

R for Data Science Solutions

Jeffrey B. Arnold

September 08, 2018

Contents

| | |
|---|-----------|
| Welcome | 7 |
| 1 Introduction | 9 |
| Acknowledgments | 9 |
| Organization | 9 |
| Dependencies | 9 |
| Bugs/Contributing | 9 |
| Colophon | 10 |
| I Explore | 17 |
| 2 Introduction | 19 |
| 3 Data Visualisation | 21 |
| 3.1 Introduction | 21 |
| 3.2 First Steps | 21 |
| 3.3 Aesthetic mappings | 24 |
| 3.4 Common problems | 28 |
| 3.5 Facets | 28 |
| 3.6 Geometric Objects | 32 |
| 3.7 Statistical Transformations | 41 |
| 3.8 Position Adjustments | 45 |
| 3.9 Coordinate Systems | 52 |
| 3.10 The Layered Grammar of Graphics | 56 |
| 4 Workflow: basics | 57 |
| Prerequisites | 57 |
| 4.1 Coding basics | 57 |
| 4.2 What's in a name? | 57 |
| 4.3 Calling functions | 57 |
| 4.4 Practice | 57 |
| 5 Data transformation | 61 |
| 5.1 Introduction | 61 |
| 5.2 Filter rows with <code>filter()</code> | 61 |
| 5.3 Arrange rows with <code>arrange()</code> | 67 |
| 5.4 Select columns with <code>select()</code> | 71 |
| 5.5 Add new variables with <code>mutate()</code> | 75 |
| 5.6 Grouped summaries with <code>summarise()</code> | 82 |
| 5.7 Grouped mutates (and filters) | 86 |

| | |
|--|------------|
| 6 Workflow: scripts | 93 |
| 6.1 Running code | 93 |
| 6.2 RStudio diagnostics | 93 |
| 6.3 Practice | 93 |
| 7 Exploratory Data Analysis | 95 |
| 7.1 Introduction | 95 |
| 7.2 Questions | 95 |
| 7.3 Variation | 95 |
| 7.4 Missing Values | 101 |
| 7.5 Covariation | 102 |
| 7.6 Patterns and models | 121 |
| 7.7 ggplot2 calls | 121 |
| 7.8 Learning more | 121 |
| 8 Workflow: projects | 123 |
| II Wrangle | 125 |
| 9 Introduction | 127 |
| 10 Tibbles | 129 |
| 10.1 Introduction | 129 |
| 10.2 Creating Tibbles | 129 |
| 10.3 Tibbles vs. data.frame | 129 |
| 10.4 Subsetting | 129 |
| 10.5 Interacting with older code | 129 |
| 10.6 Exercises | 129 |
| 11 Data import | 135 |
| 11.1 Introduction | 135 |
| 11.2 Getting started | 135 |
| 11.3 Parsing a vector | 137 |
| 11.4 Parsing a file | 141 |
| 11.5 Writing to a file | 141 |
| 11.6 Other Types of Data | 141 |
| 12 Tidy Data | 143 |
| 12.1 Introduction | 143 |
| 12.2 Tidy data | 143 |
| 12.3 Spreading and Gathering | 146 |
| 12.4 Separating and Uniting | 150 |
| 12.5 Missing Values | 153 |
| 12.6 Case Study | 154 |
| 12.7 Non-Tidy Data | 157 |
| 13 Relational data | 159 |
| 13.1 Introduction | 159 |
| 13.2 Keys | 160 |
| 13.3 Mutating Joins | 163 |
| 13.4 Filtering Joins | 168 |
| 13.5 Join problems | 171 |
| 13.6 Set operations | 171 |

| | |
|--|------------|
| 14 Strings | 173 |
| 14.1 Introduction | 173 |
| 14.2 String Basics | 173 |
| 14.3 Matching Patterns and Regular Expressions | 175 |
| 14.4 Tools | 181 |
| 14.5 Other types of patterns | 186 |
| 14.6 Other uses of regular expressions | 186 |
| 14.7 stringi | 186 |
| 15 Factors | 189 |
| 15.1 Introduction | 189 |
| 15.2 Creating Factors | 189 |
| 15.3 General Social Survey | 189 |
| 15.4 Modifying factor order | 193 |
| 15.5 Modifying factor levels | 196 |
| 16 Dates and times | 199 |
| 16.1 Introduction | 199 |
| 16.2 Creating date/times | 199 |
| 16.3 Date-Time Components | 200 |
| 16.4 Time Spans | 206 |
| 16.5 Time Zones | 207 |
| III Program | 209 |
| 17 Introduction | 211 |
| 18 Pipes | 213 |
| 19 Functions | 215 |
| 19.1 Introduction | 215 |
| 19.2 When should you write a function? | 215 |
| 19.3 Functions are for humans and computers | 219 |
| 19.4 Conditional execution | 221 |
| 19.5 Function arguments | 224 |
| 19.6 Return values | 226 |
| 19.7 Environment | 226 |
| 20 Vectors | 227 |
| 20.1 Introduction | 227 |
| 20.2 Vector Basics | 227 |
| 20.3 Important Types of Atomic Vector | 227 |
| 20.4 Using atomic vectors | 231 |
| 20.5 Recursive Vectors (lists) | 237 |
| 20.6 Attributes | 238 |
| 20.7 Augmented Vectors | 238 |
| 21 Iteration | 241 |
| 21.1 Introduction | 241 |
| 21.2 For Loops | 241 |
| 21.3 For loop variations | 249 |
| 21.4 For loops vs. functionals | 252 |
| 21.5 The map functions | 253 |
| 21.6 Dealing with Failure | 256 |

| | |
|---|------------|
| 21.7 Mapping over multiple arguments | 256 |
| 21.8 Walk | 256 |
| 21.9 Other patterns of for loops | 257 |
| IV Model | 259 |
| 22 Introduction | 261 |
| 23 Model basics | 263 |
| 23.1 Prerequisites | 263 |
| 23.2 A simple model | 263 |
| 23.3 Visualizing Models | 267 |
| 23.4 Formulas and Model Families | 272 |
| 23.5 Missing values | 276 |
| 23.6 Other model families | 276 |
| 24 Model building | 277 |
| 24.1 Introduction | 277 |
| 24.2 Why are low quality diamonds more expensive? | 277 |
| 24.3 What affects the number of daily flights? | 278 |
| 24.4 Learning more about models | 284 |
| 25 Many models | 285 |
| 25.1 Introduction | 285 |
| 25.2 Gapminder | 285 |
| 25.3 List-columns | 288 |
| 25.4 Creating list-columns | 288 |
| 25.5 Simplifying list-columns | 289 |
| V Communicate | 291 |
| 26 Introduction | 293 |
| 27 R Markdown | 295 |
| 27.1 Introduction | 295 |
| 27.2 R Markdown Basics | 295 |
| 27.3 Text formatting with R Markdown | 296 |
| 27.4 Code Chunks | 299 |
| 27.5 YAML header | 301 |
| 27.6 Learning more | 301 |
| 28 Graphics for communication | 303 |
| 28.1 Introduction | 303 |
| 28.2 Label | 303 |
| 28.3 Annotations | 305 |
| 28.4 Scales | 309 |
| 28.5 Zooming | 313 |
| 28.6 Themes | 313 |
| 28.7 Saving your plots | 314 |
| 28.8 Learning more | 314 |
| 29 R Markdown formats | 315 |
| 30 R Markdown workflow | 317 |

Welcome

This contains solutions to the exercise in *R for Data Science*, byn Hadley Wickham and Garret Grolemund (Wickham and Grolemund 2017). The website for that book is r4ds.had.co.nz, and a physical copy is published by O'Reilly and available from amazon.

This work is licensed under a Creative Commons Attribution 4.0 International License

Chapter 1

Introduction

Acknowledgments

All the credit should go to Garrett Grolemund and Hadley Wickham for writing the truly fantastic *R for Data Science* book, without which these solutions would not exist—literally.

Special thanks to @dongzhuoer for a careful reading of the book and noticing numerous issues and proposing fixes.

These solutions have benefited from those who fixed problems and contributed solutions. Thank you to all of those who contributed on GitHub (in alphabetical order): (in alphabetical order) @adamblake, @benherbertson, @carajoos, @dcgreaves, @decoursin, @dongzhuoer, @JamesCuster, @jmclawson, @KleinGeard, @mjones01, @nickcorona, @nzwang, and @tinhb92.

Organization

The solutions are organized in the same order, and with the same numbers as in *R for Data Science*. Sections without exercises are given a placeholder.

Like *R for Data Science*, packages used in each chapter are loaded in a code chunk at the start of the chapter in a section titled “Prerequisites”. If a package is used infrequently in solutions it may not be loaded, and functions using it will be called using the package name followed by two colons, as in `dplyr::mutate()` (see the *R for Data Science* Introduction). We will also use `::` to be explicit about the package of a function.

Dependencies

You can install all packages used in the solutions with the following line of code.

```
devtools::install_github("jrnold/r4ds-exercise-solutions")
```

Bugs/Contributing

If you find any typos, errors in the solutions, have an alternative solution, or think the solution could be improved, I would love your contributions. Please open an issue at <https://github.com/jrnold/r4ds-exercise-solutions/issues> or a pull request at <https://github.com/jrnold/r4ds-exercise-solutions/pulls>.

Colophon

HTML and PDF versions of this book are available at <http://jrnold.github.io/r4ds-exercise-solutions>. The book is powered by bookdown which makes it easy to turn R markdown files into HTML, PDF, and EPUB.

The source of this book is available at <https://github.com/jrnold/r4ds-exercise-solutions> This book was built from commit b457d90.

This book was built with:

```
devtools::session_info("R4DSSolutions")
#> Session info -----
#>   setting  value
#>   version  R version 3.5.1 (2018-07-02)
#>   system    x86_64, darwin15.6.0
#>   ui        X11
#>   language (EN)
#>   collate   en_US.UTF-8
#>   tz        America/Los_Angeles
#>   date      2018-09-07

#> Packages -----
#>   package     * version  date
#>   assertthat    0.2.0    2017-04-11
#>   babynames     0.3.0    2017-04-14
#>   backports      1.1.2    2017-12-13
#>   base64enc     0.1-3    2015-07-28
#>   BH            1.66.0-1  2018-02-13
#>   bindr          0.1.1    2018-03-13
#>   bindrcpp      * 0.2.2    2018-03-29
#>   bookdown       0.7.17   2018-08-10
#>   brew           1.0-6    2011-04-13
#>   broom          0.5.0    2018-07-17
#>   callr          3.0.0    2018-08-24
#>   cellranger     1.1.0    2016-07-27
#>   cli             1.0.0    2017-11-05
#>   clipr           0.4.1    2018-06-23
#>   codetools       0.2-15   2016-10-05
#>   colorspace      1.3-2    2016-12-14
#>   condvis         0.4-2    2017-10-11
#>   crayon          1.3.4    2017-09-16
#>   crosstalk       1.0.0    2016-12-21
#>   curl             3.2     2018-03-28
#>   datamodelr     0.2.2.9002 2018-05-27
#>   DBI             1.0.0    2018-05-02
#>   dbplyr          1.2.2    2018-07-25
#>   DiagrammeR      1.0.0    2018-03-01
#>   digest          0.6.16   2018-08-22
#>   downloader       0.4     2015-07-09
#>   dplyr          * 0.7.6    2018-06-29
#>   evaluate        0.11    2018-07-17
#>   fansi            0.3.0    2018-08-13
#>   forcats          0.3.0    2018-02-19
#>   gapminder       0.3.0    2017-10-31
#>   ggplot2         3.0.0.9000 2018-08-24
#>   ggrepel          0.8.0.9000 2018-08-10
```

```

#> glue           1.3.0    2018-08-24
#> graphics       * 3.5.1    2018-07-05
#> grDevices      * 3.5.1    2018-07-05
#> grid            3.5.1    2018-07-05
#> gridExtra        2.3     2017-09-09
#> gtable          0.2.0    2016-02-26
#> haven            1.1.2    2018-06-27
#> hexbin          1.27.2   2018-01-15
#> highr             0.7     2018-06-09
#> hms              0.4.2    2018-03-10
#> htmldeps         0.1.1    2018-08-10
#> htmltools         0.3.6    2017-04-28
#> htmlwidgets       1.2     2018-04-19
#> httpuv            1.4.5    2018-07-19
#> httr              1.3.1    2017-08-20
#> igraph            1.2.2    2018-07-27
#> influenceR        0.1.0    2015-09-03
#> jpeg              0.1-8    2014-01-23
#> jsonlite          1.5     2017-06-01
#> knitr             1.20    2018-02-20
#> labeling           0.3     2014-08-23
#> Lahman            6.0-0    2017-08-15
#> later              0.7.3    2018-06-08
#> lattice            0.20-35  2017-03-25
#> lazyeval           0.2.1    2017-10-29
#> leaflet            2.0.1    2018-06-04
#> lubridate          1.7.4    2018-04-11
#> magrittr           * 1.5     2014-11-22
#> maps              3.3.0    2018-04-03
#> markdown            0.8     2017-04-20
#> MASS                7.3-50   2018-04-30
#> Matrix              1.2-14   2018-04-13
#> methods            * 3.5.1    2018-07-05
#> mgcv                1.8-24   2018-06-23
#> microbenchmark     1.4-4     2018-01-24
#> mime                 0.5     2016-07-07
#> modelr              0.1.2    2018-05-11
#> munsell             0.5.0    2018-06-12
#> nlme                3.1-137   2018-04-07
#> nycflights13        1.0.0    2018-06-26
#> openssl              1.0.2    2018-07-30
#> pillar               1.3.0    2018-07-14
#> pkgconfig            2.0.2    2018-08-16
#> plogr                0.2.0    2018-03-25
#> plyr                 1.8.4    2016-06-08
#> png                  0.1-7    2013-12-03
#> processx             3.2.0    2018-08-16
#> promises             1.0.1    2018-04-13
#> pryr                 0.1.4    2018-02-18
#> ps                   1.1.0    2018-08-10
#> purrrr              0.2.5    2018-05-29
#> r4ds                 0.1     2018-08-10
#> R4DSSolutions        0.1     2018-08-10

```

```

#> R6                      2.2.2    2017-06-17
#> raster                   2.6-7    2017-11-13
#> RColorBrewer              1.1-2    2014-12-07
#> Rcpp                     0.12.18   2018-07-23
#> readr                     1.1.1    2017-05-16
#> readxl                   1.1.0    2018-04-20
#> rematch                  1.0.1    2016-04-21
#> reprex                   0.2.0    2018-06-22
#> reshape2                 1.4.3    2017-12-11
#> rgexf                    0.15.3    2015-03-24
#> rlang                     0.2.2    2018-08-16
#> rmarkdown                1.10.12   2018-08-24
#> Rook                     1.1-1    2014-10-20
#> rstudioapi                0.7     2017-09-07
#> rvest                     0.3.2    2016-06-17
#> scales                   1.0.0    2018-08-09
#> selectr                   0.4-1    2018-04-06
#> shiny                     1.1.0    2018-05-17
#> sourcetools               0.1.7    2018-04-25
#> sp                        1.3-1    2018-06-05
#> stats                     * 3.5.1    2018-07-05
#> stringi                   1.2.4    2018-07-20
#> stringr                   1.3.1    2018-05-10
#> tibble                    1.4.2    2018-01-22
#> tidyverse                  1.2.1    2017-11-14
#> tinytex                   0.7     2018-08-22
#> tools                     3.5.1    2018-07-05
#> utf8                      1.1.4    2018-05-24
#> utils                     * 3.5.1    2018-07-05
#> viridis                   0.5.1    2018-03-29
#> viridisLite                0.3.0    2018-02-01
#> visNetwork                 2.0.4    2018-06-14
#> whisker                   0.3-2    2013-04-28
#> withr                     2.1.2    2018-03-15
#> xfun                      0.3     2018-07-06
#> XML                       3.98-1.16   2018-08-19
#> xml2                      1.2.0    2018-01-24
#> xtable                     1.8-2    2016-02-05
#> yaml                      2.2.0    2018-07-25
#> source
#> CRAN (R 3.5.0)
#> Github (rstudio/bookdown@002b0fd1)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.1)

```

```
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.1)
#> CRAN (R 3.5.0)
#> cran (@0.4-2)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> Github (bergant/datamodelr@68ea364)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.1)
#> cran (@0.11)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> cran (@0.3.0)
#> Github (tidyverse/ggplot2@adce4e2)
#> Github (slowkow/ggrepel@200571c)
#> Github (tidyverse/glue@4e74901)
#> local
#> local
#> local
#> CRAN (R 3.5.0)
#> Github (rstudio/htmldeps@c1023e0)
#> CRAN (R 3.5.0)
#> cran (@0.1-8)
#> CRAN (R 3.5.0)
#> cran (@1.20)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.1)
#> CRAN (R 3.5.0)
#> cran (@2.0.1)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
```

```
#> CRAN (R 3.5.1)
#> local
#> CRAN (R 3.5.1)
#> cran (@1.4-4)
#> CRAN (R 3.5.0)
#> cran (@0.1.2)
#> cran (@0.5.0)
#> CRAN (R 3.5.1)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.1)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> cran (@0.1-7)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> cran (@0.2.5)
#> Github (hadley/r4ds@03eb8d0)
#> local (jrnold/r4ds-exercise-solutions@NA)
#> CRAN (R 3.5.0)
#> cran (@2.6-7)
#> CRAN (R 3.5.0)
#> cran (@0.12.18)
#> CRAN (R 3.5.0)
#> cran (@0.2.2)
#> Github (rstudio/rmarkdown@6e56ad8)
#> CRAN (R 3.5.0)
#> local
#> CRAN (R 3.5.0)
#> cran (@1.3.1)
#> CRAN (R 3.5.0)
#> cran (@0.8.1)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> local
#> CRAN (R 3.5.0)
#> local
```

```
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.1)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> CRAN (R 3.5.0)
#> cran (02.2.0)
```


Part I

Explore

Chapter 2

Introduction

No exercises.

