sim1_loess)
#> # A tibble: 20 x 3
#>   model        x  pred
#>   <chr>    <int> <dbl>
#> 1 sim1_mod     1  6.27
#> 2 sim1_mod     2  8.32
#> 3 sim1_mod     3 10.4
#> 4 sim1_mod     4 12.4
#> 5 sim1_mod     5 14.5
#> 6 sim1_mod     6 16.5
#> # ... with 14 more rows
```

The function `spread_predictions` adds predictions from multiple models by adding multiple columns (postfixed with the model name) with predictions from each model.

```
grid %>%
  spread_predictions(sim1_mod, sim1_loess)
#> # A tibble: 10 x 3
#>       x sim1_mod sim1_loess
#>   <int>    <dbl>      <dbl>
#> 1     1     6.27       5.34
#> 2     2     8.32       8.27
#> 3     3    10.4       10.8
#> 4     4    12.4       12.8
#> 5     5    14.5       14.6
#> 6     6    16.5       16.6
#> # ... with 4 more rows
```

The function `spread_predictions` is similar to the example which runs `add_predictions` for each model, and is equivalent to running `spread` after running `gather_predictions`:

```
grid %>%
  gather_predictions(sim1_mod, sim1_loess) %>%
  spread(model, pred)
#> # A tibble: 10 x 3
#>       x sim1_loess sim1_mod
#>   <int>      <dbl>    <dbl>
#> 1     1       5.34     6.27
#> 2     2       8.27     8.32
#> 3     3      10.8     10.4
#> 4     4      12.8     12.4
#> 5     5      14.6     14.5
#> 6     6      16.6     16.5
#> # ... with 4 more rows
```

22.3.3 Exercise 3

What does `geom_ref_line()` do? What package does it come from? Why is displaying a reference line in plots showing residuals useful and important?

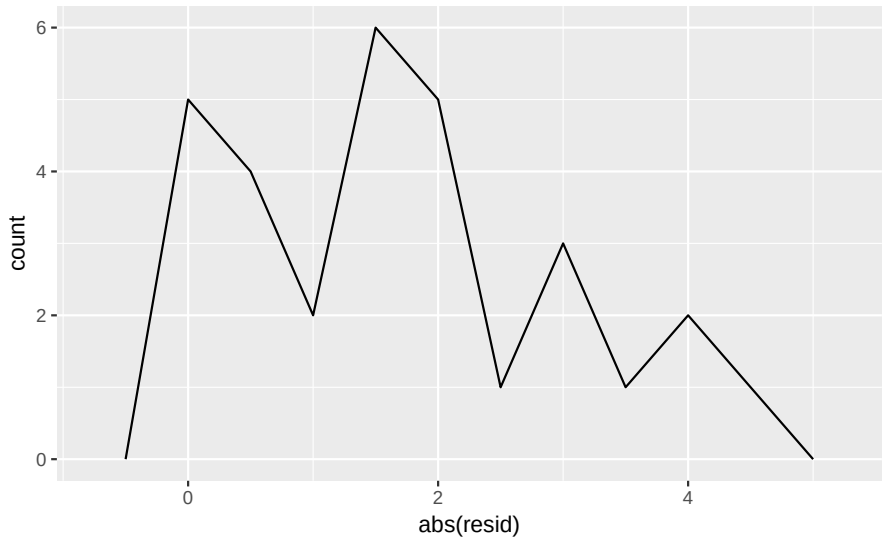
The geom `geom_ref_line()` adds as reference line to a plot. It is equivalent to running `geom_hline` or `geom_vline` with default settings that are useful for visualizing models. Putting a reference line at zero for residuals is important because good models (generally) should have residuals centered at zero, with approximately the same variance (or distribution) over the support of x, and no correlation. A zero reference line makes it easier to judge these characteristics visually.

22.3.4 Exercise 4

Why might you want to look at a frequency polygon of absolute residuals? What are the pros and cons compared to looking at the raw residuals?

Showing the absolute values of the residuals makes it easier to view the spread of the residuals. The model assumes the the residuals have mean zero, and using the absolute values of the residuals effectively doubles the number of residuals.

```
sim1_mod <- lm(y ~ x, data = sim1)

sim1 <- sim1 %>%
  add_residuals(sim1_mod)

ggplot(sim1, aes(x = abs(resid))) +
  geom_freqpoly(binwidth = 0.5)
```



However, using the absolute values of residuals throws away information about the sign, meaning that the frequency polygon cannot show whether the model systematically over- or under-estimates the residuals.

22.4 Formulas and Model Families

22.4.1 Exercise 1

What happens if you repeat the analysis of `sim2` using a model without an intercept. What happens to the model equation? What happens to the predictions?

To run a model without an intercept, add `- 1` or `+ 0` to the right-hand-side o f the formula:

```
mod2a <- lm(y ~ x - 1, data = sim2)
```

```
mod2 <- lm(y ~ x, data = sim2)
```

The predictions are exactly the same in the models with and without an intercept:

```
grid <- sim2 %>%
  data_grid(x) %>%
  spread_predictions(mod2, mod2a)
grid
#> # A tibble: 4 x 3
#>   x      mod2 mod2a
#>   <chr> <dbl> <dbl>
#> 1 a      1.15  1.15
#> 2 b      8.12  8.12
#> 3 c      6.13  6.13
#> 4 d      1.91  1.91
```

22.4.2 Exercise 2

Use `model_matrix()` to explore the equations generated for the models I fit to `sim3` and `sim4`. Why is `*` a good shorthand for interaction?

For `x1 * x2` when `x2` is a categorical variable produces indicator variables `x2b`, `x2c`, `x2d` and variables `x1:x2b`, `x1:x2c`, and `x1:x2d` which are the products of `x1` and `x2*` variables:

```
x3 <- model_matrix(y ~ x1 * x2, data = sim3)
x3
#> # A tibble: 120 x 8
#>   `(Intercept)`    x1   x2b   x2c   x2d `x1:x2b` `x1:x2c` `x1:x2d`
#>           <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>    <dbl>    <dbl>
#> 1            1.    1.    0.    0.    0.       0.       0.       0.
#> 2            1.    1.    0.    0.    0.       0.       0.       0.
#> 3            1.    1.    0.    0.    0.       0.       0.       0.
#> 4            1.    1.    1.    0.    0.       1.       0.       0.
#> 5            1.    1.    1.    0.    0.       1.       0.       0.
#> 6            1.    1.    1.    0.    0.       1.       0.       0.
#> # ... with 114 more rows
```

We can confirm that the variables `x1:x2b` is the product of `x1` and `x2b`,

```
all(x3[["x1:x2b"]] == (x3[["x1"]] * x3[["x2b"]]))
#> [1] TRUE
```

and similarly for `x1:x2c` and `x2c`, and `x1:x2d` and `x2d`:

```
all(x3[["x1:x2c"]] == (x3[["x1"]] * x3[["x2c"]]))
#> [1] TRUE
all(x3[["x1:x2d"]] == (x3[["x1"]] * x3[["x2d"]]))
#> [1] TRUE
```

For `x1 * x2` where both `x1` and `x2` are continuous variables, `model_matrix()` creates variables `x1`, `x2`, and `x1:x2`:

```
x4 <- model_matrix(y ~ x1 * x2, data = sim4)
x4
#> # A tibble: 300 x 4
#>   `(Intercept)`    x1     x2 `x1:x2`
#>           <dbl> <dbl>  <dbl>   <dbl>
#> 1            1.   -1. -1.00    1.00
#> 2            1.   -1. -1.00    1.00
#> 3            1.   -1. -1.00    1.00
#> 4            1.   -1. -0.778   0.778
#> 5            1.   -1. -0.778   0.778
#> 6            1.   -1. -0.778   0.778
#> # ... with 294 more rows
```

Confirm that `x1:x2` is the product of the `x1` and `x2`,

```
all(x4[["x1"]] * x4[["x2"]] == x4[["x1:x2"]])
#> [1] TRUE
```

The asterisk `*` is good shorthand for an interaction since an interaction between `x1` and `x2` includes terms for `x1`, `x2`, and the product of `x1` and `x2`.

22.4.3 Exercise 3

Using the basic principles, convert the formulas in the following two models into functions. (Hint: start by converting the categorical variable into 0-1 variables.)

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

```
model_matrix_mod1 <- function(.data) {
  mutate(.data,
         `x2b` = as.numeric(x2 == "b"),
         `x2c` = as.numeric(x2 == "c"),
         `x2d` = as.numeric(x2 == "d"),
         `x1:x2b` = x1 * x2b,
         `x1:x2c` = x1 * x2c,
         `x1:x2d` = x1 * x2d) %>%
    select(x1, x2b, x2c, x2d, `x1:x2b`, `x1:x2c`, `x1:x2d`)
}
```

```
model_matrix_mod1(sim3)
#> # A tibble: 120 x 7
#>      x1   x2b   x2c   x2d `x1:x2b` `x1:x2c` `x1:x2d`
#>   <int> <dbl> <dbl> <dbl>    <dbl>    <dbl>    <dbl>
#> 1     1    0.    0.    0.       0.       0.       0.
#> 2     1    0.    0.    0.       0.       0.       0.
#> 3     1    0.    0.    0.       0.       0.       0.
#> 4     1    1.    0.    0.       1.       0.       0.
#> 5     1    1.    0.    0.       1.       0.       0.
#> 6     1    1.    0.    0.       1.       0.       0.
#> # ... with 114 more rows
```

```
model_matrix_mod2 <- function(.data) {
  mutate(.data, `x1:x2` = x1 * x2) %>%
    select(x1, x2, `x1:x2`)
}
model_matrix_mod2(sim4)
#> # A tibble: 300 x 3
#>      x1     x2 `x1:x2`
#>   <dbl>  <dbl>   <dbl>
#> 1   -1. -1.00    1.00
#> 2   -1. -1.00    1.00
#> 3   -1. -1.00    1.00
#> 4   -1. -0.778   0.778
#> 5   -1. -0.778   0.778
#> 6   -1. -0.778   0.778
#> # ... with 294 more rows
```

A more general function for model `mod1` is:

```
model_matrix_mod1 <- function(x1, x2) {
  out <- tibble(x1 = x1)
  # find levels of x2
  x2 <- as.factor(x2)
  x2lvls <- levels(x2)
  # create an indicator variable for each level
  for (lvl in x2lvls[2:nlevels(x2)]) {
    out[[str_c("x2", lvl)]] <- as.numeric(x2 == lvl)
  }
  # create interactions for each level
  for (lvl in x2lvls[2:nlevels(x2)]) {
    out[[str_c("x1:x2", lvl)]] <- (x2 == lvl) * x1
  }
  out
```

```
}
```

```
model_matrix_mod2 <- function(x1, x2) {
  out <- tibble(x1 = x1,
                x2 = x2,
                `x1:x2` = x1 * x2)
}
```

22.4.4 Exercise 4

For `sim4`, which of `mod1` and `mod2` is better? I think `mod2` does a slightly better job at removing patterns, but it's pretty subtle. Can you come up with a plot to support my claim?

Estimate models `mod1` and `mod2` on `sim4`,

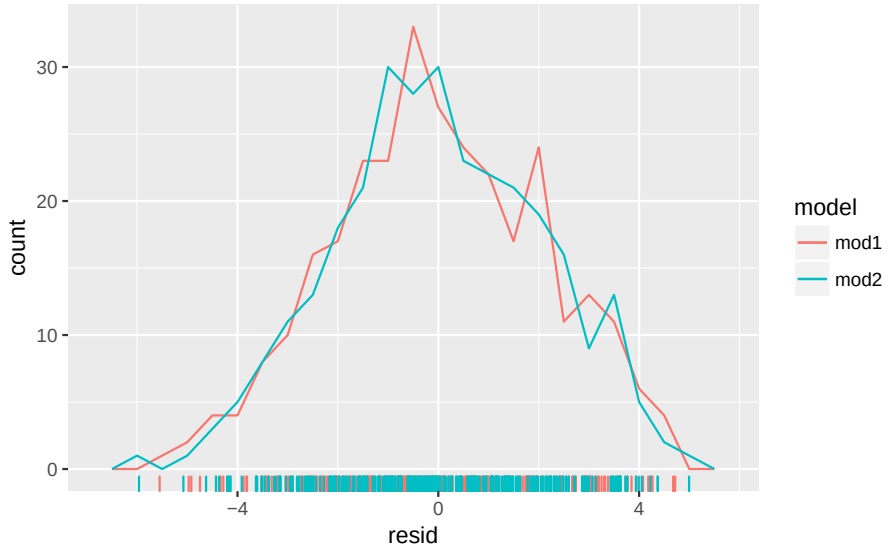
```
mod1 <- lm(y ~ x1 + x2, data = sim4)
mod2 <- lm(y ~ x1 * x2, data = sim4)
```

and add the residuals from these models to the `sim4` data,

```
sim4_mods <- gather_residuals(sim4, mod1, mod2)
```

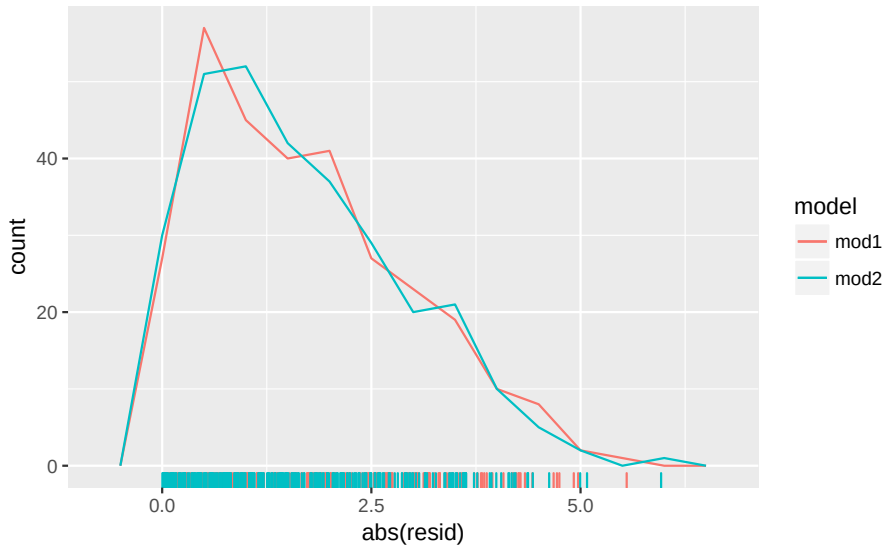
Frequency plots of both the residuals,

```
ggplot(sim4_mods, aes(x = resid, color = model)) +
  geom_freqpoly(binwidth = 0.5) +
  geom_rug()
```



and the absolute values of the residuals,

```
ggplot(sim4_mods, aes(x = abs(resid), color = model)) +
  geom_freqpoly(binwidth = 0.5) +
  geom_rug()
```



does not show much difference in the residuals between the models. However, `mod2` appears to have fewer residuals in the tails of the distribution between 2.5 and 5 (although the most extreme residuals are from `mod2`.

This is confirmed by checking the standard deviation of the residuals of these models,

```
sim4_mods %>%
  group_by(model) %>%
  summarise(resid = sd(resid))
#> # A tibble: 2 x 2
#>   model resid
#>   <chr> <dbl>
#> 1 mod1   2.10
#> 2 mod2   2.07
```

The standard deviation of the residuals of `mod2` is smaller than that of `mod1`.

22.5 Missing values

No exercises

22.6 Other model families

No exercises

Chapter 23

Model Building

23.1 Introduction

```
library(tidyverse)
library(modelr)
options(na.action = na.warn)
library("broom")

library(nycflights13)
library(lubridate)
```

23.2 Why are low quality diamonds more expensive?

```
diamonds2 <- diamonds %>%
  filter(carat <= 2.5) %>%
  mutate(lprice = log2(price), lcarat = log2(carat))

mod_diamond2 <- lm(lprice ~ lcarat + color + cut + clarity, data = diamonds2)
```

23.2.1 Exercises

23.2.1.1 Exercise 1

In the plot of `lcarat` vs. `lprice`, there are some bright vertical strips. What do they represent?

The distribution of diamonds has more diamonds at round or otherwise human friendly numbers (fractions).

23.2.1.2 Exercise 2

If `log(price) = a_0 + a_1 * log(carat)`, what does that say about the relationship between `price` and `carat`?

An 1% increase in carat is associated with an a_1% increase in price.

23.2.1.3 Exercise 3

Extract the diamonds that have very high and very low residuals. Is there anything unusual about these diamonds? Are the particularly bad or good, or do you think these are pricing errors?

This was already discussed in the text. I don't see anything either.

23.2.1.4 Exercise 4

Does the final model, `mod_diamonds2`, do a good job of predicting diamond prices? Would you trust it to tell you how much to spend if you were buying a diamond?

```
diamonds2 %>%
  add_predictions(mod_diamond2) %>%
  add_residuals(mod_diamond2) %>%
  summarise(sq_err = sqrt(mean(resid^2)),
            abs_err = mean(abs(resid)),
            p975_err = quantile(resid, 0.975),
            p025_err = quantile(resid, 0.025))
#> # A tibble: 1 x 4
#>   sq_err abs_err p975_err p025_err
#>    <dbl>   <dbl>    <dbl>    <dbl>
#> 1  0.192   0.149    0.384   -0.369
```

The average squared and absolute errors are $2^0.19 = 1.14$ and $2^0.10$ so on average, the error is $\pm 10 - -15\%$. And the 95% range of residuals is about $2^0.37 = 1.3$ so within $\pm 30\%$. This doesn't seem terrible to me.

23.2.2 What affects the number of daily flights?

```
library("nycflights13")
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  group_by(date) %>%
  summarise(n = n())
daily
#> # A tibble: 365 x 2
#>   date           n
#>   <date>     <int>
#> 1 2013-01-01   842
#> 2 2013-01-02   943
#> 3 2013-01-03   914
#> 4 2013-01-04   915
#> 5 2013-01-05   720
#> 6 2013-01-06   832
#> # ... with 359 more rows

daily <- daily %>%
  mutate(wday = wday(date, label = TRUE))

term <- function(date) {
  cut(date,
    breaks = ymd(20130101, 20130605, 20130825, 20140101),
    labels = c("spring", "summer", "fall")
  )
```

```
}

daily <- daily %>%
  mutate(term = term(date))

mod <- lm(n ~ wday, data = daily)

daily <- daily %>%
  add_residuals(mod)

mod1 <- lm(n ~ wday, data = daily)
mod2 <- lm(n ~ wday * term, data = daily)
```

23.2.3 Exercises

23.2.3.1 Exercise 1

Use your Google sleuthing skills to brainstorm why there were fewer than expected flights on Jan 20, May 26, and Sep 1. (Hint: they all have the same explanation.) How would these days generalize to another year?

These are the Sundays before Monday holidays Martin Luther King Day, Memorial Day, and Labor Day.

23.2.3.2 Exercise 2

```
daily %>%
  top_n(3, resid)
#> # A tibble: 3 x 5
#>   date           n wday  term  resid
#>   <date>     <int> <ord> <fct> <dbl>
#> 1 2013-11-30   857 Sat   fall  112.
#> 2 2013-12-01   987 Sun   fall   95.5
#> 3 2013-12-28   814 Sat   fall   69.4
```

23.2.3.3 Exercise 3

Create a new variable that splits the `wday` variable into terms, but only for Saturdays, i.e. it should have `Thurs`, `Fri`, but `Sat-summer`, `Sat-spring`, `Sat-fall` How does this model compare with the model with every combination of `wday` and `term`?

I'll use the function `case_when` to do this, though there are other ways which it could be solved.

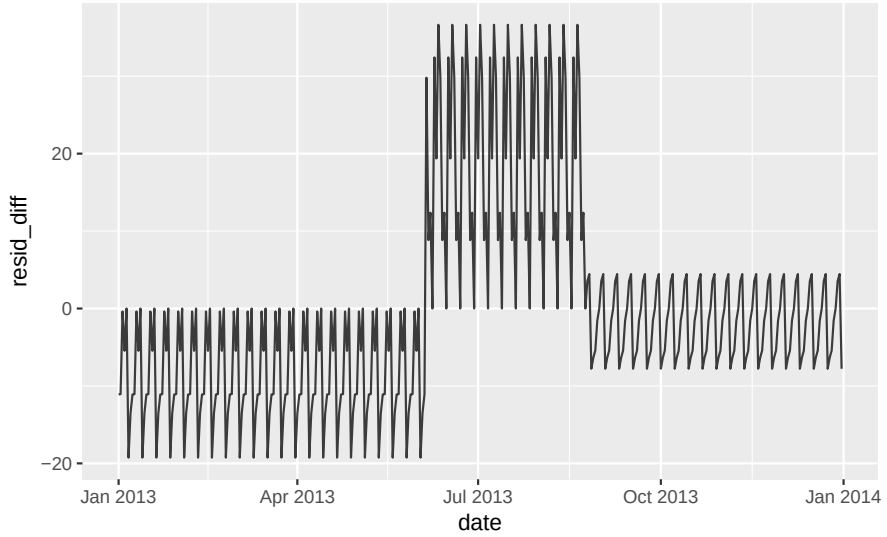
```
daily <- daily %>%
  mutate(wday2 =
         case_when(.$wday == "Sat" & .$term == "summer" ~ "Sat-summer",
         .$wday == "Sat" & .$ term == "fall" ~ "Sat-fall",
         .$wday == "Sat" & .$term == "spring" ~ "Sat-spring",
         TRUE ~ as.character(.$wday)))
```