Chapter 3

Data Visualisation

3.1 Introduction

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

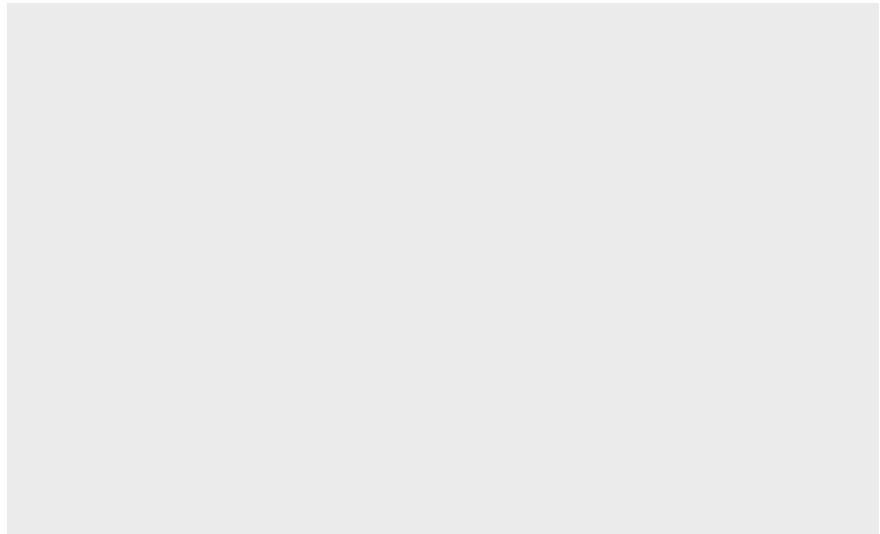
No exercises.

3.2 First Steps

Exercise 3.2.1.

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.

Exercise 3.2.2.

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, 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, 1, 1, 0, 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, 1, 1, ...
#> $ gear <dbl> 4, 4, 4, 3, 3, 3, 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, ...
```

Exercise 3.2.3.

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

The `drv` categorizes cars by which wheels the engine provides torque to, or drives: the front two wheels, the rear two wheels, or all four wheels.¹

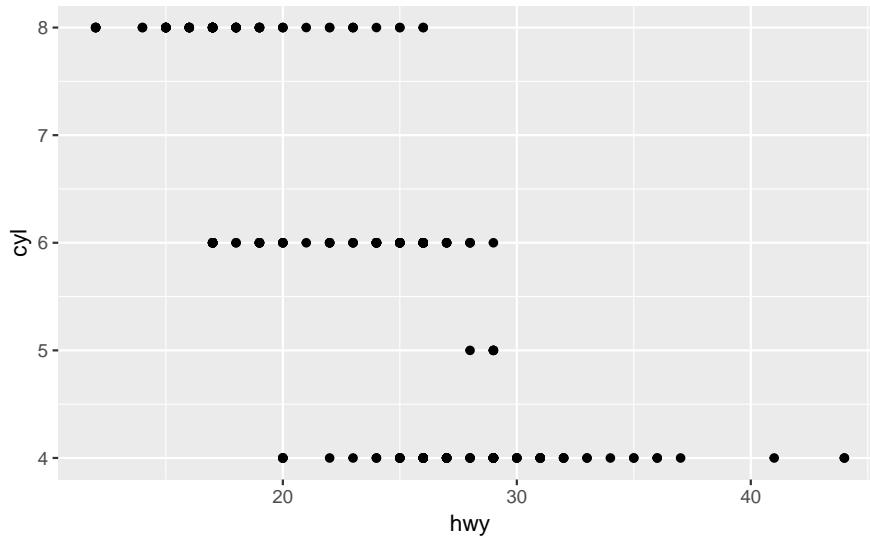
| Value | Description |
|-------|-------------------|
| "f" | front-wheel drive |
| "r" | rear-wheel drive |
| "4" | four-wheel drive |

Exercise 3.2.4.

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

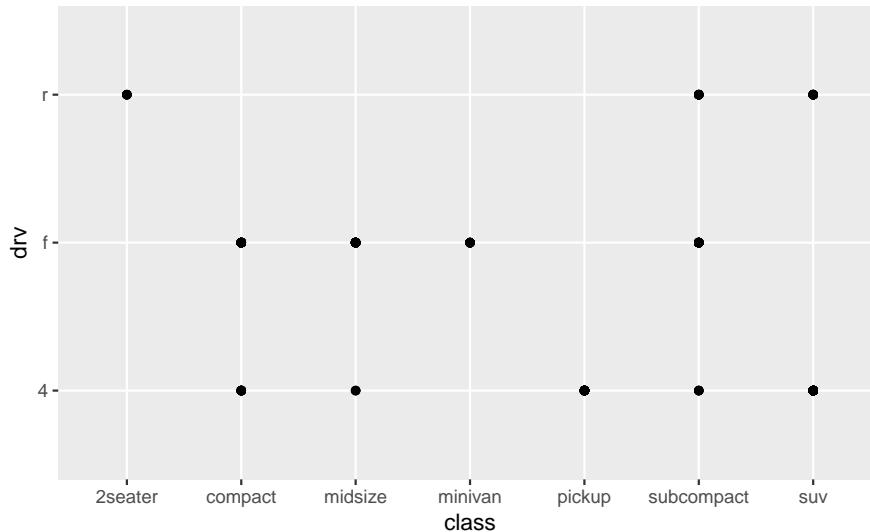
```
ggplot(mpg, aes(x = hwy, y = cyl)) +
  geom_point()
```

¹See the Wikipedia article on Automobile layout.

**Exercise 3.2.5.**

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
```

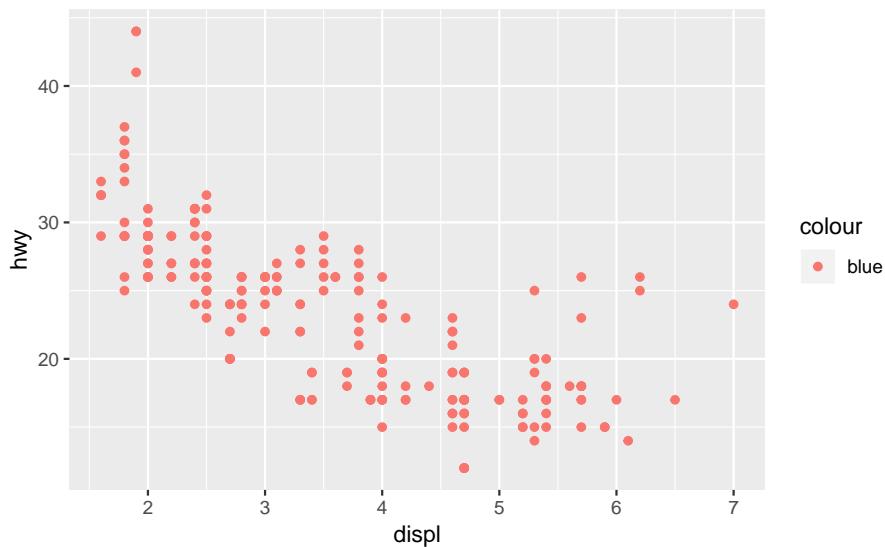
The scatter plot cannot show which are overlapping or not. Later chapters discuss means to deal with this, including alternative plots and jittering the points so they don't overlap.

3.3 Aesthetic mappings

Exercise 3.3.1

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

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



Since `colour = "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 `colour = 1:234` or `colour = 1` would be interpreted by `aes()`.

Exercise 3.3.2

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      1.8  1999     4 auto~ f       18    29 p     comp~
#> 2 audi         a4      1.8  1999     4 manu~ f       21    29 p     comp~
```

```
#> 3 audi      a4      2   2008     4 manu~ f      20   31 p   comp~
#> 4 audi      a4      2   2008     4 auto~ f     21   30 p   comp~
#> 5 audi      a4      2.8 1999     6 auto~ f     16   26 p   comp~
#> 6 audi      a4      2.8 1999     6 manu~ f     18   26 p   comp~
#> # ... with 228 more rows
```

Alternatively, `glimpse()` displays the type of each column:

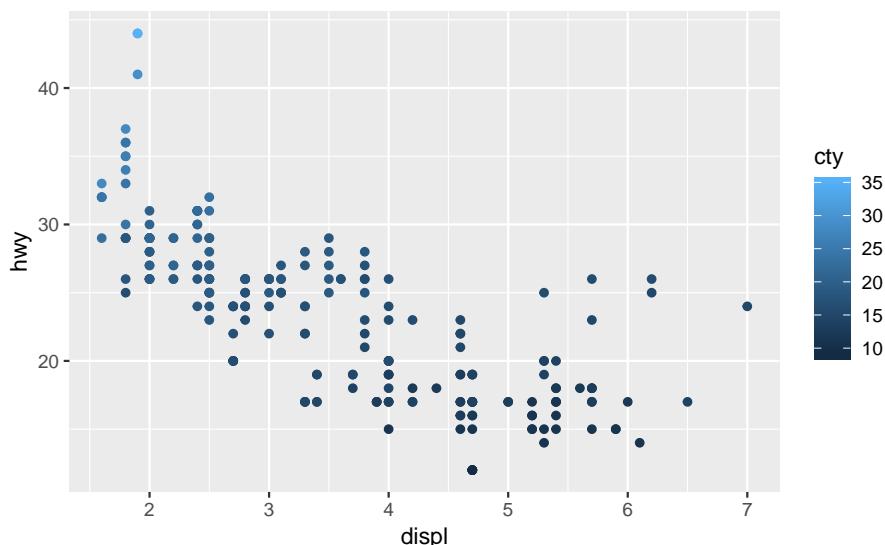
```
glimpse(mpg)
#> Observations: 234
#> Variables: 11
#> $ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", ...
#> $ model          <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4 qua...
#> $ displ           <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, ...
#> $ year            <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1...
#> $ cyl             <int> 4, 4, 4, 4, 6, 6, 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", "4", "4", "4", ...
#> $ cty              <int> 18, 21, 20, 21, 16, 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", ...
#> $ class            <chr> "compact", "compact", "compact", "compact", "comp...
```

Exercise 3.3.3

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

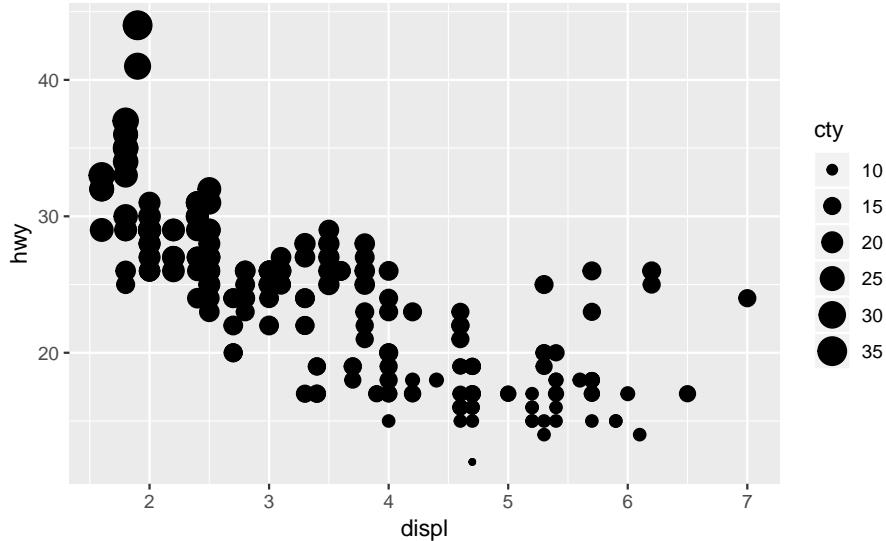
The variable `cty`, city highway miles per gallon, is a continuous variable:

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

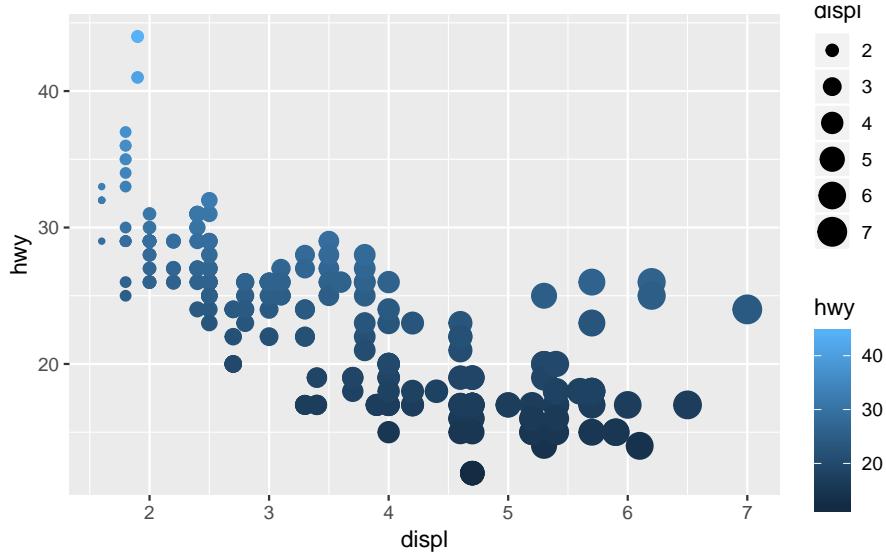
```
ggplot(mpg, aes(x = displ, y = hwy, shape = cty)) +
  geom_point()
#> Error: A continuous variable can not be mapped to shape
```

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.

Exercise 3.3.4

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

```
ggplot(mpg, aes(x = displ, y = hwy, colour = 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.

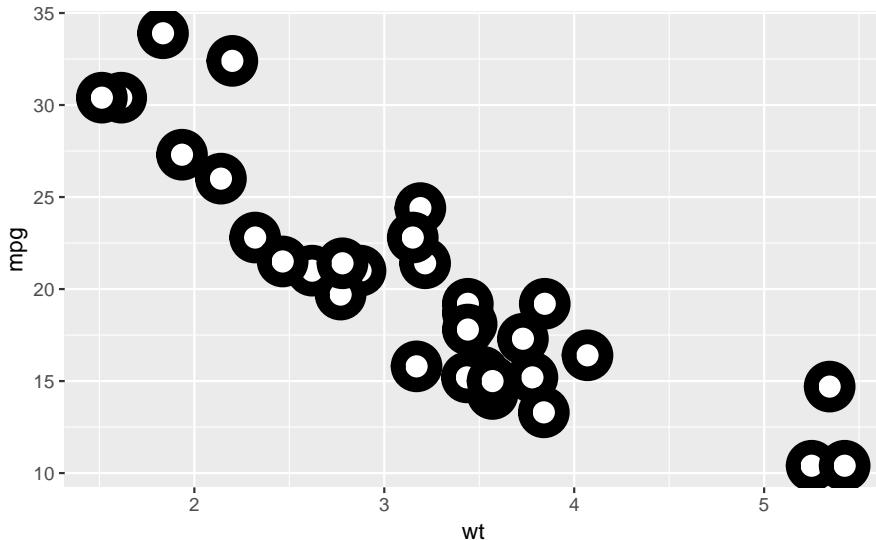
Exercise 3.3.5

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

Stroke changes the size of the border for shapes (21-25). These are filled shapes in which the color and size of the border can differ from that of the filled interior of the shape.

For example

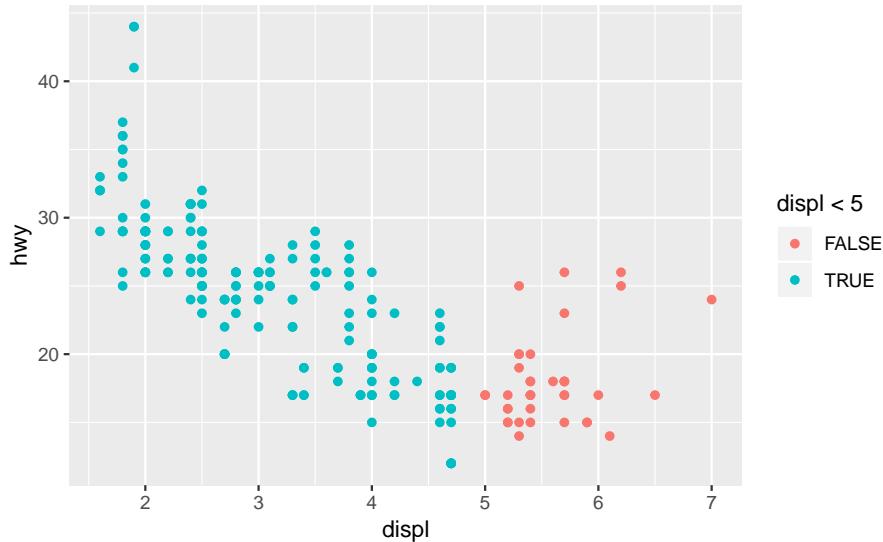
```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(shape = 21, colour = "black", fill = "white", size = 5, stroke = 5)
```



Exercise 3.3.6.