```
mod4 <- lm(n ~ wday2, data = daily)

daily %>%
```

```
  gather_residuals(sat_term = mod4, all_interact = mod2) %>%
  ggplot(aes(date, resid, colour = model)) +
    geom_line(alpha = 0.75)
```

I think the overlapping plot is hard to understand. If we are interested in the differences, it is better to plot the differences directly. In this code I use `spread_residuals` to add one *column* per model, rather than `gather_residuals` which creates a new row for each model.

```
daily %>%
  spread_residuals(sat_term = mod4, all_interact = mod2) %>%
  mutate(resid_diff = sat_term - all_interact) %>%
  ggplot(aes(date, resid_diff)) +
    geom_line(alpha = 0.75)
```



The model with terms x Saturday has higher residuals in the fall, and lower residuals in the spring than the model with all interactions.

Using overall model comparison terms, `mod4` has a lower R^2 and regression standard error, $\hat{\sigma}$, despite using fewer variables. More importantly for prediction purposes, it has a higher AIC - which is an estimate of the out of sample error.

```
glance(mod4) %>% select(r.squared, sigma, AIC, df)
#>   r.squared sigma  AIC df
#> 1     0.736  47.4 3863  9
```

```
glance(mod2) %>% select(r.squared, sigma, AIC, df)
#>   r.squared sigma  AIC df
#> 1     0.757  46.2 3856 21
```

23.2.3.4 Exercise 4

Create a new `wday` variable that combines the day of week, term (for Saturdays), and public holidays. What do the residuals of that model look like?

The question is unclear how to handle the public holidays. We could include a dummy for all public holidays? or the Sunday before public holidays?

Including a level for the public holidays themselves is insufficient because (1) public holiday's effects on travel varies dramatically, (2) the effect can occur on the day itself or the day before and after, and (3) with Thanksgiving and Christmas there are increases in travel as well.

```
daily <- daily %>%
  mutate(wday3 =
         case_when(
           .$date %in% lubridate::ymd(c(20130101, # new years
                                        20130121, # mlk
                                        20130218, # presidents
                                        20130527, # memorial
                                        20130704, # independence
                                        20130902, # labor
                                        20131028, # columbus
                                        20131111, # veterans
                                        20131128, # thanksgiving
                                        20131225)) ~
             "holiday",
           .$wday == "Sat" & .$term == "summer" ~ "Sat-summer",
           .$wday == "Sat" & .$ term == "fall" ~ "Sat-fall",
           .$wday == "Sat" & .$term == "spring" ~ "Sat-spring",
           TRUE ~ as.character(.$wday)))

mod5 <- lm(n ~ wday3, data = daily)

daily %>%
  spread_residuals(mod5) %>%
  arrange(desc(abs(resid))) %>%
  slice(1:20) %>% select(date, wday, resid)
#> # A tibble: 20 x 3
#>   date       wday  resid
#>   <date>     <ord> <dbl>
#> 1 2013-11-28 Thu   -332.
#> 2 2013-11-29 Fri   -306.
#> 3 2013-12-25 Wed   -244.
#> 4 2013-07-04 Thu   -229.
#> 5 2013-12-24 Tue   -190.
#> 6 2013-12-31 Tue   -175.
```

```
#> # ... with 14 more rows
```

23.2.3.5 Exercise 5

What happens if you fit a day of week effect that varies by month (i.e. `n ~ wday * month`)? Why is this not very helpful?

There are only 4-5 observations per parameter since only there are only 4-5 weekdays in a given month.

23.2.3.6 Exercise 6

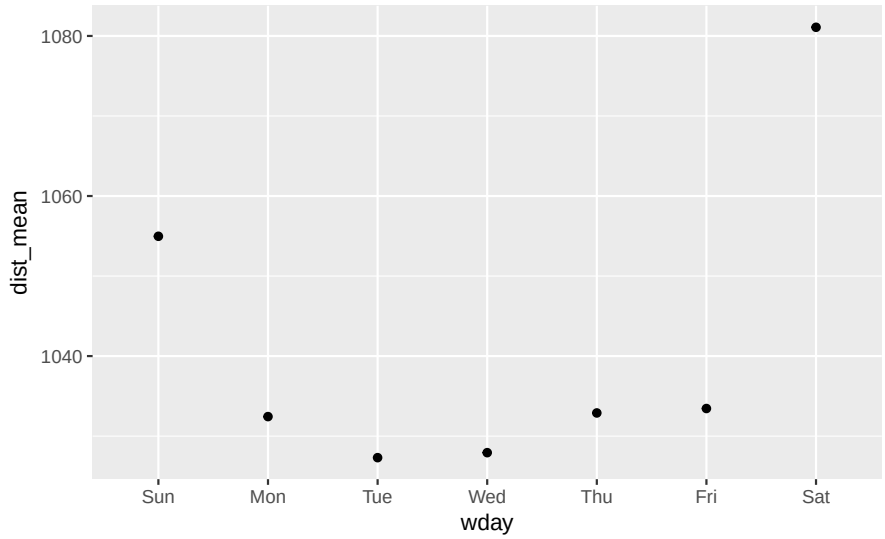
It will estimate a smooth seasonal trend (`ns(date, 5)`) with a day of the week cyclicality, (`wday`). It probably will not be effective since

23.2.3.7 Exercise 7

We hypothesized that people leaving on Sundays are more likely to be business travelers who need to be somewhere on Monday. Explore that hypothesis by seeing how it breaks down based on distance and time: if it's true, you'd expect to see more Sunday evening flights to places that are far away.

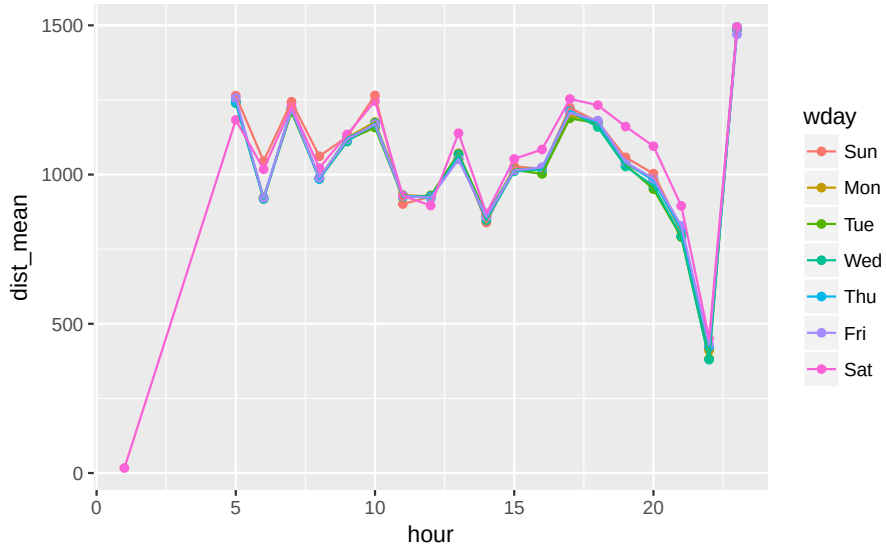
Looking at only day of the week, we see that Sunday flights are on average longer than the rest of the day of the week flights, but not as long as Saturday flights (perhaps vacation flights?).

```
flights %>%
  mutate(date = make_date(year, month, day),
         wday = wday(date, label = TRUE)) %>%
  group_by(wday) %>%
  summarise(dist_mean =  mean(distance),
            dist_median = median(distance)) %>%
  ggplot(aes(y = dist_mean, x = wday)) +
  geom_point()
```



However, breaking it down by hour, I don't see much evidence at first. Conditional on hour, the distance of Sunday flights seems similar to that of other days (excluding Saturday):

```
flights %>%
  mutate(date = make_date(year, month, day),
         wday = wday(date, label = TRUE)) %>%
  group_by(wday, hour) %>%
  summarise(dist_mean =  mean(distance),
            dist_median = median(distance)) %>%
  ggplot(aes(y = dist_mean, x = hour, colour = wday)) +
  geom_point() +
  geom_line()
```



Can someone think of a better way to check this?

23.2.3.8 Exercise 8

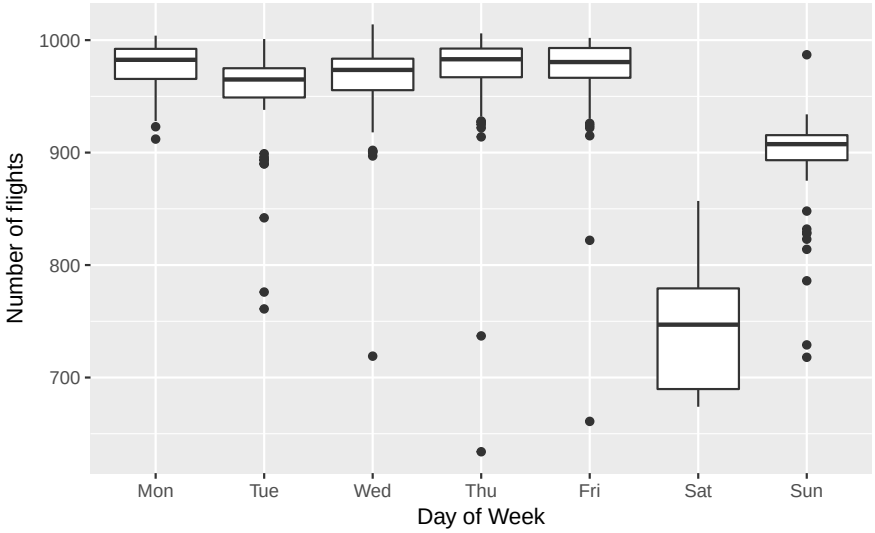
It's a little frustrating that Sunday and Saturday are on separate ends of the plot. Write a small function to set the levels of the factor so that the week starts on Monday.

See the chapter Factors for the function `fct_relevel`. I use `fct_relevel` to put all levels in-front of the first level ("Sunday").

```
monday_first <- function(x) {
  forcats::fct_relevel(x, levels(x)[-1])
}
```

Now Monday is the first day of the week,

```
daily <- daily %>%
  mutate(wday = wday(date, label = TRUE))
ggplot(daily, aes(monday_first(wday), n)) +
  geom_boxplot() +
  labs(x = "Day of Week", y = "Number of flights")
```


1000
900
800
700
Number of flights
Mon
Tue
Wed
Thu
Fri
Sat
Sun
Day of Week

Chapter 24

Many Models

24.1 Introduction

```
library("modelr")
library("tidyverse")
#> -- Attaching packages ----------------------------------------------- tidyverse 1.2.1 --
#> ✓ ggplot2 2.2.1     ✓ purrr   0.2.4
#> ✓ tibble  1.4.2     ✓ dplyr   0.7.4
#> ✓ tidyr   0.8.0     ✓ stringr 1.3.0
#> ✓ readr   1.1.1     ✓ forcats 0.3.0
#> -- Conflicts -------------------------------------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library("gapminder")
```

24.2 Gapminder

24.2.1 Exercise 1

A linear trend seems to be slightly too simple for the overall trend. Can you do better with a quadratic polynomial? How can you interpret the coefficients of the quadratic? (Hint you might want to transform year so that it has mean zero.)

The following code replicates the analysis in the chapter but the function `country_model` is replaced with a regression that includes the year squared.

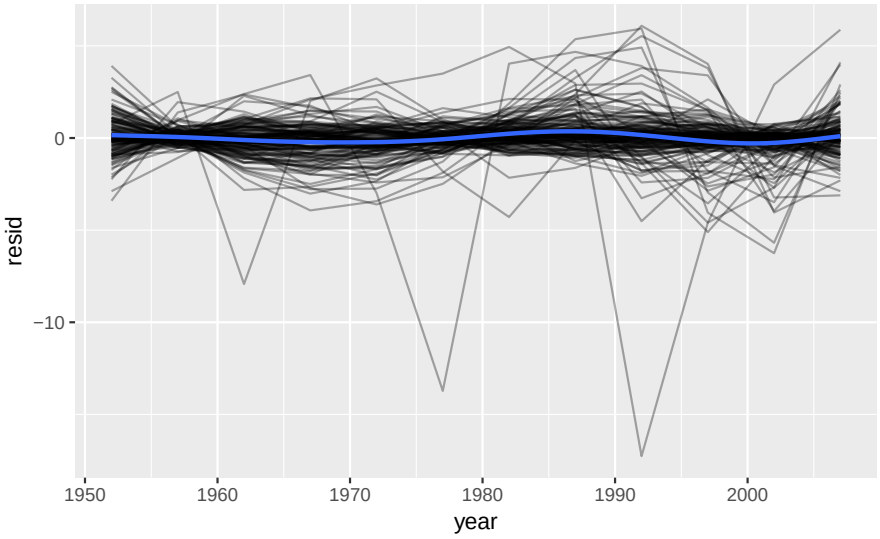
```
lifeExp ~ poly(year, 2)
```

```
country_model <- function(df) {
  lm(lifeExp ~ poly(year - median(year), 2), data = df)
}

by_country <- gapminder %>%
  group_by(country, continent) %>%
  nest()
```

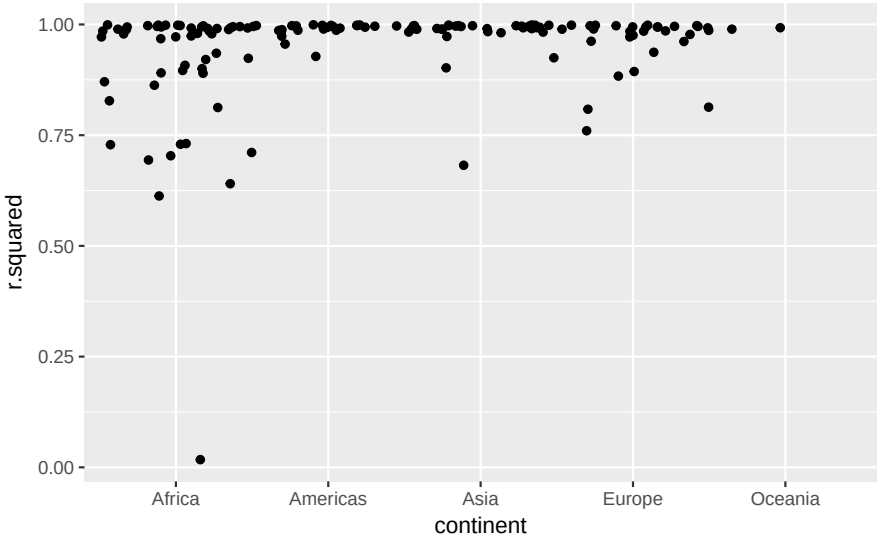
```
by_country <- by_country %>%
  mutate(model = map(data, country_model))
```

```
by_country <- by_country %>%
  mutate(
    resids = map2(data, model, add_residuals)
  )
by_country
#> # A tibble: 142 x 5
#>   country     continent data              model    resids
#>   <fct>       <fct>     <list>            <list>   <list>
#> 1 Afghanistan Asia      <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
#> 2 Albania     Europe    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
#> 3 Algeria     Africa    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
#> 4 Angola      Africa    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
#> 5 Argentina   Americas  <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
#> 6 Australia   Oceania   <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
#> # ... with 136 more rows
```

```
unnest(by_country, resids) %>%
ggplot(aes(year, resid)) +
  geom_line(aes(group = country), alpha = 1 / 3) +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'gam'
```




```
by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance, .drop = TRUE) %>%
  ggplot(aes(continent, r.squared)) +
  geom_jitter(width = 0.5)
```

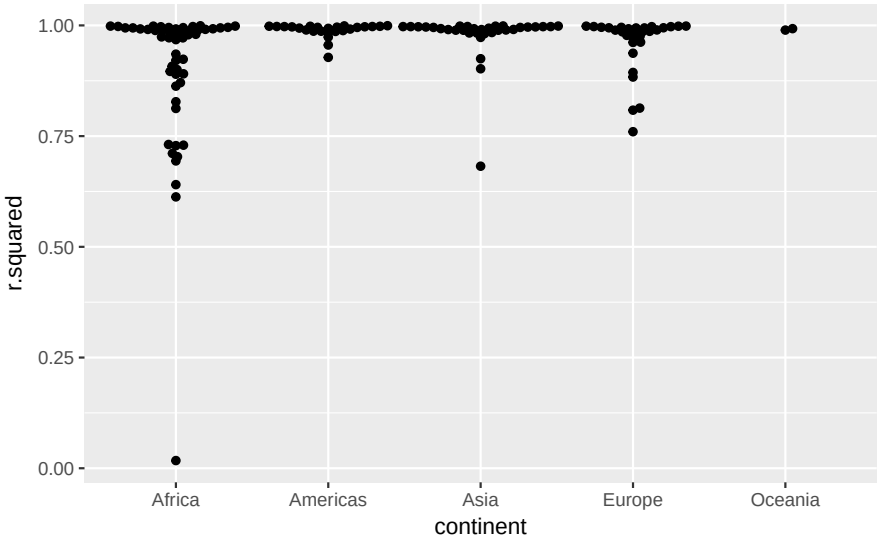


24.2.1.1 Exercise 2

Explore other methods for visualizing the distribution of R^2 per continent. You might want to try the **ggbeeswarm** package, which provides similar methods for avoiding overlaps as jitter, but uses deterministic methods.

See exercise 7.5.1.1.6 for more on **ggbeeswarm**

```
library("ggbeeswarm")
by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance, .drop = TRUE) %>%
  ggplot(aes(continent, r.squared)) +
  geom_beeswarm()
```



24.3 Creating list-columns

24.3.1 Exercises

24.3.1.1 Exercise 1

List all the functions that you can think of that take a atomic vector and return a list.

E.g. Many of the **stringr** functions.

24.3.1.2 Exercise 2

Brainstorm useful summary functions that, like `quantile()`, return multiple values.

Some examples of summary functions that return multiple values are `range` and `fivenum`.

24.3.1.3 Exercise 3

What's missing in the following data frame? How does `quantile()` return that missing piece? Why isn't that helpful here?

```
mtcars %>%
  group_by(cyl) %>%
  summarise(q = list(quantile(mpg))) %>%
  unnest()
#> # A tibble: 15 x 2
#>     cyl     q
#>   <dbl> <dbl>
#> 1    4.  21.4
#> 2    4.  22.8
#> 3    4.  26.0
#> 4    4.  30.4
#> 5    4.  33.9
#> 6    6.  17.8
#> # ... with 9 more rows
```

The particular quantiles of the values are missing, e.g. `0%`, `25%`, `50%`, `75%`, `100%`. `quantile()` returns these in the names of the vector.

```
quantile(mtcars$mpg)
#>   0%  25%  50%  75% 100%
#> 10.4 15.4 19.2 22.8 33.9
```

Since the `unnest` function drops the names of the vector, they aren't useful here.

24.3.1.4 Exercise 4

What does this code do? Why might might it be useful?