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, `colour = "blue"` created a categorical variable that only had one category: “blue”.

3.4 Common problems

No exercises

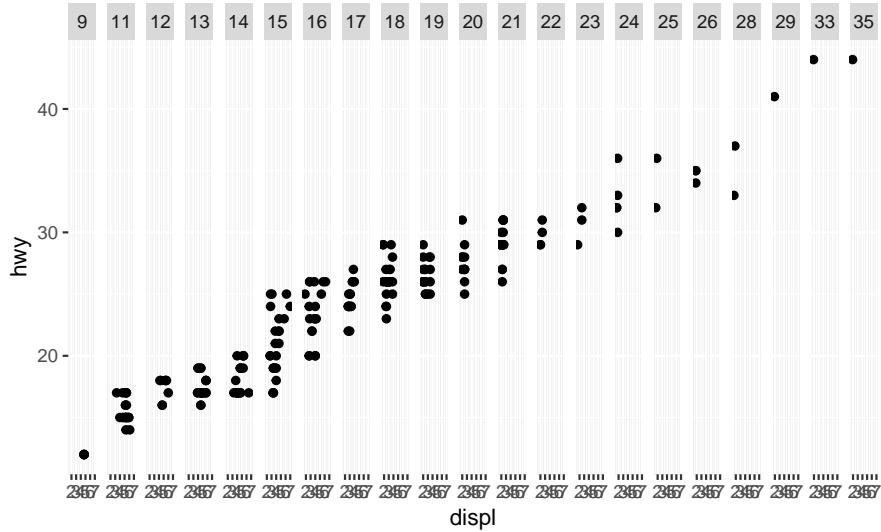
3.5 Facets

Exercise 3.5.1

What happens if you facet on a continuous variable?

Let's see.

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



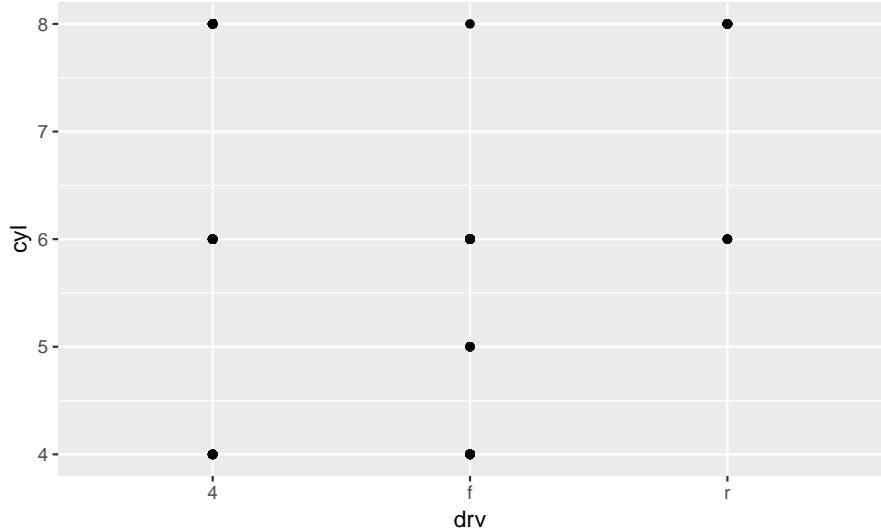
It converts the continuous variable to a factor and creates facets for **all** unique values of it.

Exercise 3.5.2

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.

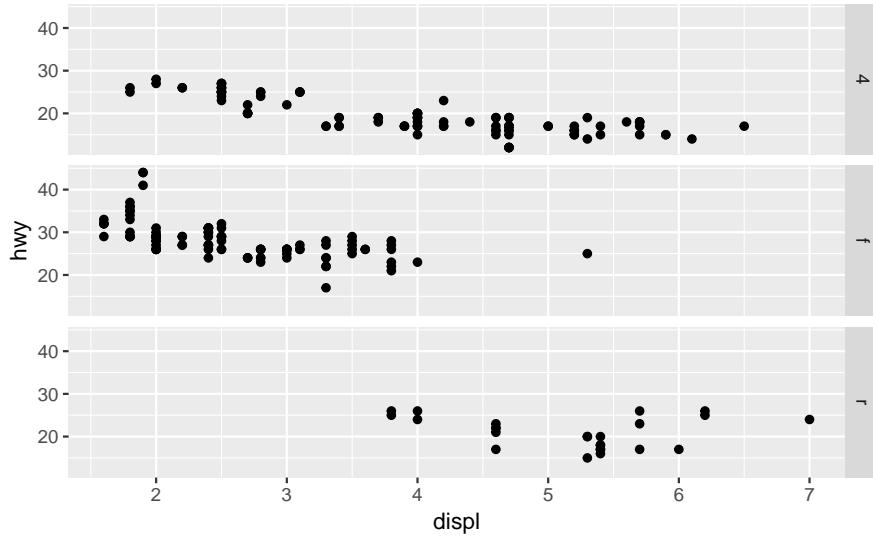
Exercise 3.5.3

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

The symbol `.` ignores that dimension for facetting.

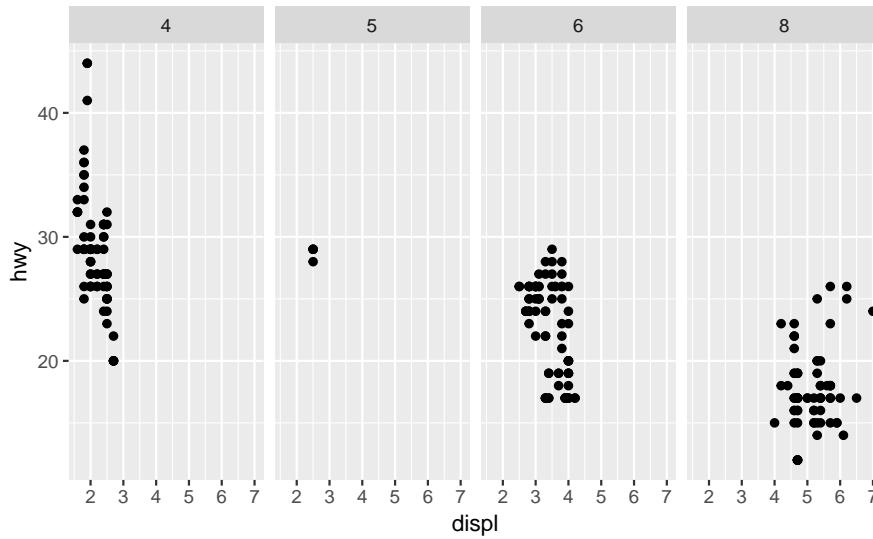
This plot facets by values of `drv` on the y-axis:

```
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:

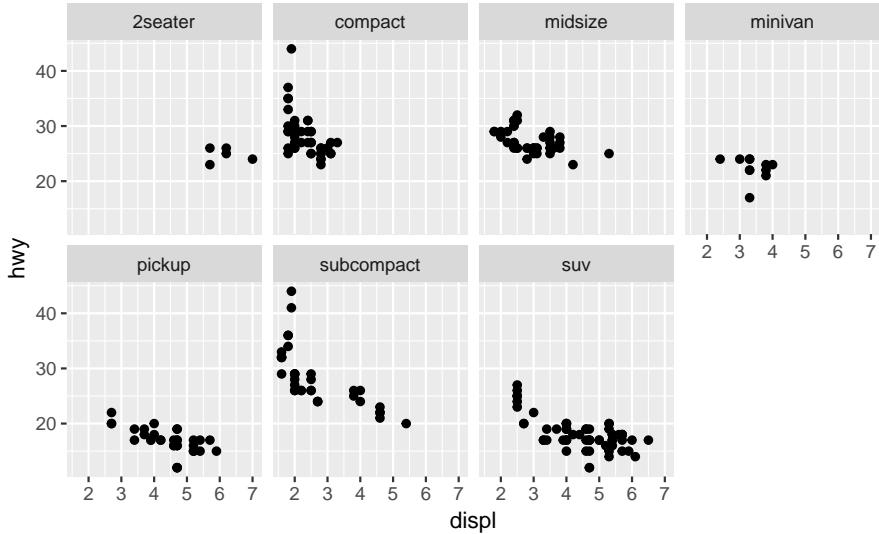
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(. ~ cyl)
```



Exercise 3.5.4

Take the first faceted plot in this section:

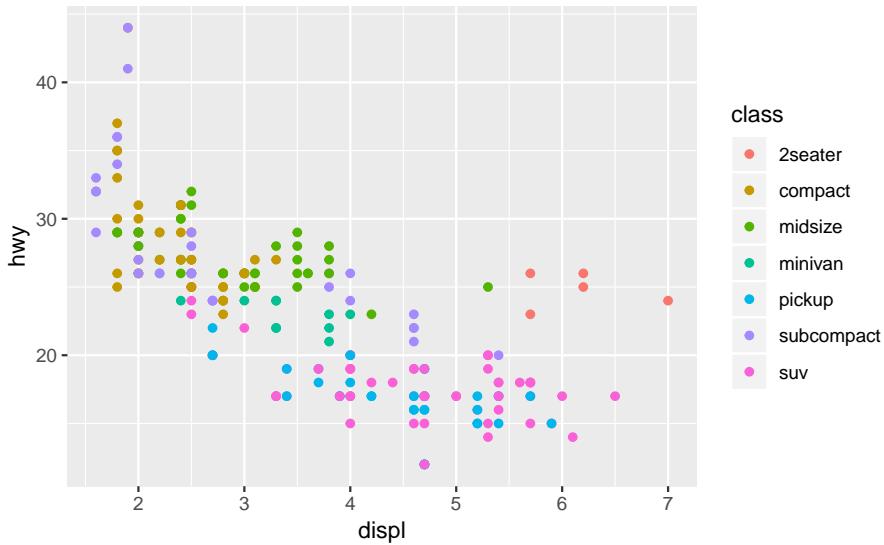
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```



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

This is what the plot looks like when `class` is represented by the colour the color aesthetic instead of faceting.

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



Advantages of encoding `class` with facets instead of color include the ability to encode more distinct categories. For me, it is difficult to distinguish color of "midsize" and the teal of "minivan" points are difficult to distinguish. Given human visual perception, the max number of colors to use when encoding unordered categorical (qualitative) data is nine, and in practice, often much less than that. Also, while placing points in different categories in different scales makes it difficult to directly compare values of individual points in different categories, it can make it easier to compare patterns between categories.

Disadvantages of encoding `class` with facets instead of color are that different the different class is that the points for each category are on different plots, making it more difficult to directly compare the locations of individual points. Using the same x- and y-scales for all facets lessens this disadvantage. Since encoding class within color also places all points on the same plot, it visualizes the unconditional relationship between the x and y variables; with facets, the unconditional relationship is no longer visualized since the points are spread across multiple plots.

The benefits encoding a variable through facetting over color become more advantageous as either the number of points or the number of categories increase. In the former, as the number of points increase, there is likely to be more overlap.

It is difficult to handle overlapping points with color. Jittering will still work with color. But jittering will only work well if there are few points and the classes do not overlap much, otherwise the colors of areas will no longer be distinct and it will be hard to visually pick out the patterns of different categories. Transparency (`alpha`) does not work well with colors since the mixing of overlapping transparent colors will no longer represent the colors of the categories. Binning methods use already color to encode density, so color cannot be used to encode categories.

As noted before, as the number of categories increases, the difference between colors decreases, to the point that the color of categories will no longer be visually distinct.

Exercise 3.5.5

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.

Exercise 3.5.6

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

If the plot is laid out horizontally, there will be more space for columns. 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.

3.6 Geometric Objects

Exercise 3.6.1

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()`
- area chart: `geom_area()`

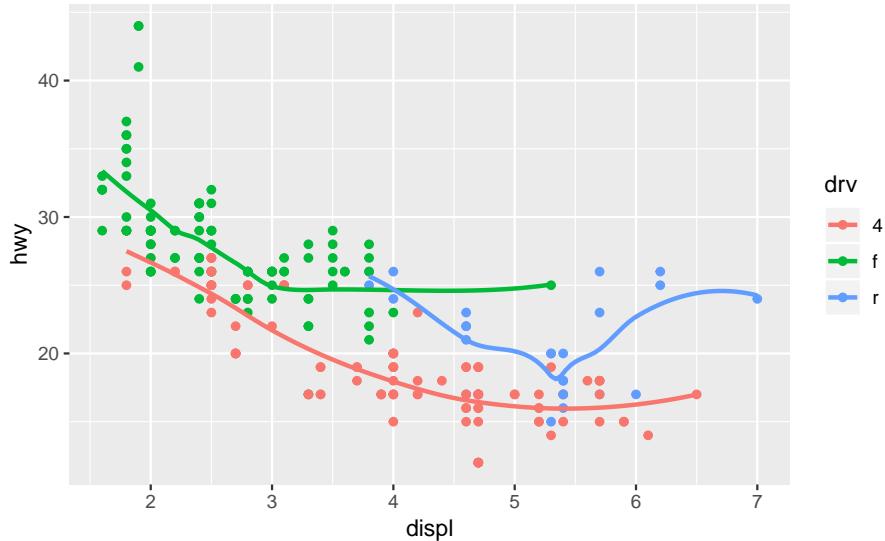
Exercise 3.6.2

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, colour = 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, colour = drv)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

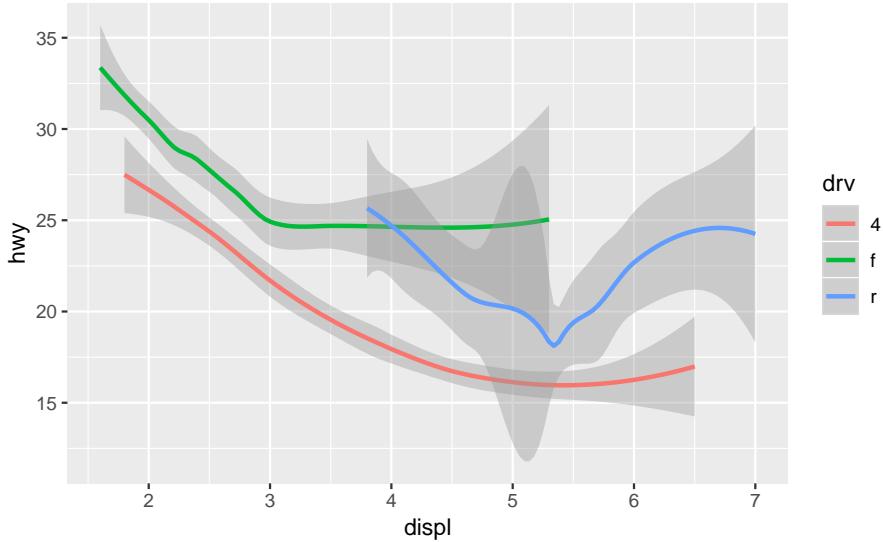


Exercise 3.6.3

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

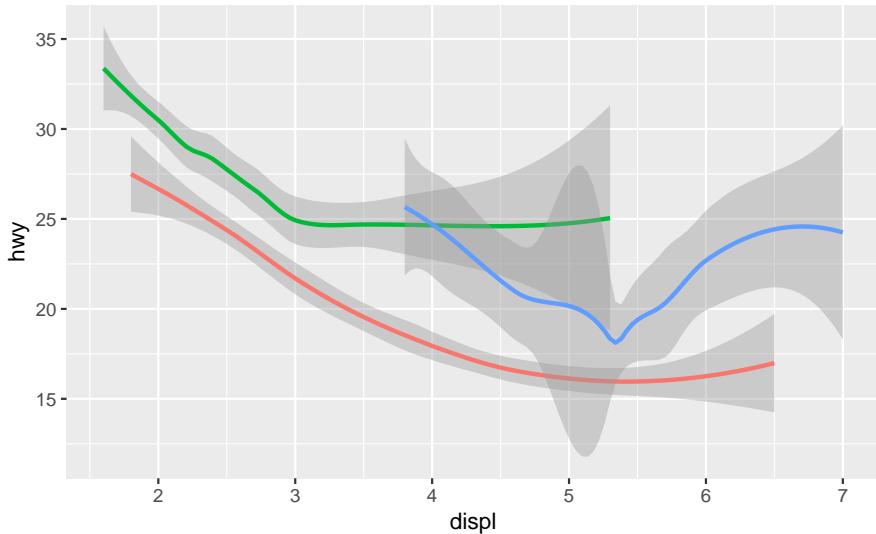
Show legend hides the legend box. In this code, without show legend, there is a legend.