```
mtcars %>%
  group_by(cyl) %>%
  summarise_each(funs(list))
#> `summarise_each()` is deprecated.
#> Use `summarise_all()`, `summarise_at()` or `summarise_if()` instead.
#> To map `funs` over all variables, use `summarise_all()`
```

```
#> # A tibble: 3 x 11
#>     cyl mpg        disp   hp     drat  wt    qsec  vs    am    gear  carb
#>   <dbl> <list>     <list> <list> <lis> <lis> <lis> <lis> <lis> <lis> <lis>
#> 1    4. <dbl [11]> <dbl ~ <dbl ~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~
#> 2    6. <dbl [7]>  <dbl ~ <dbl ~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~
#> 3    8. <dbl [14]> <dbl ~ <dbl ~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~
```

It creates a data frame in which each row corresponds to a value of `cyl`, and each observation for each column (other than `cyl`) is a vector of all the values of that column for that value of `cyl`. It seems like it should be useful to have all the observations of each variable for each group, but off the top of my head, I can't think of a specific use for this. But, it seems that it may do many things that `dplyr::do` does.

24.4 Simplifying list-columns

24.4.1 Exercises

24.4.1.1 Exercise 1

Why might the `lengths()` function be useful for creating atomic vector columns from list-columns?

The `lengths()` function gets the lengths of each element in a list. It could be useful for testing whether all elements in a list-column are the same length. You could get the maximum length to determine how many atomic vector columns to create. It is also a replacement for something like `map_int(x, length)` or `sapply(x, length)`.

24.4.1.2 Exercise 2

List the most common types of vector found in a data frame. What makes lists different?

The common types of vectors in data frames are:

- `logical`
- `numeric`
- `integer`
- `character`
- `factor`

All of the common types of vectors in data frames are atomic. Lists are not atomic (they can contain other lists and other vectors).

Part V

Communicate

Chapter 25

Introduction

Chapter 26

R Markdown

26.1 R Markdown Basics

26.1.1 Exercises

26.1.1.1 Exercise 1

Create a new notebook using *File > New File > R Notebook*. Read the instructions. Practice running the chunks. Verify that you can modify the code, re-run it, and see modified output.

This exercise is left to the reader.

26.1.1.2 Exercise 2

Create a new R Markdown document with *File > New File > R Markdown …* Knit it by clicking the appropriate button. Knit it by using the appropriate keyboard short cut. Verify that you can modify the input and see the output update.

This exercise is mostly left to the reader. Recall that the keyboard shortcut to knit a file is `Cmd/Ctrl + Alt + I`.

26.1.1.3 Exercise 3

Compare and contrast the R notebook and R markdown files you created above. How are the outputs similar? How are they different? How are the inputs similar? How are they different? What happens if you copy the YAML header from one to the other?

R notebook files show the output inside the editor, while hiding the console. R markdown files shows the output inside the console, and does not show output inside the editor. They differ in the value of `output` in their YAML headers.

The YAML header for the R notebook will have the line,

```
---
ouptut: html_notebook
---
```

For example, this is a R notebook,

```
---
title: "Diamond sizes"
date: 2016-08-25
output: html_notebook
---

Text of the document.
```

The YAML header for the R markdown file will have the line,

```
ouptut: html_document
```

For example, this is a R markdown file.

```
---
title: "Diamond sizes"
date: 2016-08-25
output: html_document
---

Text of the document.
```

Copying the YAML header from an R notebook to a R markdown file changes it to an R markdown file, and vice-versa. More specifically, changing the value of `output` to This is because the RStudio IDE when opening and the **rmarkdown** package when knitting uses the YAML header of a file, and in particular the value of the `output` key in the YAML header, to determine what type of document it is.

26.1.1.4 Exercise 4

Create one new R Markdown document for each of the three built-in formats: HTML, PDF and Word. Knit each of the three documents. How does the output differ? How does the input differ? (You may need to install LaTeX in order to build the PDF output — RStudio will prompt you if this is necessary.)

They produce different outputs, both in the final documents and intermediate files (notably the type of plots produced). The only difference in the inputs is the value of `output` in the YAML header: `word_document` for Word documents, `pdf_document` for PDF documents, and `html_document` for HTML documents.

26.2 Text formatting with R Markdown

26.2.1 Exercise 1

Practice what you've learned by creating a brief CV. The title should be your name, and you should include headings for (at least) education or employment. Each of the sections should include a bulleted list of jobs/degrees. Highlight the year in bold.

TODO

26.2.2 Exercise 2

Using the R Markdown quick reference, figure out how to:

1. Add a footnote.
2. Add a horizontal rule.
3. Add a block quote.

```
The quick brown fox jumped over the lazy dog.[^quick-fox]

Use three or more `-` for a horizontal rule. For example,

---

The horizontal rule uses the same syntax as a YAML block? So how does R markdown
distinguish between the two?  Three dashes ("---") is only treated the start of
a YAML block if it is at the start of the document.

> This would be a block quote. Generally, block quotes are used to indicate
> quotes longer than a three or four lines.


[^quick-fox]: This is an example of a footnote. The sentence this is footnoting
  is often used for displaying fonts because it includes all 26 letters of the
  English alphabet.
```

26.2.3 Exercise 3

Copy and paste the contents of `diamond-sizes.Rmd` from https://github.com/hadley/r4ds/tree/master/rmarkdown in to a local R markdown document. Check that you can run it, then add text after the frequency polygon that describes its most striking features.

For an example R markdown document, see the exercises in the next section.

26.3 Code Chunks

Exercises 1–3 are answered in …

```
#> ---
#> title: "Exercise 24.4.7.4"
#> author: "Jeffrey Arnold"
#> date: "2/1/2018"
#> output: html_document
#> ---
#>
#> ```{r setup, include=FALSE}
#> knitr::opts_chunk$set(echo = TRUE, cache = TRUE)
#> ```
#>
#> The chunk `a` has no dependencies.
#> ```{r a}
#> print(lubridate::now())
#> x <- 1
#> ```
#>
#> The chunk `b` depends on `a`.
#> ```{r b, dependson = c("a")}
#> print(lubridate::now())
#> y <- x + 1
#> ```
```

```
#>
#> The chunk `c` depends on `a`.
#> ```{r c, dependson = c("a")}
#> print(lubridate::now())
#> z <- x * 2
#> ```
#>
#> The chunk `d` depends on `c` and `b`:
#> ```{r d, dependson = c("c", "b")}
#> print(lubridate::now())
#> w <- y + z
#> ```
#>
#> If this document is knit repeatedly, the value  printed by `lubridate::now()` will be the same for all chur
#> and the same as the first time the document was run with caching.
```

26.3.1 Exercise 1

Add a section that explores how diamond sizes vary by cut, color, and clarity. Assume you're writing a report for someone who doesn't know R, and instead of setting `echo = FALSE` on each chunk, set a global option.

26.3.2 Exercise 2

Download `diamond-sizes.Rmd` from https://github.com/hadley/r4ds/tree/master/rmarkdown. Add a section that describes the largest 20 diamonds, including a table that displays their most important attributes.

For the this, I use `arrange()` and `slice()` to select the largest twenty diamonds, and `knitr::kable()` to produce a formatted table.

26.3.3 Exercise 3

Modify `diamonds-sizes.Rmd` to use comma() to produce nicely formatted output. Also include the percentage of diamonds that are larger than 2.5 carats.

I moved the computation of the number larger and percent of diamonds larger than 2.5 carats into a code chunk. I find that it is best to keep inline R expressions simple, usually consisting of an object and a formatting function. This makes it both easier to read and test the R code, while simultaneously making the prose easier to read. It helps the readability of the code and document to keep the computation of objects used in prose close to their use. Calculating those objects in a code chunk with the `include = FALSE` option (as is done in `diamonds-size.Rmd`) is useful in this regard.

4. Set up a network of chunks where `d` depends on `c` and `b`, and both `b` and `c` depend on `a`. Have each chunk print lubridate::now(), set cache = TRUE, then verify your understanding of caching.

```
cat(readr::read_file("rmarkdown/caching.Rmd"))
#> ---
#> title: "Exercise 24.4.7.4"
#> author: "Jeffrey Arnold"
#> date: "2/1/2018"
#> output: html_document
#> ---
#>
#> ```{r setup, include=FALSE}
```

```
#> knitr::opts_chunk$set(echo = TRUE, cache = TRUE)
#> ```
#>
#> The chunk `a` has no dependencies.
#> ```{r a}
#> print(lubridate::now())
#> x <- 1
#> ```
#>
#> The chunk `b` depends on `a`.
#> ```{r b, dependson = c("a")}
#> print(lubridate::now())
#> y <- x + 1
#> ```
#>
#> The chunk `c` depends on `a`.
#> ```{r c, dependson = c("a")}
#> print(lubridate::now())
#> z <- x * 2
#> ```
#>
#> The chunk `d` depends on `c` and `b`:
#> ```{r d, dependson = c("c", "b")}
#> print(lubridate::now())
#> w <- y + z
#> ```
#>
#> If this document is knit repeatedly, the value  printed by `lubridate::now()` will be the same for a
#> and the same as the first time the document was run with caching.
```

Chapter 27

Graphics for communication

27.1 Introduction

```
library("tidyverse")
#> -- Attaching packages ------------------------------------------------------ tidyverse 1.2.1 --
#> ✓ ggplot2 2.2.1     ✓ purrr   0.2.4
#> ✓ tibble  1.4.2     ✓ dplyr   0.7.4
#> ✓ tidyr   0.8.0     ✓ stringr 1.3.0
#> ✓ readr   1.1.1     ✓ forcats 0.3.0
#> -- Conflicts --------------------------------------------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library("modelr")
library("lubridate")
#> Loading required package: methods
#>
#> Attaching package: 'lubridate'
#> The following object is masked from 'package:base':
#>
#>     date
```

27.2 Label

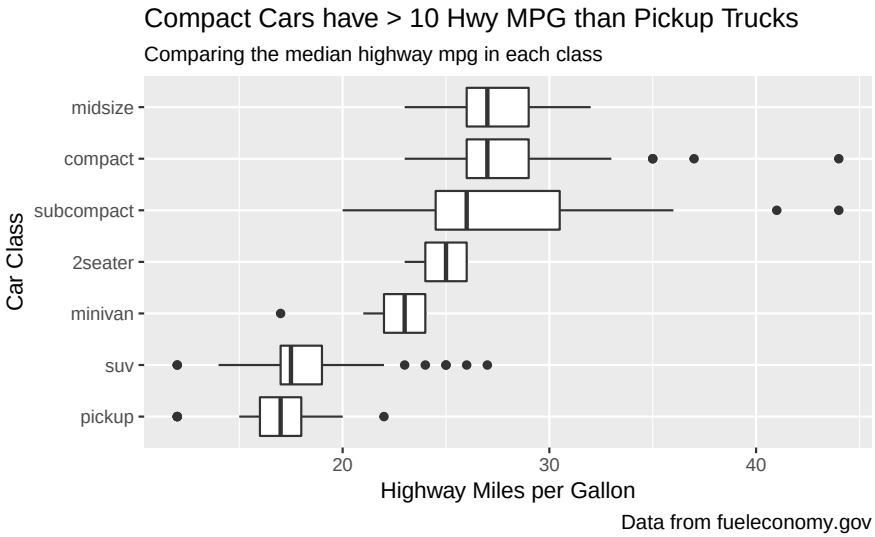
27.2.1 Exercises

27.2.1.1 Exercise 1

Create one plot on the fuel economy data with customized `title`, `subtitle`, `caption`, `x`, `y`, and `colour` labels.

```
ggplot(data = mpg,
       mapping = aes(x = reorder(class, hwy, median), y = hwy)) +
  geom_boxplot() +
  coord_flip() +
  labs(
    title = "Compact Cars have > 10 Hwy MPG than Pickup Trucks",
    subtitle = "Comparing the median highway mpg in each class",
```

```
    caption = "Data from fueleconomy.gov",
    x = "Car Class",
    y = "Highway Miles per Gallon"
  )
```

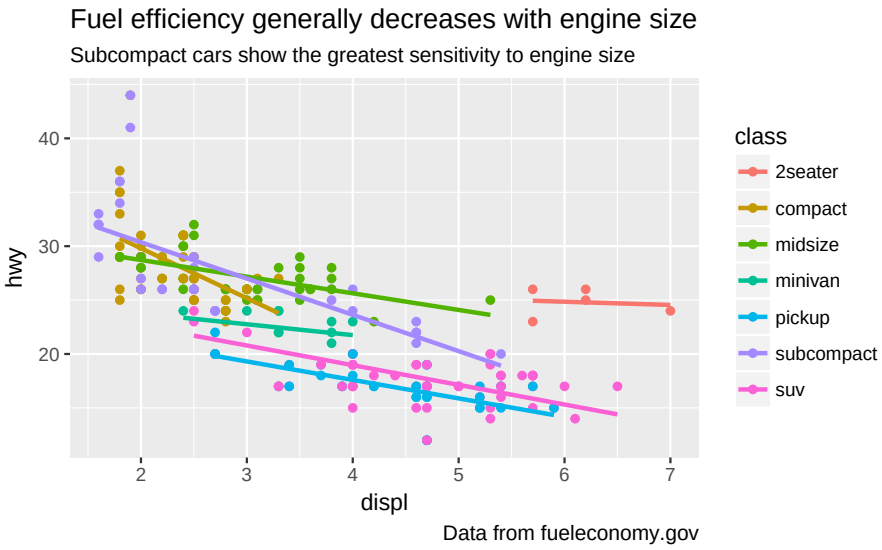


27.2.2 Exercise 3

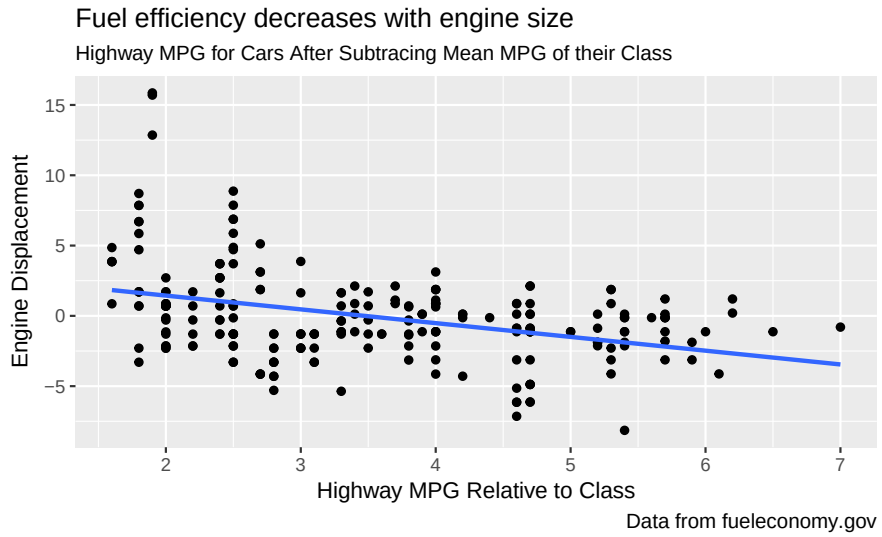
The `geom_smooth()` is somewhat misleading because the `hwy` for large engines is skewed upwards due to the inclusion of lightweight sports cars with big engines. Use your modeling tools to fit and display

a better model.

```
ggplot(mpg, aes(displ, hwy, colour = class)) +
  geom_point(aes(colour = class)) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(
    title = "Fuel efficiency generally decreases with engine size",
    subtitle = "Subcompact cars show the greatest sensitivity to engine size",
    caption = "Data from fueleconomy.gov"
  )
```




```
mod <- lm(hwy ~ class, data = mpg)
mpg %>%
  add_residuals(mod) %>%
  ggplot(aes(displ, resid)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(
    title = "Fuel efficiency decreases with engine size",
    subtitle = "Highway MPG for Cars After Subtracing Mean MPG of their Class",
    caption = "Data from fueleconomy.gov",
    x = "Highway MPG Relative to Class",
    y = "Engine Displacement"
  )
```



27.2.2.1 Exercise 3

Take an exploratory graphic that you've created in the last month, and add informative titles to make it easier for others to understand.

This exercise is by is intrinsically left to readers.

27.3 Annotations

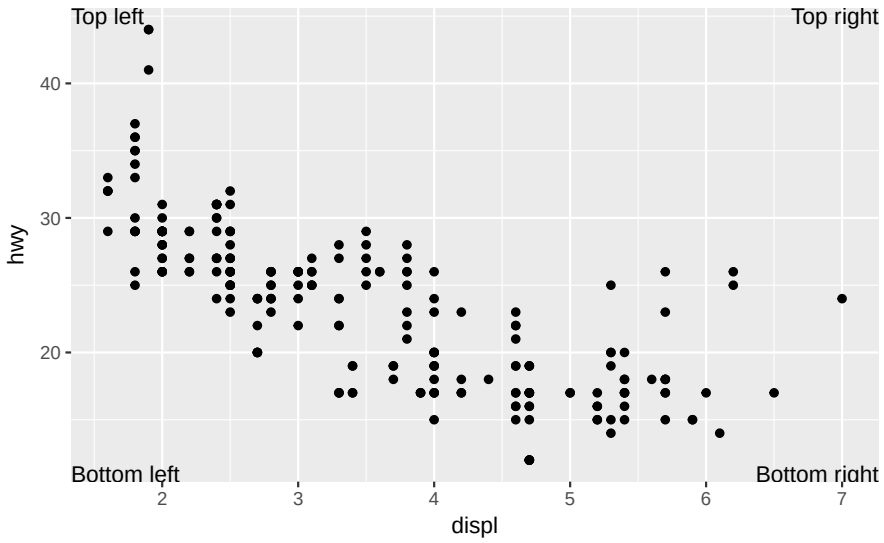
27.3.1 Exercises

27.3.1.1 Exercise 1

Use `geom_text()` with infinite positions to place text at the four corners of the plot.

I can use similar code as the example in the text. However, I need to use `vjust` and `hjust` in order for the text to appear in the plot, and these need to be different for each corner. But, `geom_text` takes `hjust` and `vjust` as aesthetics, I can add them to the data and mappings, and use a single `geom_text` call instead of four different `geom_text` calls with four different data arguments, and four different values of `hjust` and `vjust` arguments.

```
label <- tribble(
  ~displ, ~hwy, ~label, ~vjust, ~hjust,
     Inf,  Inf,    "Top right", "top", "right",
     Inf, -Inf,    "Bottom right", "bottom", "right",
    -Inf,  Inf,    "Top left", "top", "left",
    -Inf, -Inf,    "Bottom left", "bottom", "left"
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(aes(label = label, vjust = vjust, hjust = hjust), data = label)
```

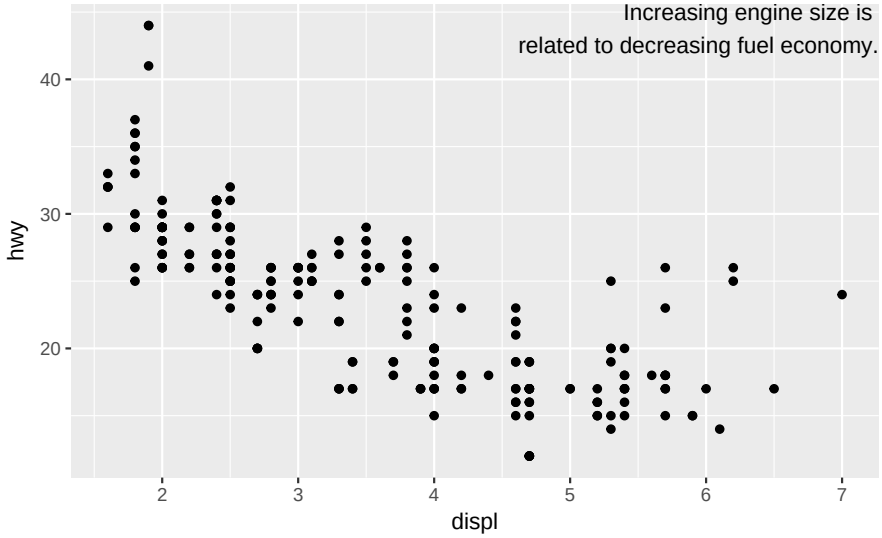


27.3.1.2 Exercise 2

Read the documentation for `annotate()`. How can you use it to add a text label to a plot without having to create a tibble?

With annotate you use what would be aesthetic mappings directly as arguments:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  annotate("text", x = Inf, y = Inf,
           label = "Increasing engine size is \nrelated to decreasing fuel economy.", vjust = "top", hj
```

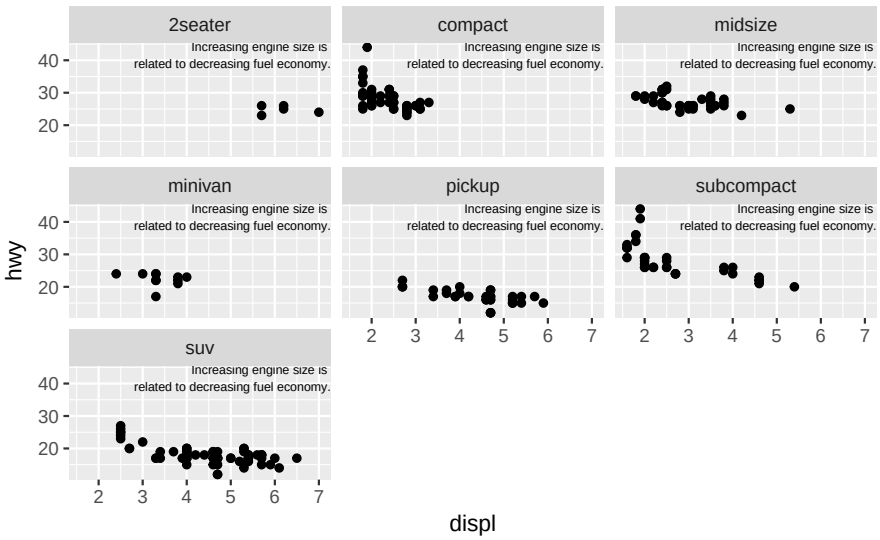


27.3.1.3 Exercise 3

How do labels with `geom_text()` interact with faceting? How can you add a label to a single facet? How can you put a different label in each facet? (Hint: think about the underlying data.)