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



But there is no legend in this code:

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



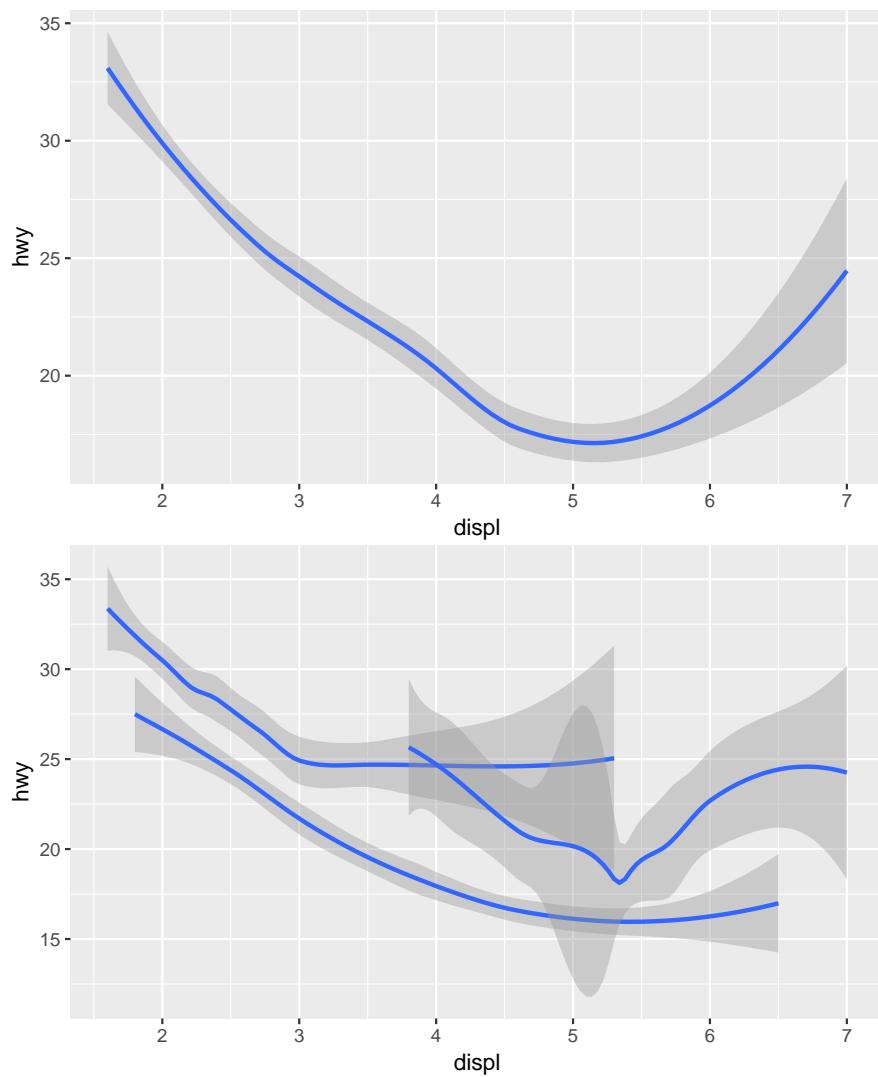
In the example earlier in the chapter,

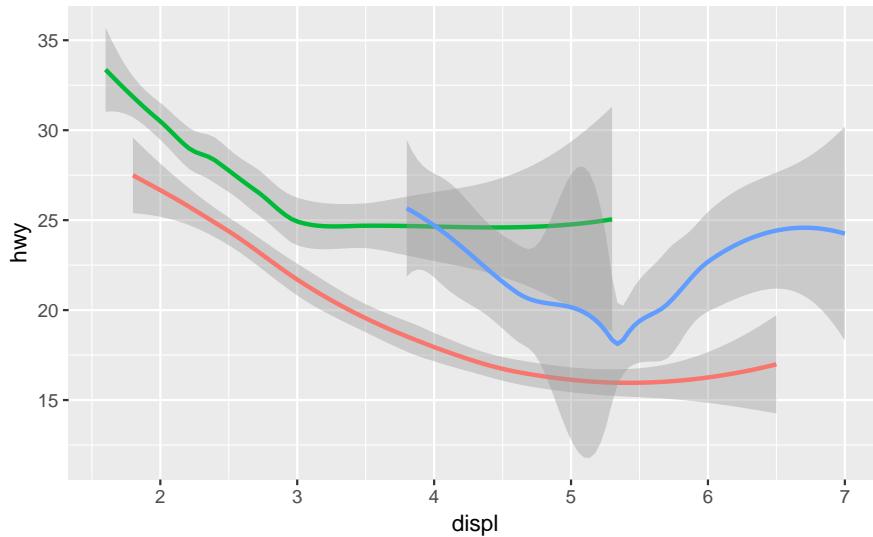
```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'

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

ggplot(data = mpg) +
```

```
geom_smooth(  
  mapping = aes(x = displ, y = hwy, colour = drv),  
  show.legend = FALSE  
)  
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```





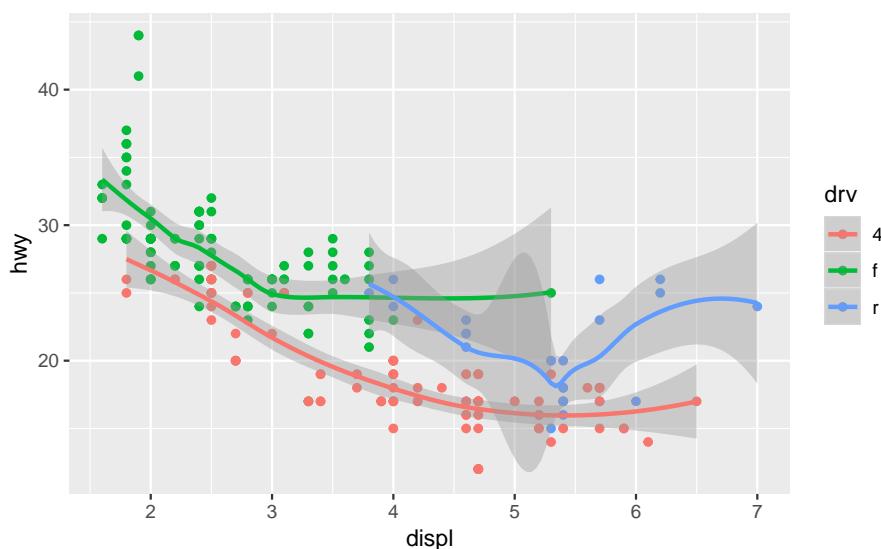
the legend is suppressed because there are three plots, and adding a legend that only appears in the last one would make the presentation asymmetric. Additionally, the purpose of this plot is to illustrate the difference between not grouping, using a `group` aesthetic, and using a `color` aesthetic (with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

Exercise 3.6.4

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

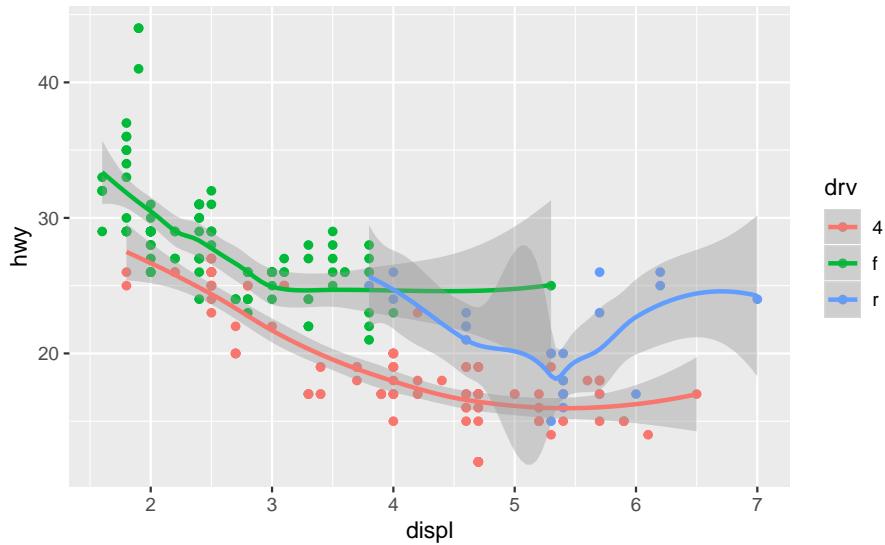
It adds standard error bands to the lines.

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



By default `se = TRUE`:

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

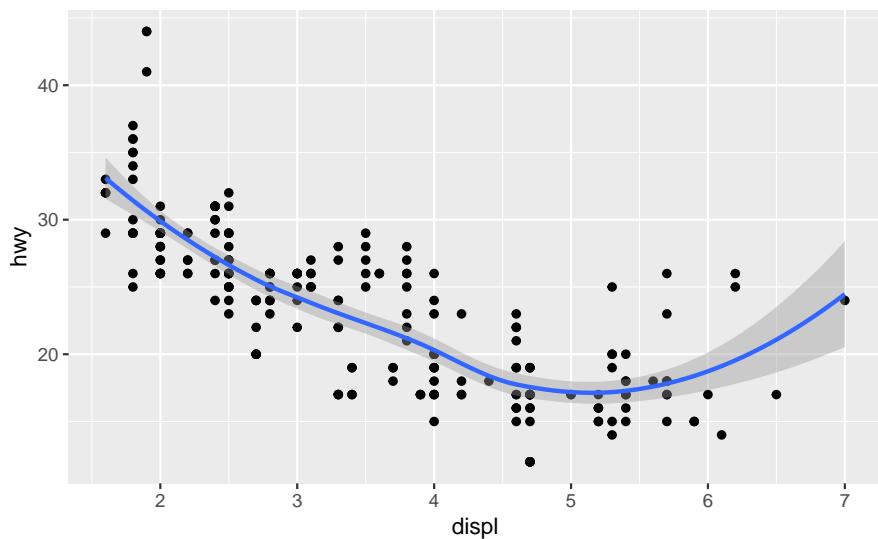


Exercise 3.6.5

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

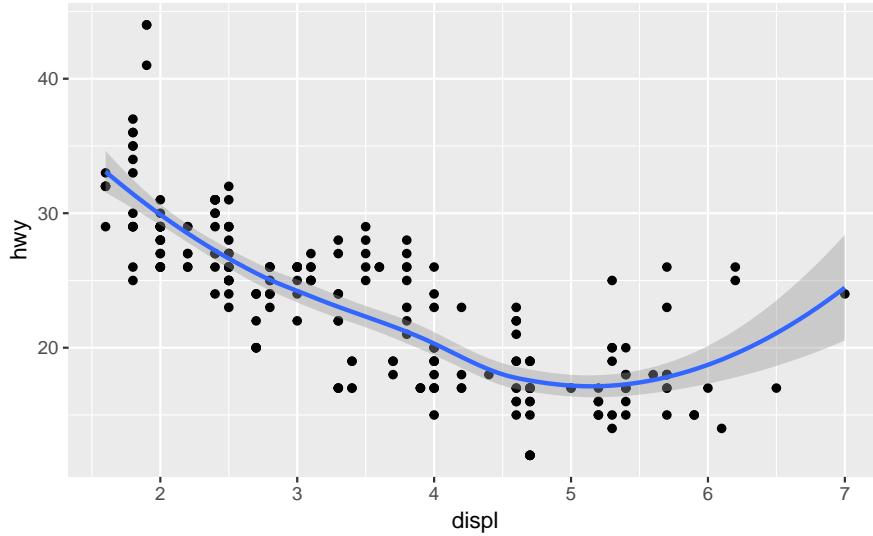
No. Because both `geom_point()` and `geom_smooth()` use the same data and mappings. They will inherit those options from the `ggplot()` object, and thus don't need to be specified again (or twice).

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



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

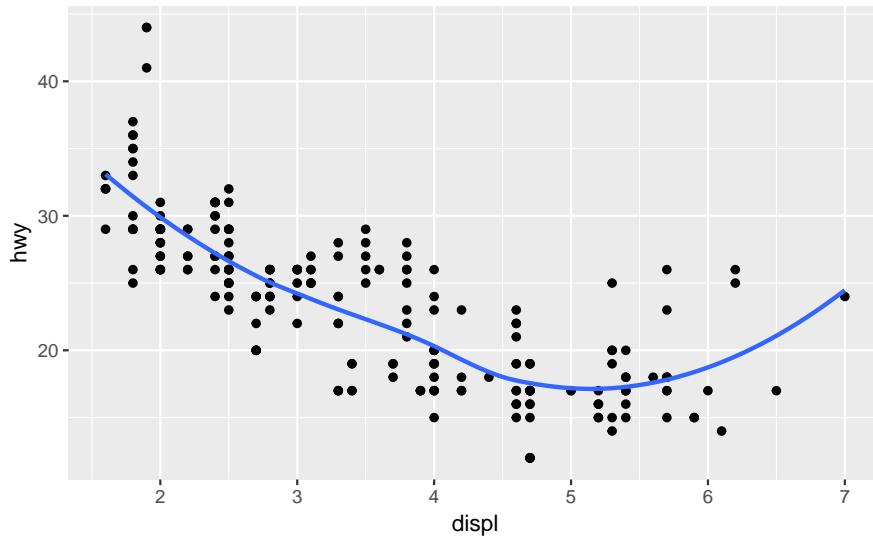
```
geom_smooth(data = mpg, mapping = aes(x = displ, y = hwy))
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



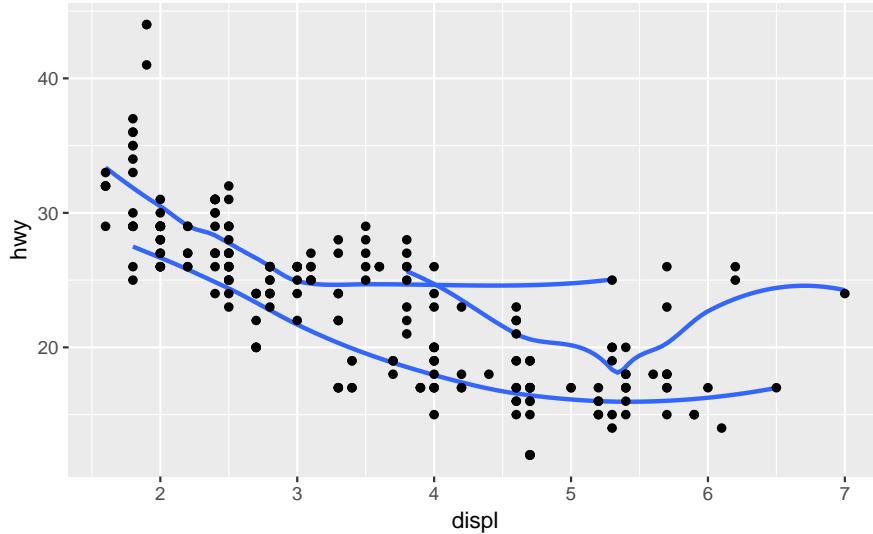
Exercise 3.6.6

Recreate the R code necessary to generate the following graphs.

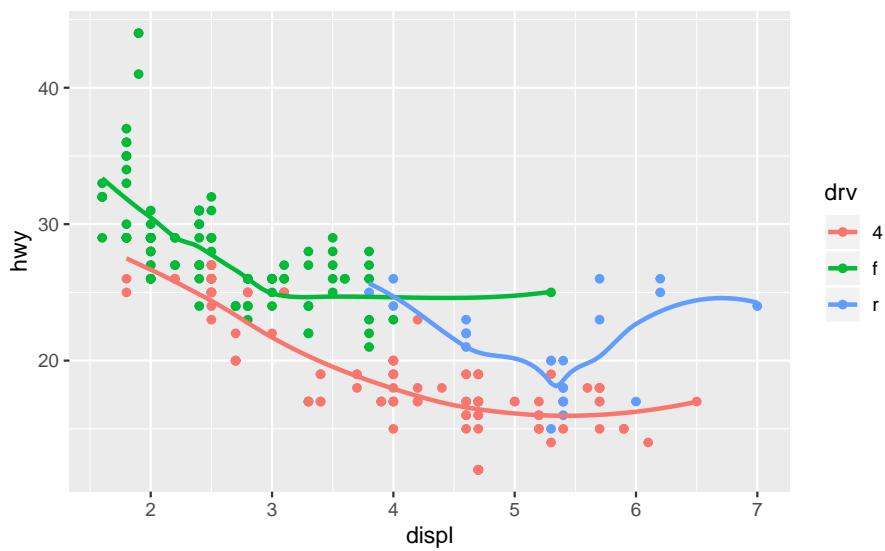
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



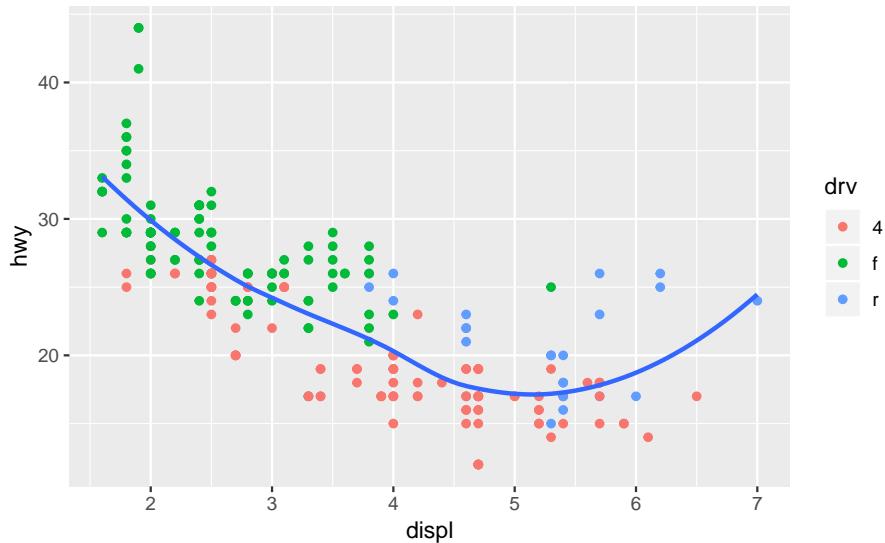
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(group = drv), se = FALSE) +
  geom_point()
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