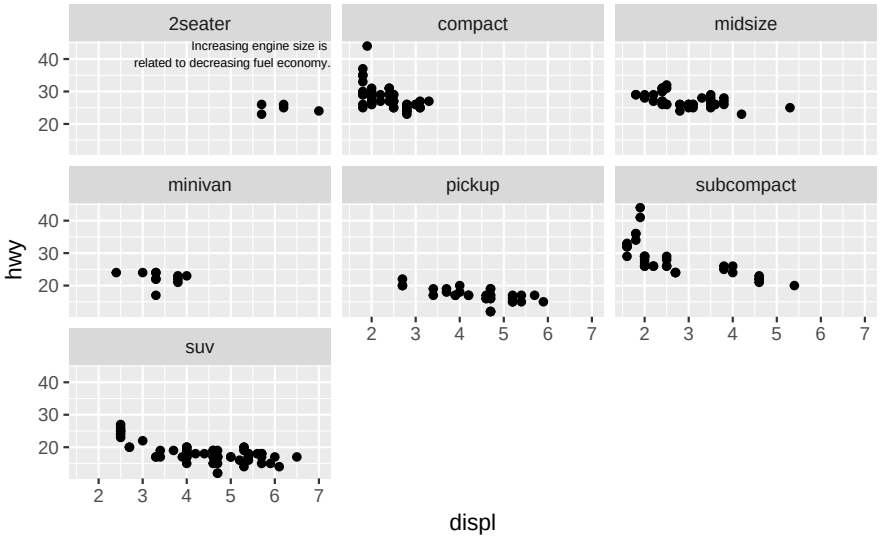
If the facet variable is not specified, the text is drawn in all facets.

```
label <- tibble(
  displ = Inf,
  hwy = Inf,
  label = "Increasing engine size is \nrelated to decreasing fuel economy."
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(aes(label = label), data = label, vjust = "top", hjust = "right",
            size = 2) +
  facet_wrap(~ class)
```



To draw the label in only one facet, add a column to the label data frame with the value of the faceting variable(s) in which to draw it.

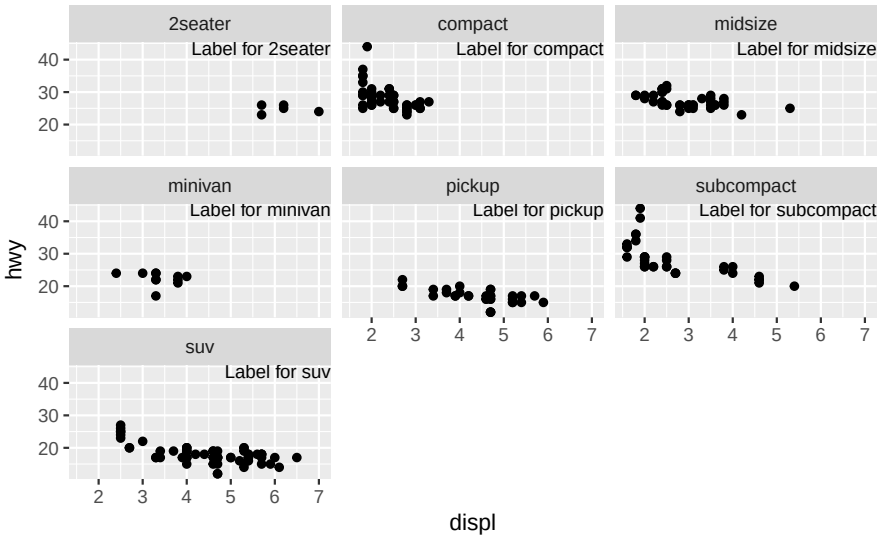
```
label <- tibble(
  displ = Inf,
  hwy = Inf,
  class = "2seater",
  label = "Increasing engine size is \nrelated to decreasing fuel economy."
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(aes(label = label), data = label, vjust = "top", hjust = "right",
            size = 2) +
  facet_wrap(~ class)
```



To draw labels in different plots, simply have the facetting variable(s):

```
label <- tibble(
  displ = Inf,
  hwy = Inf,
```

```
  class = unique(mpg$class),
  label = stringr::str_c("Label for ", class)
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(aes(label = label), data = label, vjust = "top", hjust = "right",
            size = 3) +
  facet_wrap(~ class)
```



27.3.1.4 Exercise 4

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

- `label.padding`: padding around label
- `label.r`: amount of rounding in the corners
- `label.size`: size of label border

27.3.1.5 Exercise 5

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

The four arguments are: (from the help for arrow) - `angle` : angle of arrow head - `length` : length of the arrow head - `ends`: ends of the line to draw arrow head - `type`: `"open"` or `"close"`: whether the arrow head is a closed or open triangle

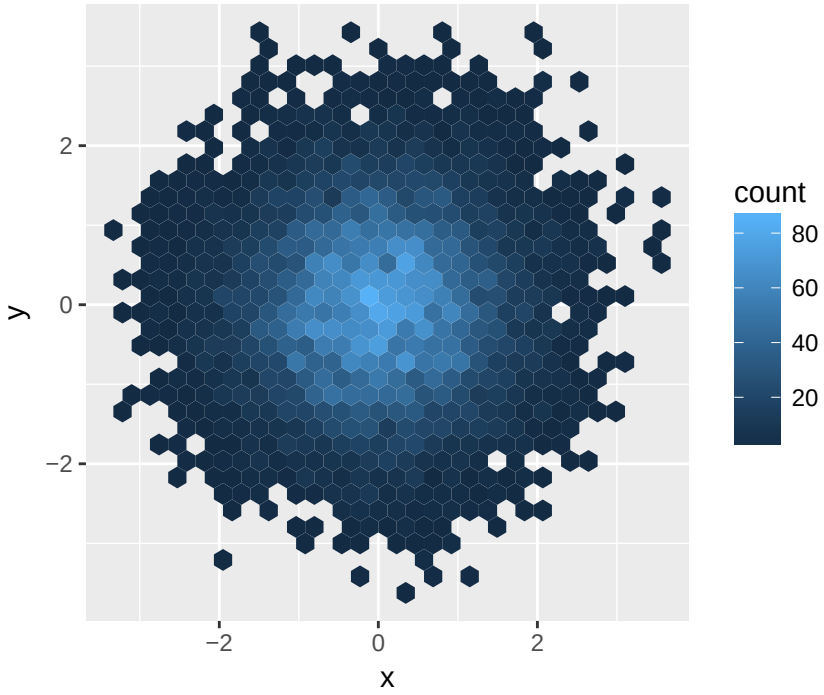
27.4 Scales

27.4.1 Exercises

27.4.1.1 Exercise 1

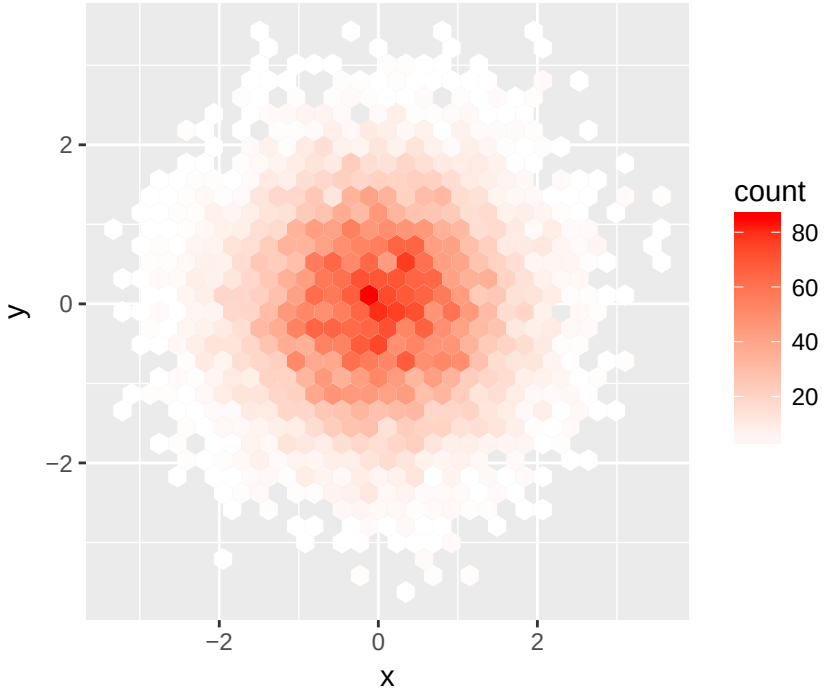
Why doesn't the following code override the default scale?