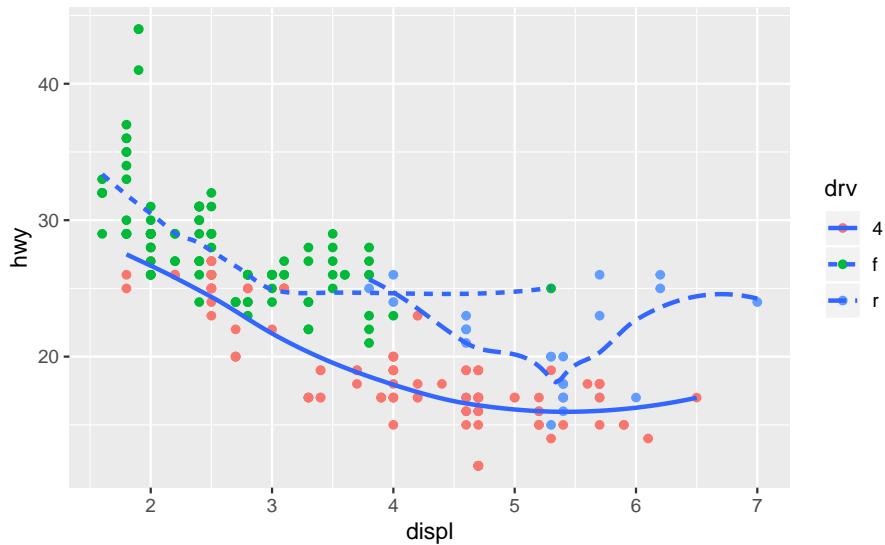
```
ggplot(mpg, aes(x = displ, y = hwy, colour = drv)) +
  geom_point() +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



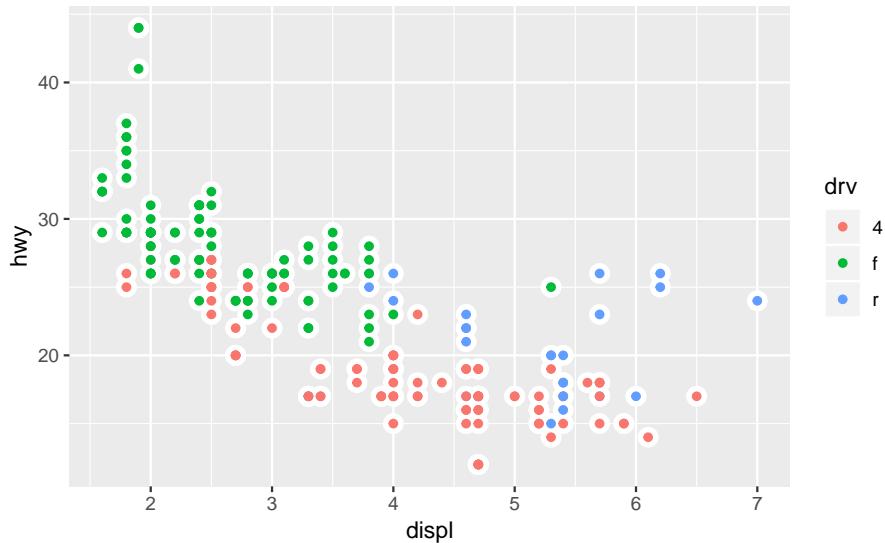
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(colour = drv)) +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



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



```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(size = 4, color = "white") +
  geom_point(aes(colour = drv))
```



3.7 Statistical Transformations

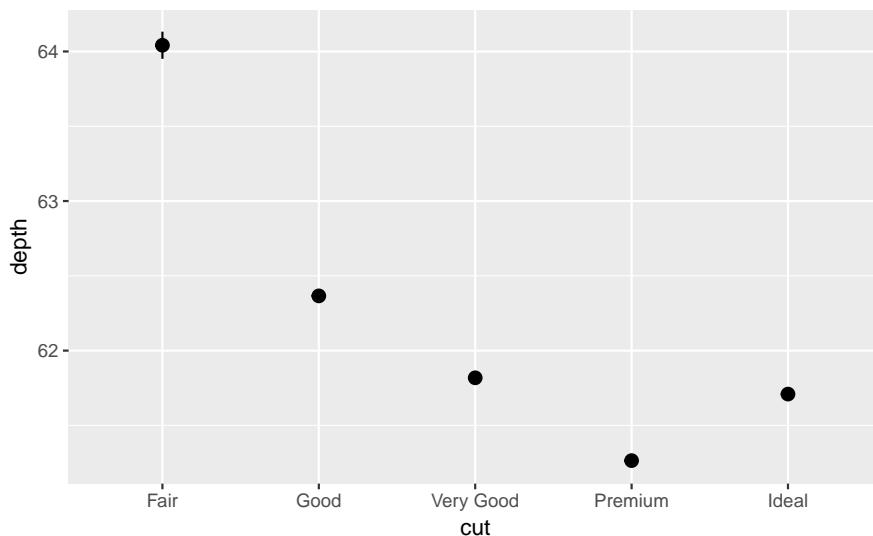
Exercise 3.7.1

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).

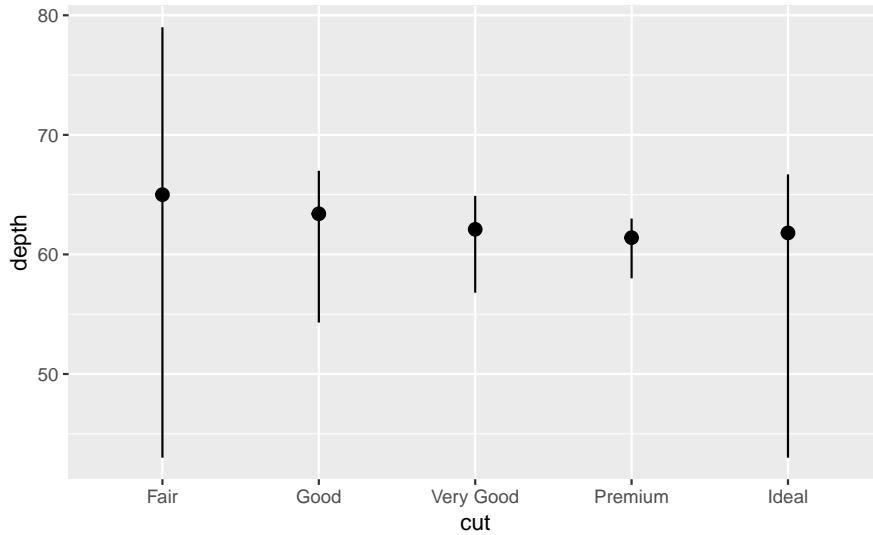
But, the default `stat` for `geom_pointrange()` is `identity()`, so use `geom_pointrange(stat = "summary")`.

```
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
  )
```



Exercise 3.7.2.

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

The `geom_col()` function has different default than `geom_bar()`. The default stat of `geom_col()` is `identity()`. This means that `geom_col()` expects that the data is already preprocessed into xvalues and yvalues representing the bar height. The default stat of `geom_bar()` is `count()`. This means that `geom_bar()` expects the xvariable to contain multiple observations for each values, and it will handle counting the number of observations for each value of x in order to create the bar heights.

Exercise 3.7.3.

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.

TODO

Exercise 3.7.4.

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

The function `stat_smooth()` calculates the following statistics:

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

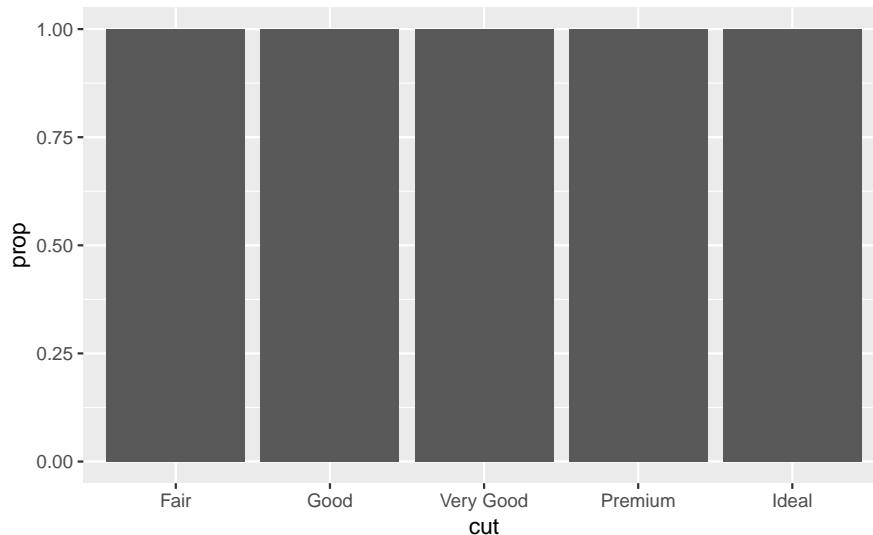
There's parameters such as `method` which determines which method is used to calculate the predictions and confidence interval, and some other arguments that are passed to that.

Exercise 3.7.5.

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.

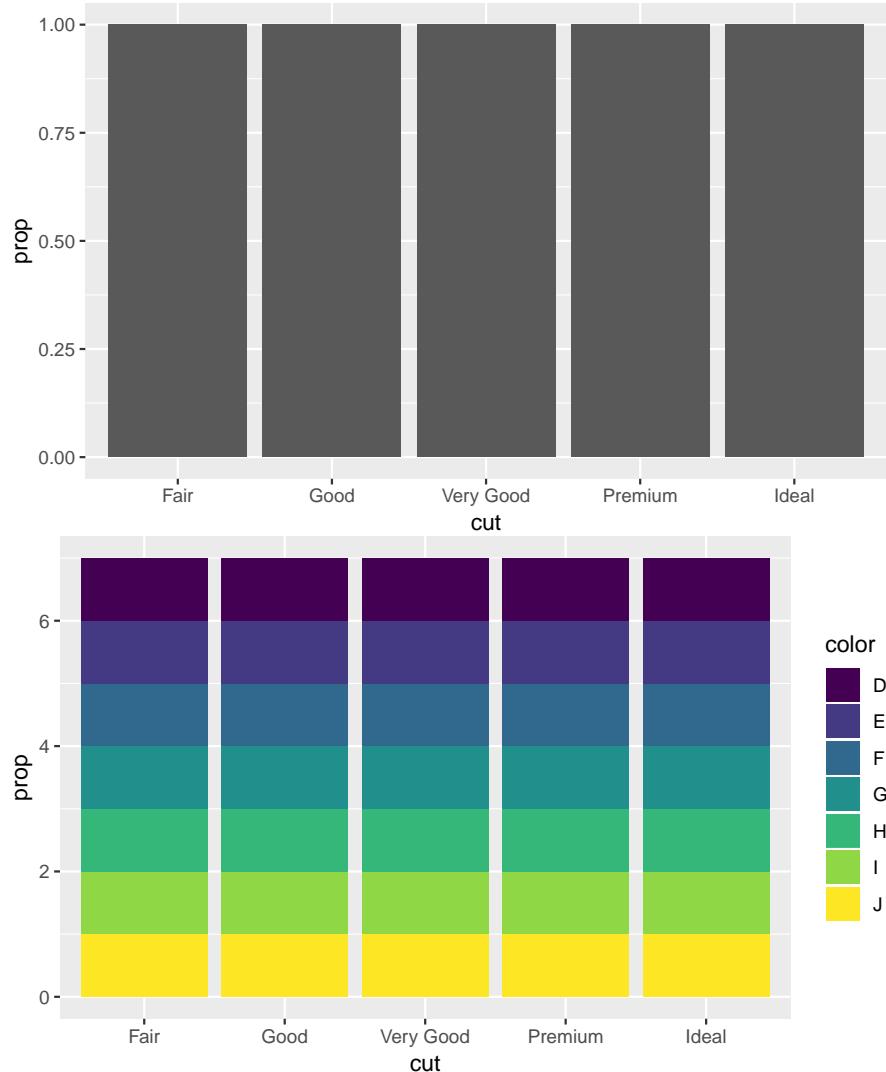
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop..))
```



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..))

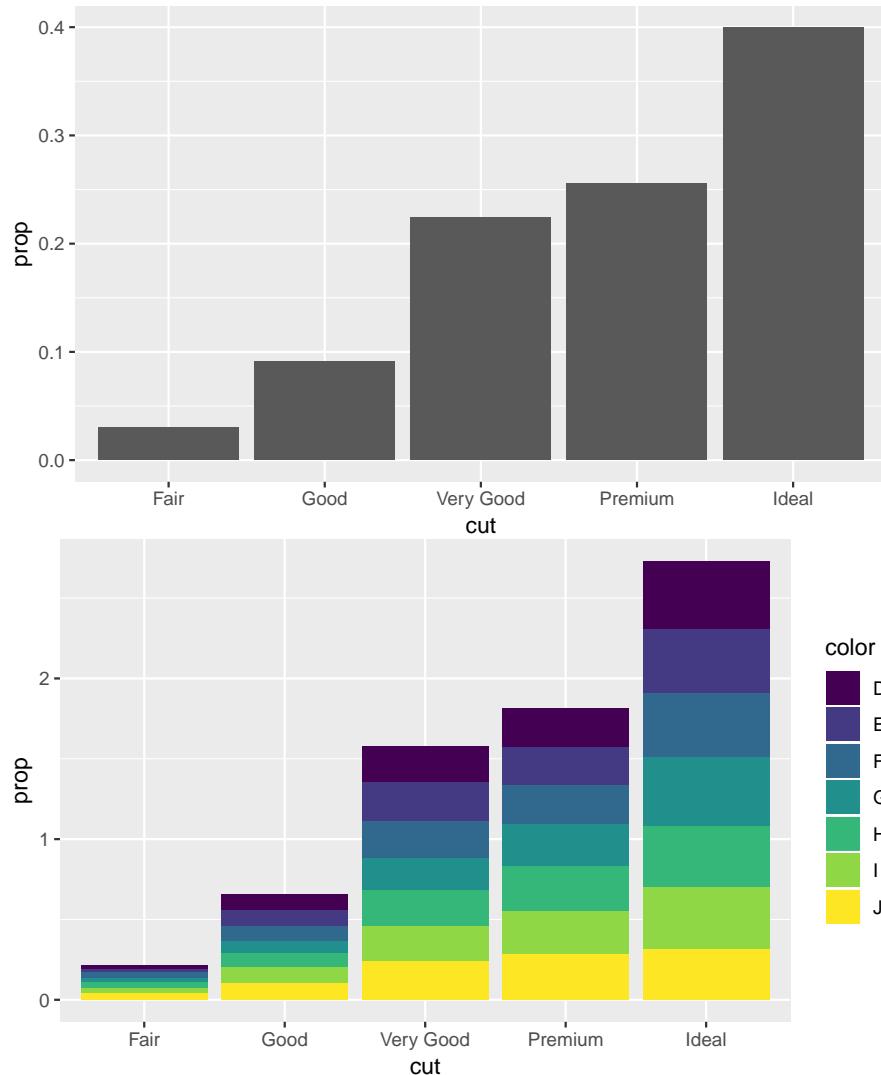
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = color, y = ..prop..))
```



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))
```



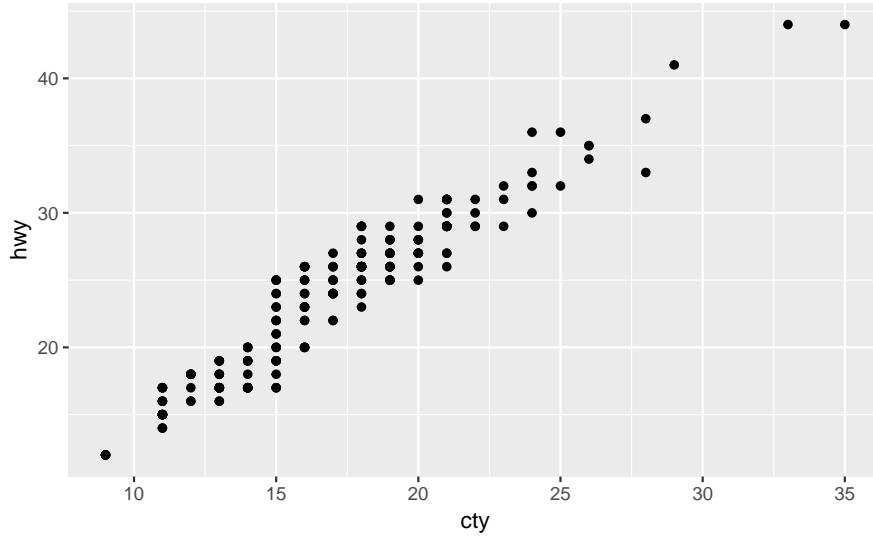
3.8 Position Adjustments

Exercise 3.8.1.

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

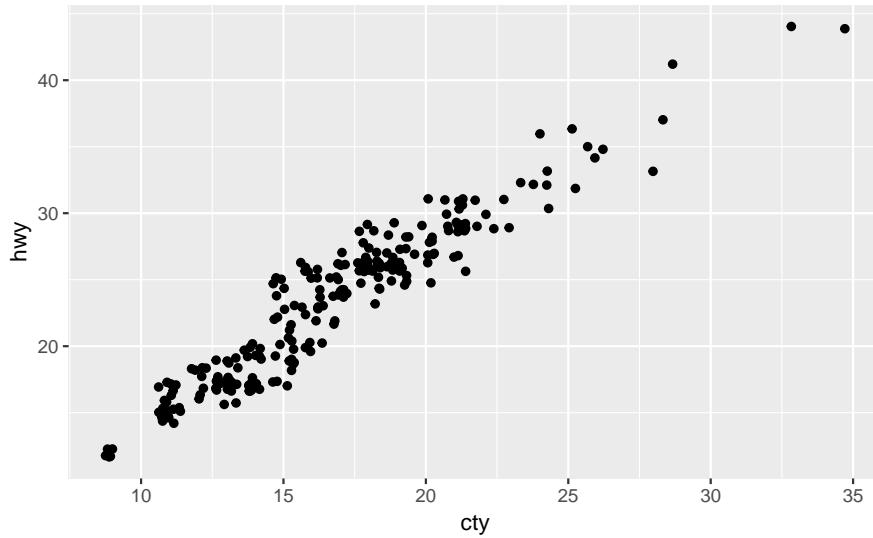
There is overplotting because there are multiple observations for each combination of `cty` and `hwy`.

```
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")
```



Exercise 3.8.2.

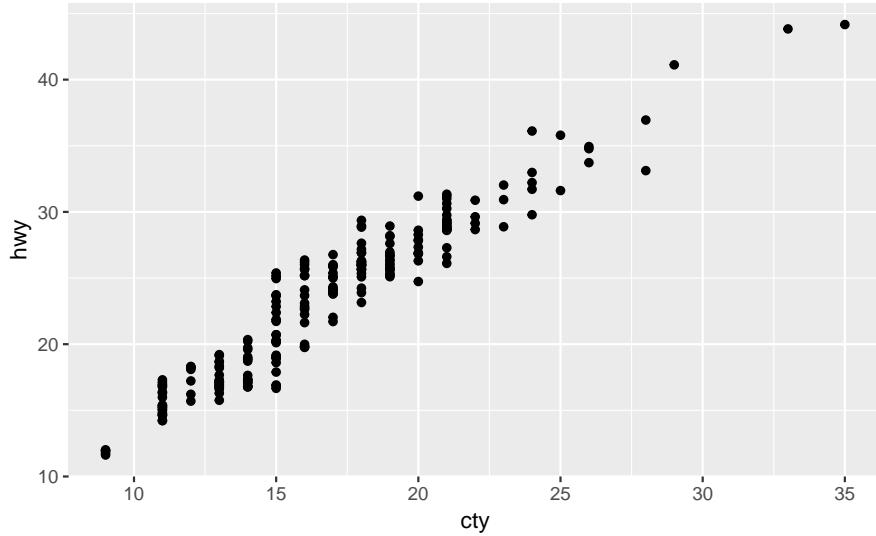
What parameters to `geom_jitter()` control the amount of jittering?

From the `geom_jitter()` documentation, there are two arguments to jitter:

- `width` controls the amount of vertical displacement, and
- `height` controls the amount of horizontal displacement.

The default values of `width` and `height` will introduce noise in both directions. Here is what the plot looks like with the default values of `height` and `width`.

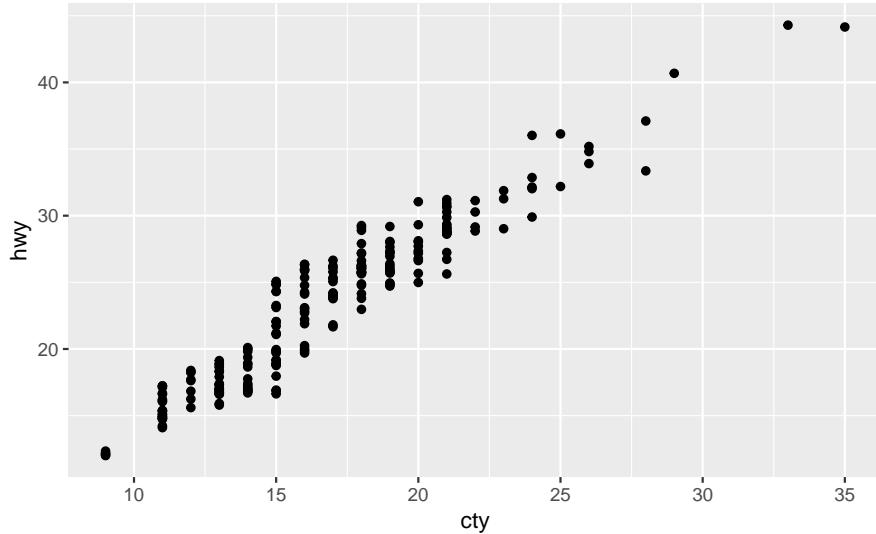
```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point(position = position_jitter(width = 0))
```



However, we can adjust them. Here are few examples to understand how adjusting these parameters affects the look of the plot.

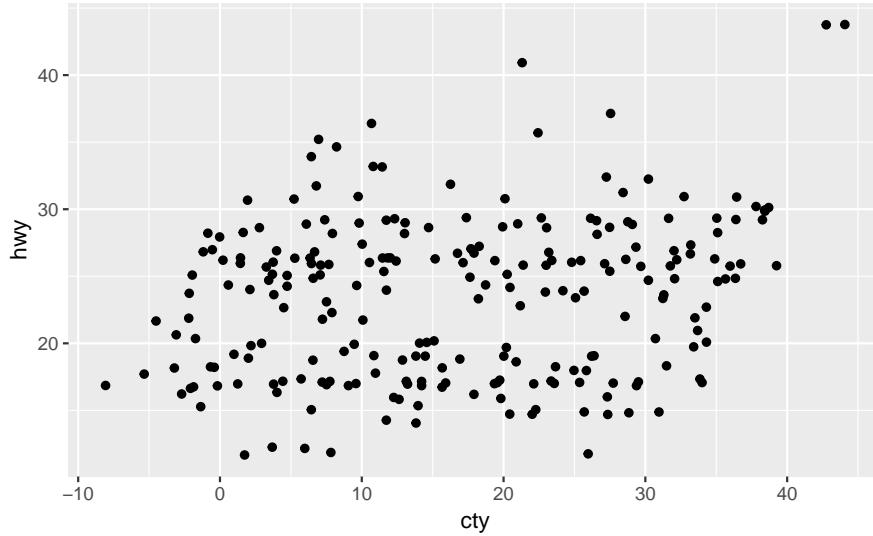
With `width = 0` there is no horizontal jitter.

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



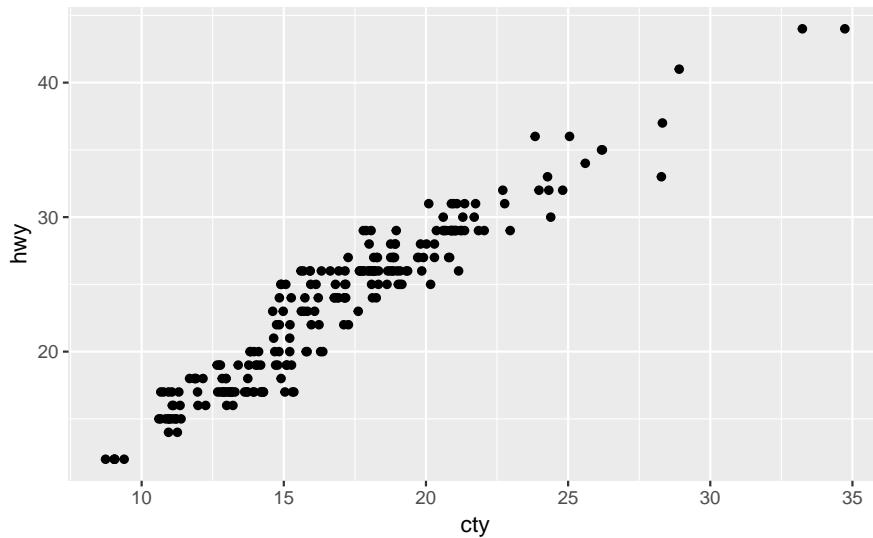
With `width = 20`, there is too much horizontal jitter.

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_jitter(width = 20)
```



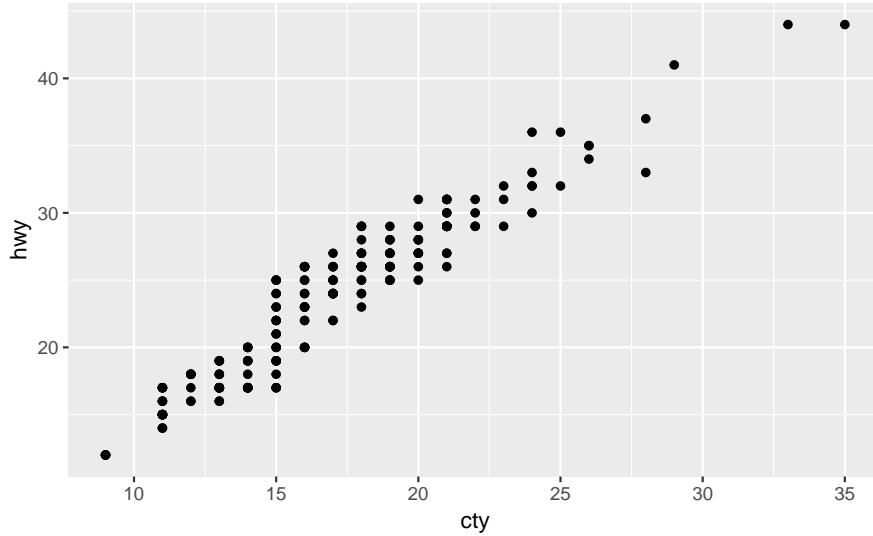
With `height = 0`, there is no vertical horizontal jitter:

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



With `height = 15`, there is too much vertical jitter.

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point(height = 15)
#> Warning: Ignoring unknown parameters: height
```



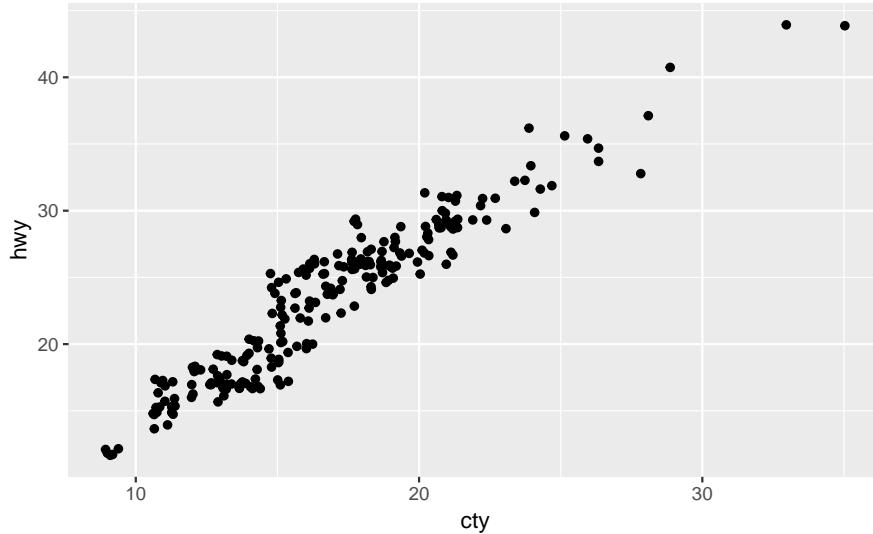
Note that the `height` and `width` arguments are in the units of the data. Thus `height = 1` corresponds to different relative amounts of jittering depending on the scale of the y variable. The default values of `height` and `width` are defined to be 80% of the `resolution()` of the data, which is the smallest non-zero distance between adjacent values of a variable. This means that if `x` and `y` are discrete variables, their resolutions are both equal to 1, and `height = 0.8` and `width = 0.8`.

Exercise 3.8.3.

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

`geom_jitter()` adds random noise to the locations points of the graph. In other words, it “jitters” the points. This method reduces overplotting since no two points are likely to have the same location after the random noise is added to their locations.

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

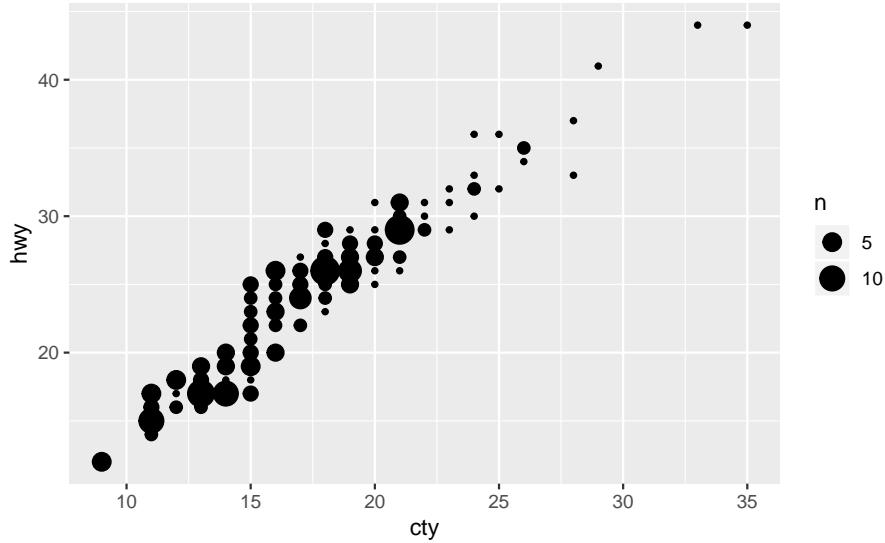


However, the reduction in overlapping comes at the cost of changing the x and y values of the points.

`geom_count()` resizes the points relative to the number of observations at each location. In other words,

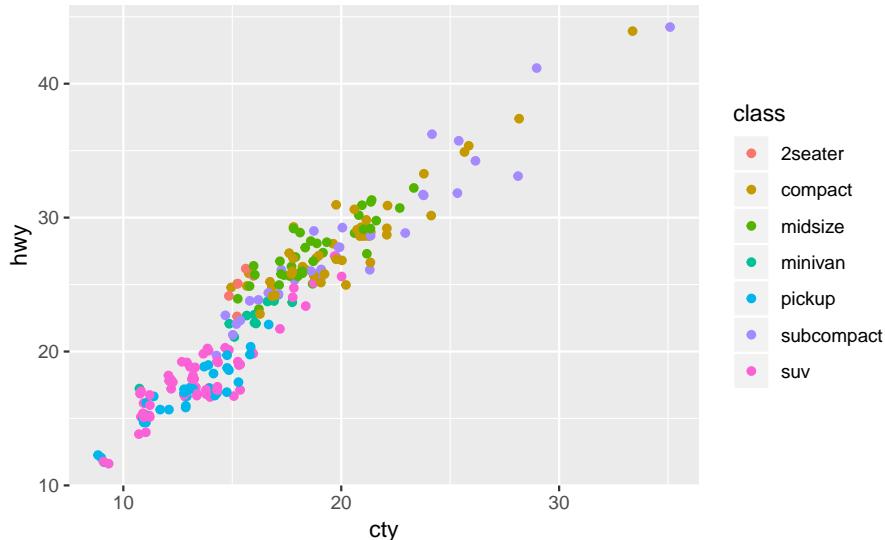
points with more observations will be larger than those with fewer observations.

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

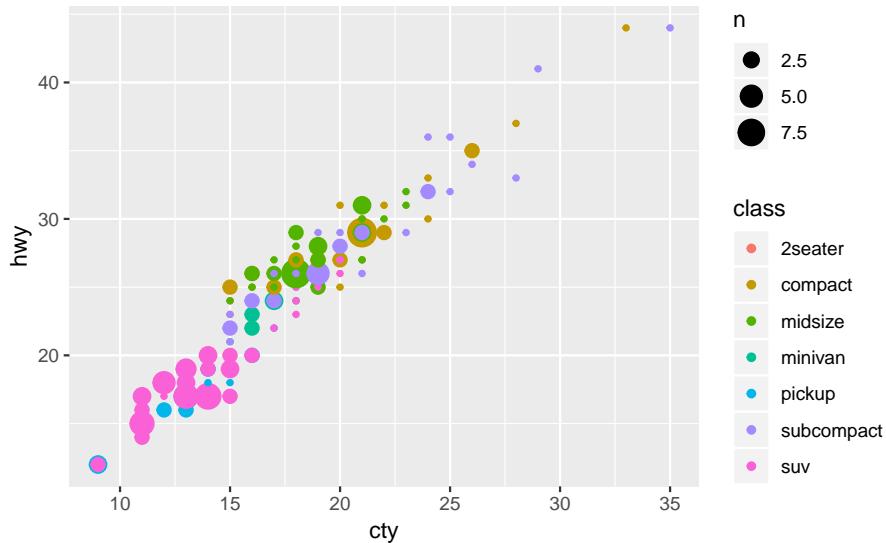


This method does not change the x and y coordinates of the points. However, if the points are close together and counts are large, the size of some points can itself introduce overplotting. For example, in the following example a third variable mapped to color is added to the plot. In this case, `geom_count()` is less readable than `geom_jitter()` when adding a third variable as color aesthetic.