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
)
ggplot(df, aes(x, y)) +
  geom_hex() +
  scale_colour_gradient(low = "white", high = "red") +
  coord_fixed()
```



It does not override the default scale because the colors in `geom_hex` are set by the `fill` aesthetic, not the `color` aesthetic.

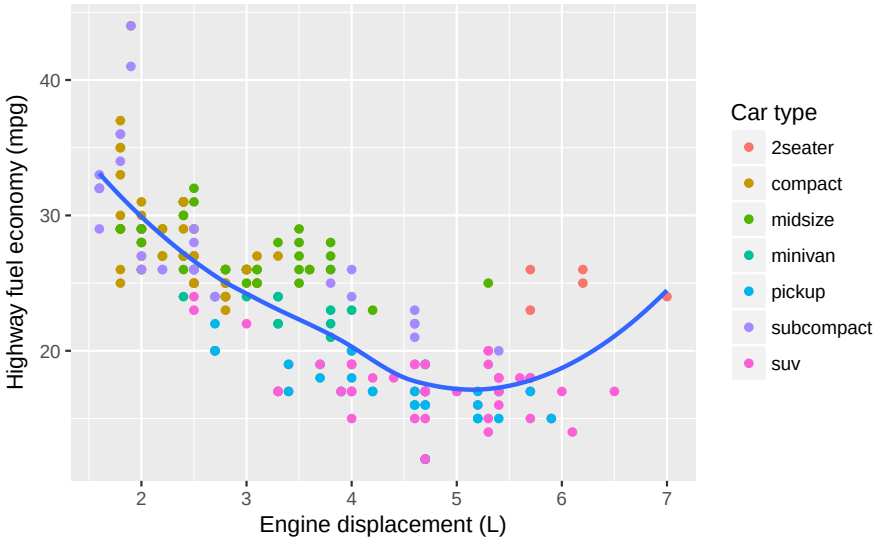
```
ggplot(df, aes(x, y)) +
  geom_hex() +
  scale_fill_gradient(low = "white", high = "red") +
  coord_fixed()
```



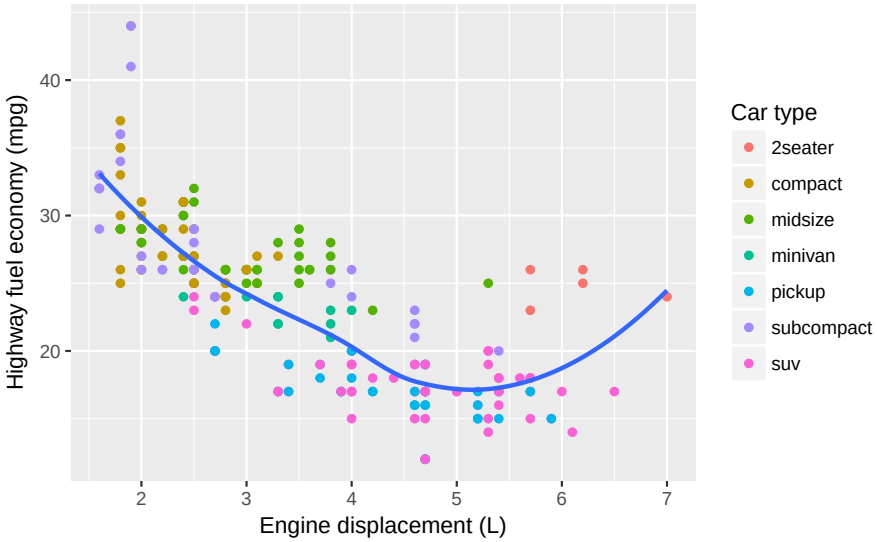
27.4.1.2 Exercise 2

The first argument to every scale is the label for the scale. It is equivalent to using the `labs` function.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Engine displacement (L)",
    y = "Highway fuel economy (mpg)",
    colour = "Car type"
  )
#> `geom_smooth()` using method = 'loess'
```



```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_smooth(se = FALSE) +
  scale_x_continuous("Engine displacement (L)") +
  scale_y_continuous("Highway fuel economy (mpg)") +
  scale_colour_discrete("Car type")
#> `geom_smooth()` using method = 'loess'
```

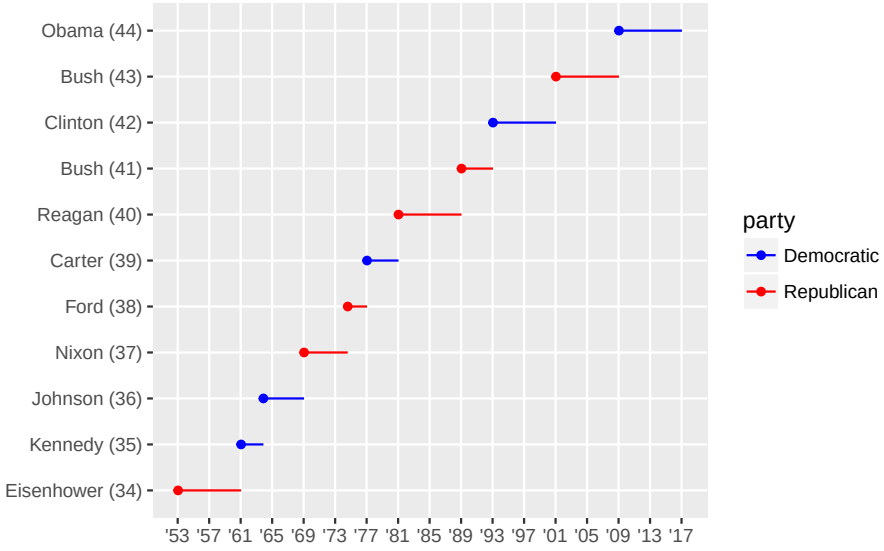


27.4.1.3 Exercise 3

Change the display of the presidential terms by:

1. Combining the two variants shown above.
2. Improving the display of the y axis.
3. Labelling each term with the name of the president.
4. Adding informative plot labels.
5. Placing breaks every 4 years (this is trickier than it seems!).

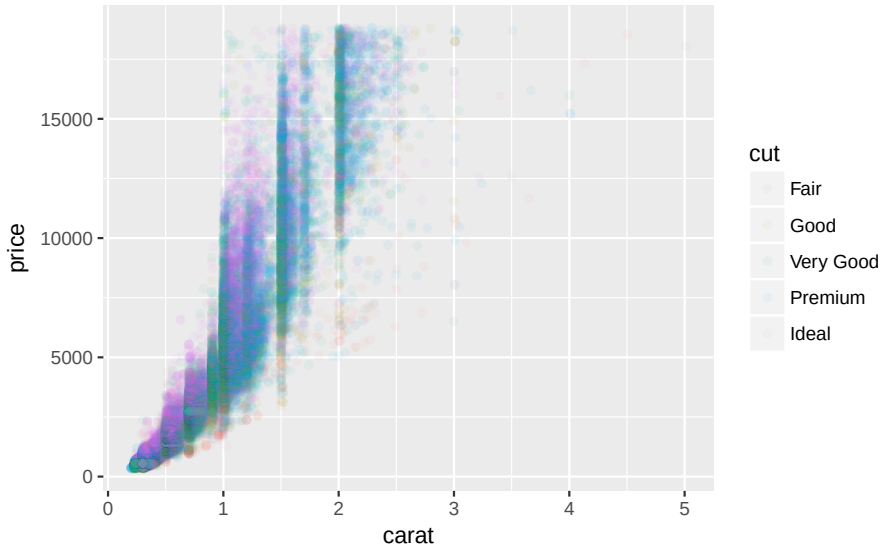
```
years <- lubridate::make_date(seq(year(min(presidential$start)),
              year(max(presidential$end)),
              by = 4), 1, 1)

presidential %>%
  mutate(id = 33 + row_number(),
         name_id = stringr::str_c(name, " (", id, ")"),
         name_id = factor(name_id, levels = name_id)) %>%
  ggplot(aes(start, name_id, colour = party)) +
    geom_point() +
    geom_segment(aes(xend = end, yend = name_id)) +
    scale_colour_manual(values = c(Republican = "red", Democratic = "blue")) +
    scale_y_discrete(NULL) +
    scale_x_date(NULL, breaks = years, date_labels = "'%y") +
    theme(panel.grid.minor = element_blank())
```



27.4.1.4 Exercise 4

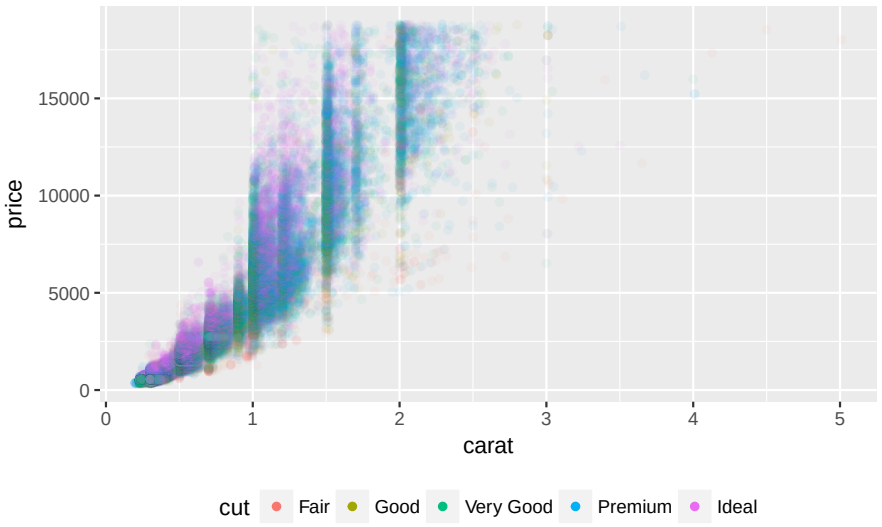
Use `override.aes` to make the legend on the following plot easier to see.

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



The problem with the legend is that the `alpha` value make the colors hard to see. So I'll override the alpha value to make the points solid in the legend.

```
ggplot(diamonds, aes(carat, price)) +
  geom_point(aes(colour = cut), alpha = 1/20)  +
  theme(legend.position = "bottom") +
  guides(colour = guide_legend(nrow = 1, override.aes = list(alpha = 1)))
```


price
15000
10000
5000
0
0
1
2
3
4
5
carat
cut
Fair
Good
Very Good
Premium
Ideal

Chapter 28

R Markdown Formats

No exercises.

This document was built with **bookdown**. You can see the source at https://github.com/jrnold/r4ds-exercise-solutions.

Chapter 29

R Markdown Workflow

No exercises