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



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



Unfortunately, there is no universal solution to overplotting. The costs and benefits of different approaches will depend on the structure of the data and the goal of the data scientist.

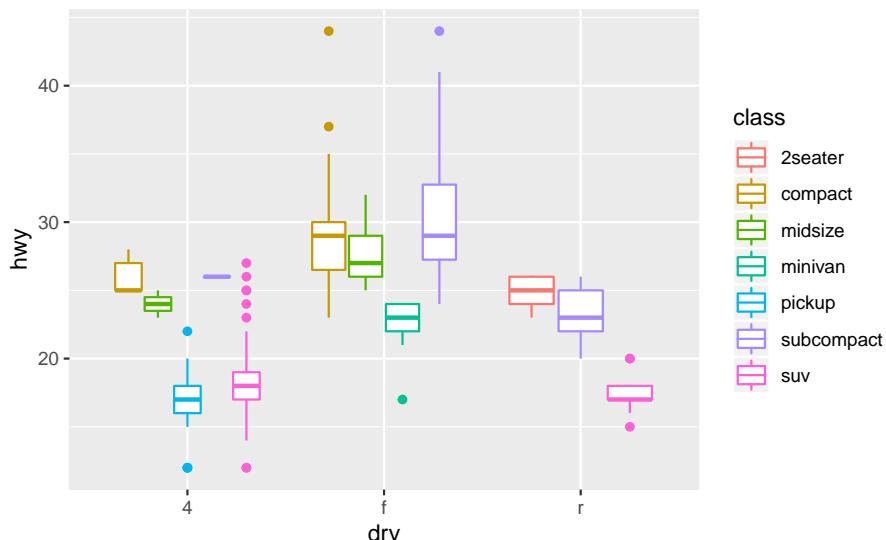
Exercise 3.8.4.

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

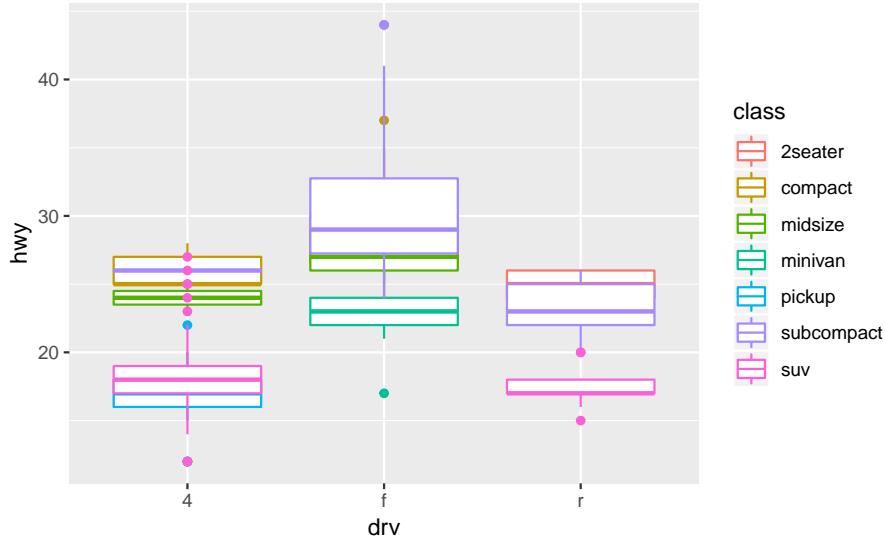
The default position for `geom_boxplot()` is `position_dodge()` (see its docs).

When we add `colour = 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, colour = class)) +
  geom_boxplot()
```



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



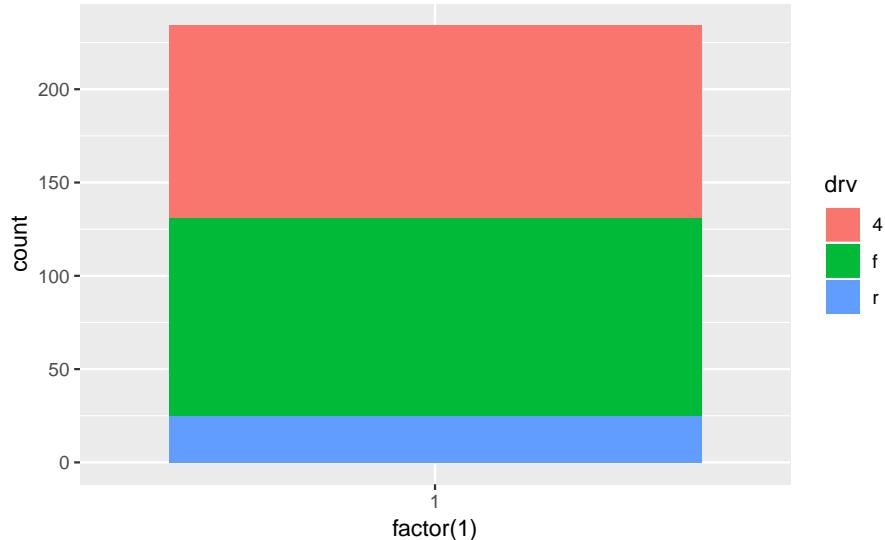
3.9 Coordinate Systems

Exercise 3.9.1

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

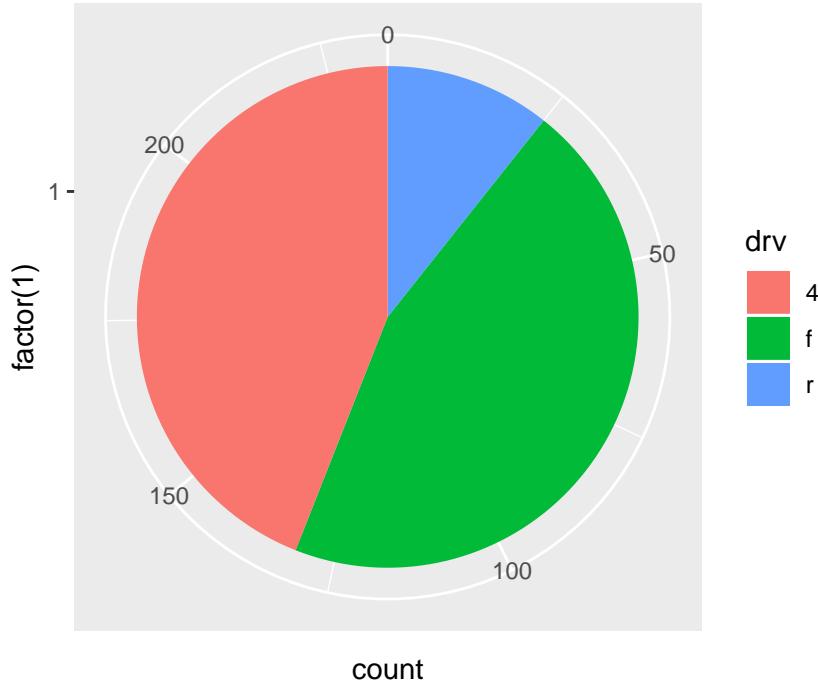
This is a stacked bar chart with a single category

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



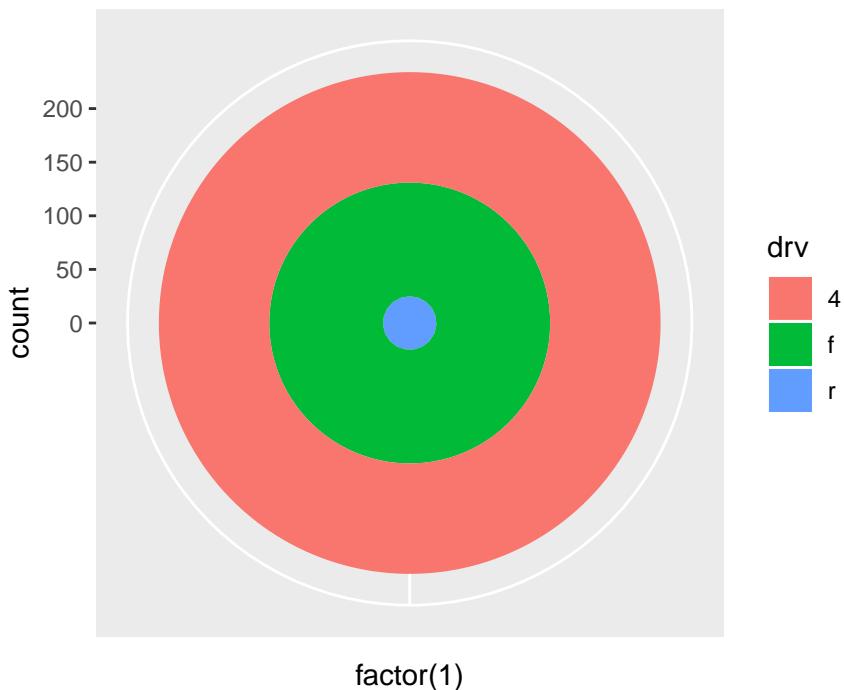
See the documentation for `coord_polar` for an example of making a pie chart. In particular, `theta = "y"`, meaning that the angle of the chart is the y variable which has to be specified.

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



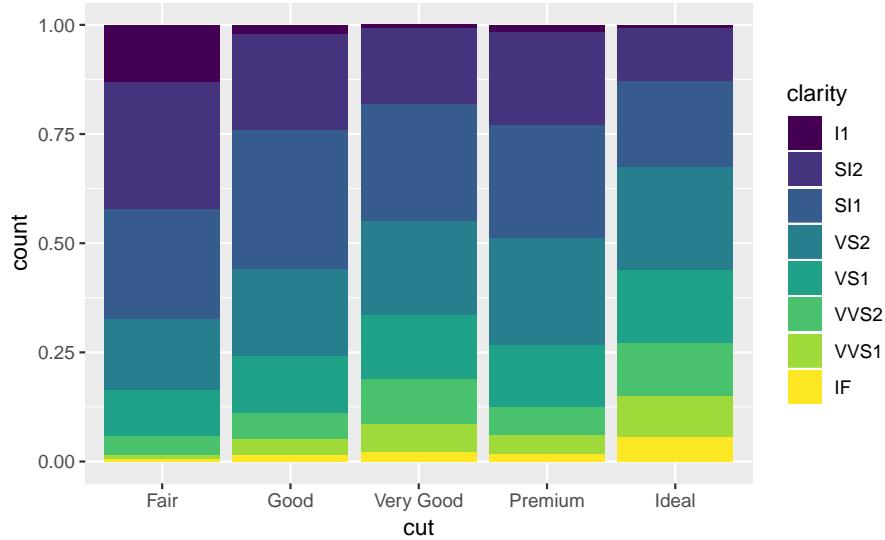
If `theta = "y"` is not specified, then you get a bull's-eye chart

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



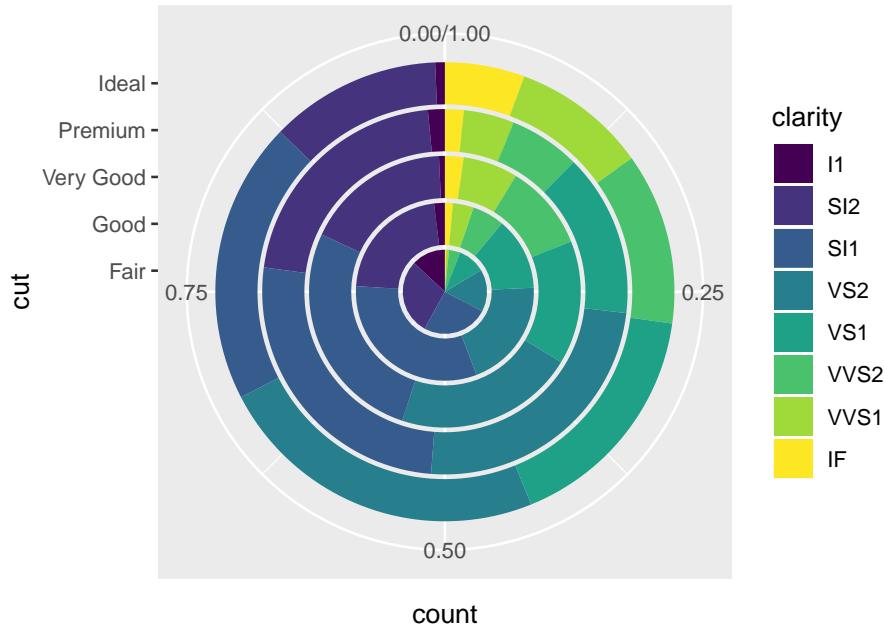
If you had a multiple stacked bar chart,

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



and apply polar coordinates to it, 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")
```

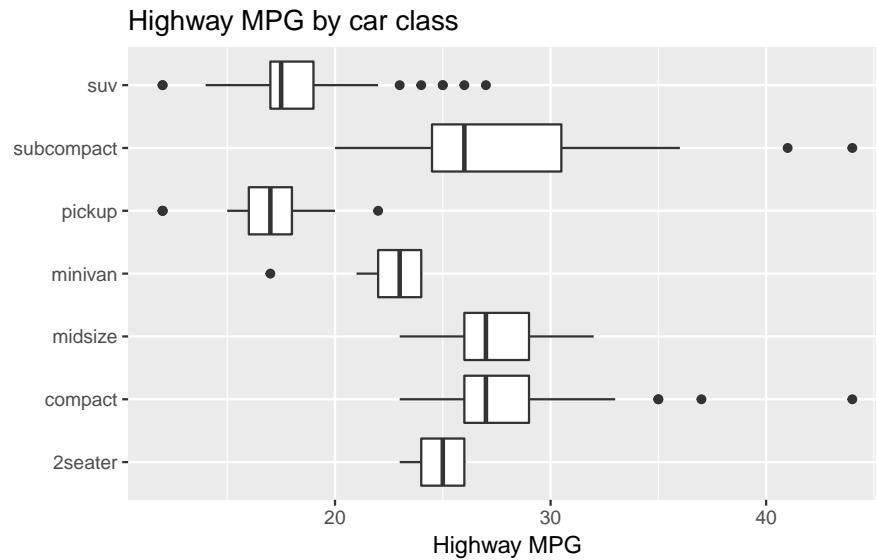


Exercise 3.9.2

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")
```



Exercise 3.9.3

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

`coord_map()` uses map projection to project 3-dimensional Earth onto a 2-dimensional plane. By default, `coord_map()` uses the Mercator projection. However, this projection must be applied to all geoms in the plot. `coord_quickmap()` uses a faster, but approximate map projection. This approximation ignores the curvature of Earth and adjusts the map for the latitude/longitude ratio. This transformation is quicker than `coord_map()` because the coordinates of the individual geoms do not need to be transformed.

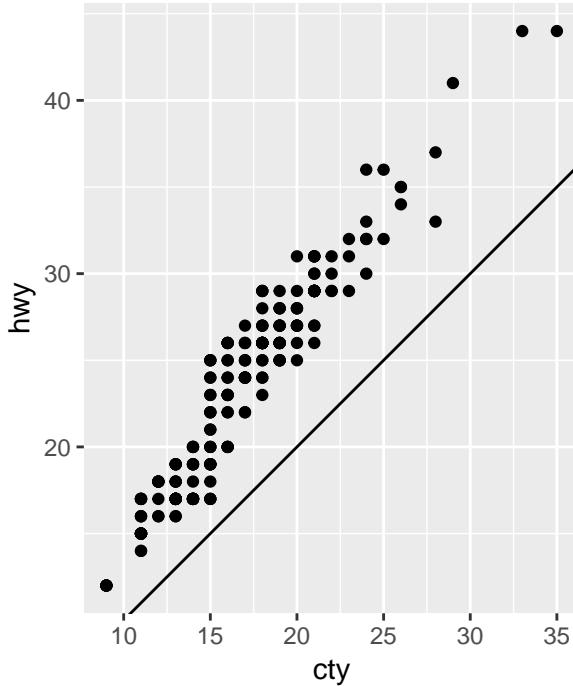
The `ggplot2` documentation contains more information on and examples for these two functions.

Exercise 3.9.4

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?

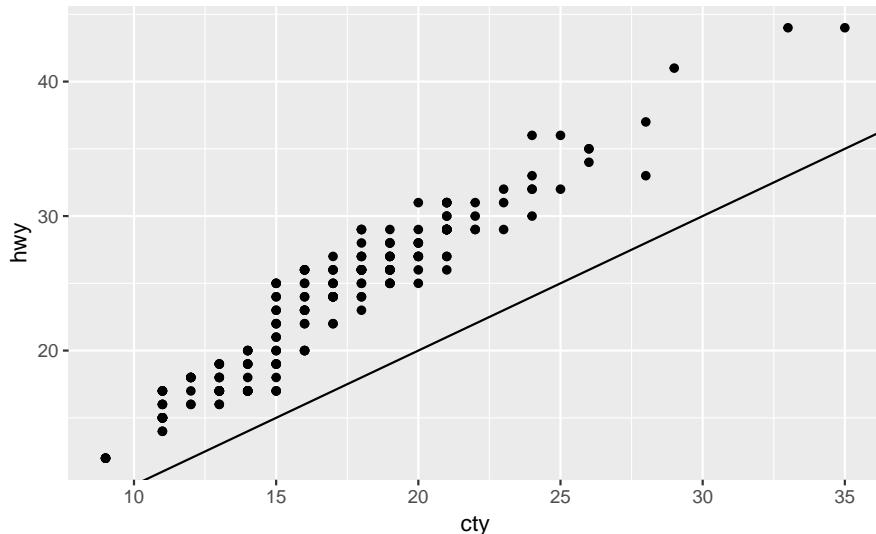
The function `coord_fixed()` ensures that the line produced by `geom_abline()` is at a 45 degree angle. The 45 degree line makes it easy to compare the highway and city mileage to the case in which city and highway MPG were equal.

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



If we didn't include `geom_coord()`, then the line would no longer have an angle of 45 degrees.

p



On average, humans are best able to perceive differences in angles relative to 45 degrees. See Cleveland (1993b), Cleveland (1994), Cleveland (1993a), Cleveland, McGill, and McGill (1988), Heer and Agrawala (2006) for discussion on how the aspect ratio of a plot affects perception of the values it encodes, evidence that 45 degrees is generally optimal, and methods to calculate the an aspect ratio to achieve it. The function `ggthemes::bank_slopes()` will calculate the optimal aspect ratio to bank slopes to 45 degrees.

3.10 The Layered Grammar of Graphics

No exercises

Chapter 4

Workflow: basics

Prerequisites

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

4.1 Coding basics

No exercises

4.2 What's in a name?

No exercises

4.3 Calling functions

No exercises

4.4 Practice

Exercise 4.4.1

Why does this code not work?

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

The variable being printed is `my_variable`, not `my_variable`: the seventh character is “I” (“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. It is especially important to distinguish between two sets of similar looking characters:

- the numeral zero (0), the Latin small letter O (o), and the Latin capital letter O (O),
- the numeral one (1), the Latin small letter I (i), the 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. Some examples of fonts with dotted or slashed zero glyphs are Consolas, Deja Vu Sans Mono, Monaco, Menlo, Source Sans Pro, and FiraCode.

Error messages of the form "object '...' not found" mean exactly what they say. R cannot find an object with that name. Unfortunately, the error does not tell you why that object cannot be found, because R does not know the reason that the object does not exist. The most common scenarios in which I encounter this error message are

1. I forgot to create the object, or an error prevented the object from being created.
2. I made a typo in the object's name, either when using it or when I created it (as in the example above), or I forgot what I had originally named it. If you find yourself often writing the wrong name for an object, it is a good indication that the original name was not a good one.
3. I forgot to load the package that contains the object using `library()`.

Exercise 4.4.2

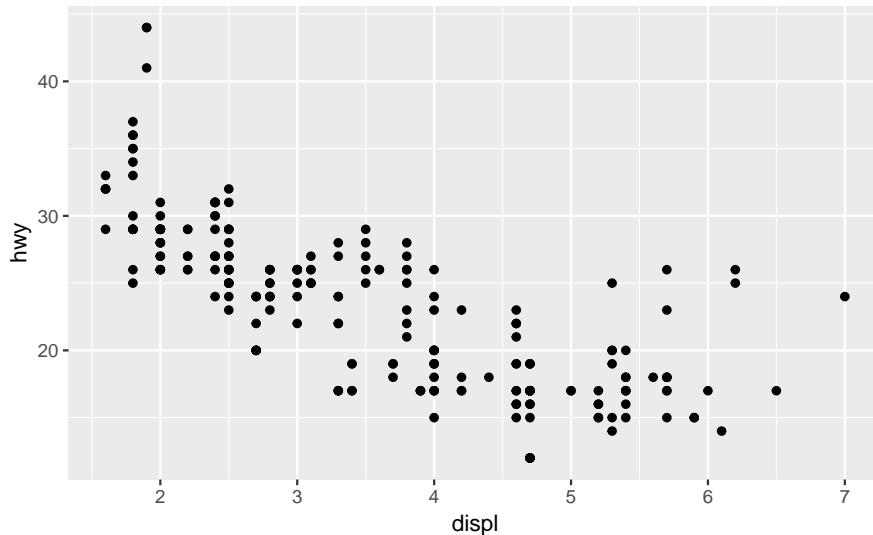
Tweak each of the following R commands so that they run correctly:

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
#> Error in FUN(X[[i]], ...): object 'displ' not found
```

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

It looks like a typo, `data` 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~   4.2  2008     8 auto~ 4      16    23 p   mids~
#> 2 chevrolet    c150~ 5.3  2008     8 auto~ r      14    20 r   suv
#> 3 chevrolet    c150~ 5.3  2008     8 auto~ r      11    15 e   suv
#> 4 chevrolet    c150~ 5.3  2008     8 auto~ r      14    20 r   suv
#> 5 chevrolet    c150~ 5.7  1999     8 auto~ r      13    17 r   suv
#> 6 chevrolet    c150~ 6.0  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  9.1   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.

Exercise 4.4.3

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()`

Exercise 5.2.1

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)

The answer to each part follows.

1. Since delay is in minutes, find flights whose arrival was delayed 120 or more minutes.

```
filter(flights, arr_delay >= 120)
#> # A tibble: 10,200 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 1.019e+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>
```

2. The flights that flew to Houston were are those flights where the destination (`dest`) is either “IAH” or “HOU”.

```
filter(flights, dest == "IAH" | dest == "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>
```

However, using `%in%` is more compact and would scale to cases where there were more than two airports we were interested in.

```
filter(flights, 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>
```

3. In the `flights` dataset, the column `carrier` indicates the airline, but it uses two-character carrier codes. We can find the carrier codes for the airlines in the `airlines` dataset. Since the carrier code dataset only has 16 rows, and the names of the airlines in that dataset are not exactly “United”, “American”, or “Delta”, it is easiest to manually look up their carrier codes in that data.

```
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
```

The carrier code for Delta is "DL", for American is "AA", and for United is "UA". Using these carriers codes, we check whether `carrier` is one of those.

```
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>
```

4. The variable `month` has the month, and it is numeric. So, the summer flights are those that departed in months 7 (July), 8 (August), and 9 (September).

```
filter(flights, month >= 7, month <= 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>
```

The `%in%` and `|` operators would also work, but using relational operators like `>=` and `<=` is preferred for numeric data.

```
filter(flights, month %in% c(7, 8, 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>
```

5. Flights that arrived more than two hours late, but didn't leave late will have an arrival delay of more than 120 minutes (`dep_delay > 120`) and an non-positive departure delay (`dep_delay <= 0`).

```
filter(flights, 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>
```

6. Were delayed by at least an hour, but made up over 30 minutes in flight. If a flight was delayed by at least an hour, then `dep_delay >= 60`. If the flight didn't make up any time in the air, then its arrival would be delayed by the same amount as its departure, meaning `dep_delay == arr_delay`, or alternatively, `dep_delay - arr_delay == 0`. If it makes up over 30 minutes in the air, then the arrival delay must be at least 30 minutes less than the departure delay, which is stated as `dep_delay - arr_delay > 30`.

```
filter(flights, 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>
```

7. Finding flights that departed between midnight and 6 am is complicated by the way in which times are represented in the data. In `dep_time`, midnight is represented by 2400, not 0. This means we cannot simply check that `dep_time < 600`, because we also have to consider the special case of midnight.

```
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>
```

Alternatively, we could use the modulo operator, `%%`. The modulo operator returns the remainder of division. Let's see how this affects our times.

```
c(600, 1200, 2400) %% 2400
#> [1] 600 1200 0
```

Since $2400 \% 2400 == 0$ and all other times are left unchanged, we can compare the result of the modulo operation to 600,

```
filter(flights, dep_time %% 2400 <= 600)
#> # 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>
```

This filter expression is more compact, but its readability will depends on the familiarity of the reader with modular arithmetic.

Exercise 5.2.2

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?

The expression `between(x, left, right)` is equivalent to `x >= left & x <= right`.

Of the answers in the previous question, we could simplify the statement of *departed in summer* (`month >= 7 & month <= 9`) using `between()` as the following

```
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>
```

Exercise 5.2.3

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

Find the rows of flights with a missing departure time (`dep_time`) using the `is.na()` function.

```
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>
```

Notably, the arrival time (`arr_time`) is also missing for these rows. These seem to be canceled flights.

Exercise 5.2.4

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 value of the missing `TRUE` or `FALSE`, `x` or `TRUE` is `TRUE` 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
```

Since $x * 0 = 0$ for all finite, numeric x , 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 $x \times \infty$ and $x \times -\infty$ is undefined. R represents undefined results as `NaN`, which is an abbreviation of “not a number”.

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

5.3 Arrange rows with `arrange()`

Exercise 5.3.1

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

We can put NA values first by sorting by both an indicator of whether the column has a missing value, and the column of interest. For example, to sort the data frame by departure time (`dep_time`) in ascending order, but place all missing values, run the following.

```
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>
```

Otherwise, regardless of whether we use `desc()` or not, missing values will be placed at the end.

```
arrange(flights, 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    13       1        2249      72     108
#> 2 2013     1    31       1        2100     181     124
#> 3 2013    11    13       1        2359      2     442
#> 4 2013    12    16       1        2359      2     447
#> 5 2013    12    20       1        2359      2     430
#> 6 2013    12    26       1        2359      2     437
#> # ... 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>
```

```
arrange(flights, desc(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    10    30      2400        2359      1     327
#> 2 2013    11    27      2400        2359      1     515
#> 3 2013    12     5      2400        2359      1     427
```

```
#> 4 2013 12 9 2400 2359 1 432
#> 5 2013 12 9 2400 2250 70 59
#> 6 2013 12 13 2400 2359 1 432
#> # ... 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>
```

Exercise 5.3.2

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

Find the most delayed flights by sorting the table by departure delay, `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>
```

The most delayed flight was HA 51, JFK to HNL, which was scheduled to leave on January 09, 2013 09:00. Note that the departure time is given as 641, which seems to be less than the scheduled departure time. But the departure was delayed 1,301 minutes, which is 21 hours, 41 minutes. The departure time is the day after the scheduled departure time. Be happy that you weren't on that flight, and if you happened to have been on that flight and are reading this, I'm sorry for you.

Similarly, the earliest departing flight can be found by sorting `dep_delay` in ascending order.

```
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>
```

Flight B6 97 (JFK to DEN) scheduled to depart on Saturday 07, 2013 at 21:23 departed 43 minutes early.

Exercise 5.3.3

Sort flights to find the fastest flights.

By “fastest” flights, I assume that the question refers to the flights with the shortest time in the air. Find these by sorting by `air_time`

```
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>
```

However, “fastest” could also be interpreted as referring to the average air speed. We can find these flights by sorting by the result of `distance / air_time * 60`, where the 60 is to convert the expression to miles per hour since `air_time` is in minutes.

```
arrange(flights, distance / air_time * 60)
#> # 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    28    1917        1825      52    2118
#> 2 2013     6    29     755        800      -5    1035
#> 3 2013     8    28     932        940      -8    1116
#> 4 2013     1    30    1037       955      42    1221
#> 5 2013    11    27     556        600      -4     727
#> 6 2013     5    21     558        600      -2     721
#> # ... 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>
```

This shows that we are not limited to sorting by columns in `arrange()`, but can sort by the results of arbitrary expressions, something which we had earlier seen with `desc()`.

Exercise 5.3.4

Which flights traveled the longest? Which traveled the shortest?

By longest (shortest), I assume that the question is asking about the distance traveled, which is given in the variable `distance`, rather than air-time.

To find the longest flight, sort `distance` in descending order using `desc()`.

```
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>
```

The longest flight is HA 51, JFK to HNL, which is 4,983 miles.

To find the shortest flight, sort `distance` in ascending order, which is the default sort order, so we don't need to use `desc()`.

```
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>
```

The shortest flight is US 1632, EWR to LGA, which is only 17 miles. This is a flight between two of the New York area airports. However, since this flight is missing a departure time so it either did not actually fly or there is a problem with the data.

However, another reasonable interpretation of "longest" and "shortest" is in terms of time, which could similarly be found by sorting by `air_time`. The shortest flights in terms of air time are

```
arrange(flights, desc(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     3     17    1337      1335      2    1937
#> 2 2013     2     6     853       900     -7    1542
#> 3 2013     3     15    1001      1000      1    1551
#> 4 2013     3     17    1006      1000      6    1607
#> 5 2013     3     16    1001      1000      1    1544
#> 6 2013     2     5     900       900      0    1555
#> # ... 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>
```

and the longest are

```
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>
```

5.4 Select columns with `select()`

Exercise 5.4.1

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

A few ways include:

- Specifying all the variables with unquoted variable names.

```
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
```

- Specifying all the variables as strings.

```
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
```

- Specifying the column numbers of the variables.

```
select(flights, 4, 5, 6, 9)
#> # A tibble: 336,776 x 4
```

```
#>   dep_time sched_dep_time dep_delay arr_delay
#>   <int>          <int>    <dbl>    <dbl>
#> 1    517           515      2       11
#> 2    533           529      4       20
#> 3    542           540      2       33
#> 4    544           545     -1      -18
#> 5    554           600     -6      -25
#> 6    554           558     -4       12
#> # ... with 3.368e+05 more rows
```

This works, but is not good practice for two reasons. First, the column location of variables may change, resulting in code that may continue to run without error, but produce the wrong answer. Second code is obfuscated, since it is not clear from the code which variables are being selected. What variable does column 5 correspond to? I just wrote the code, and I've already forgotten.

- Specifying the names of the variables with character vector and `one_of()`.

```
select(flights, one_of(c("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
```

This is useful because the names of the variables can be stored in a variable and passed to `one_of()`.

```
variables <- c("dep_time", "dep_delay", "arr_time", "arr_delay")
select(flights, one_of(variables))
#> # 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
```

- Selecting the variables by matching the start of their names using `starts_with()`.

```
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
```

- Selecting the variables using `matches()` and regular expressions, which are discussed in the Strings chapter.

```
select(flights, matches("^dep|arr)_time|delay$"))
#> # A tibble: 336,776 x 4
#>   dep_time  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
```

Some things that **don't** work are

- Matching the ends of their names using `ends_with()` since this will incorrectly include other variables. For example,

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

- Matching the names using `contains()` since there is not a pattern that can include all these variables without incorrectly including others.

```
select(flights, contains("_time"), contains("arr_"))
#> # A tibble: 336,776 x 6
#>   dep_time sched_dep_time arr_time sched_arr_time air_time arr_delay
#>   <int>        <int>     <int>        <int>     <dbl>      <dbl>
#> 1    517         515      830         819      227       11
#> 2    533         529      850         830      227       20
#> 3    542         540      923         850      160       33
#> 4    544         545     1004        1022      183      -18
#> 5    554         600      812         837      116      -25
#> 6    554         558      740         728      150       12
#> # ... with 3.368e+05 more rows
```

Exercise 5.4.2

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

The `select()` call ignores the duplication. Any duplicated variables are only included once, in the first location they appear. The `select()` function does not raise an error or warning or print any message if there are duplicated variables.

```
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
```

This behavior is useful because it means that we can use `select()` with `everything()` in order to easily change the order of columns without having to specify the names of all the columns.

```
select(flights, arr_delay, everything())
#> # A tibble: 336,776 x 19
#>   arr_delay year month   day dep_time sched_dep_time dep_delay arr_time
#>   <dbl> <int> <int> <int>      <int>      <dbl>      <int>
#> 1     11 2013     1     1     517       515        2     830
#> 2     20 2013     1     1     533       529        4     850
#> 3     33 2013     1     1     542       540        2     923
#> 4    -18 2013     1     1     544       545       -1    1004
#> 5    -25 2013     1     1     554       600       -6     812
#> 6     12 2013     1     1     554       558       -4     740
#> # ... with 3.368e+05 more rows, and 11 more variables:
#> #   sched_arr_time <int>, carrier <chr>, flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

Exercise 5.4.3

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

The `one_of()` function select variables using a character vector rather than as unquoted variable names. This function is useful because it is easier to programmatically generate character vectors with variable names than to generate unquoted variable names, which are easier to type.

```
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
```

Exercise 5.4.4

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. This may or may not surprise you. If this behavior does not surprise you, that could be why it is the default. Users searching for variable names probably have a better sense of the letters in the variable than their capitalization. A second, technical, reason is that dplyr works with more than R data frames. It can also work with a variety of databases. Some of these database engines have case insensitive column names, so making functions that match variable names case insensitive by default will make the behavior of `select()` consistent regardless of whether the table is stored as an R data frame or in a database.

To change the behavior add the argument `ignore.case = FALSE`.

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

5.5 Add new variables with `mutate()`

Exercise 5.5.1

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, 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. For example, 1504 represents 15:04 (or 3:04 PM), which is

```
15 * 9 + 4
#> [1] 139
```

minutes after midnight. In order to generalize this approach, we need a way to split out the hour digits from the minutes digits. Dividing by 100 and discarding the remainder using the integer division operator, `%/%` gives us the

```
1504 %/% 100
#> [1] 15
```

Instead of `%/%` could also use `/` along with `trunc()` or `floor()`, but `round()` would not work. To get the minutes, instead of discarding the remainder of the division by 100, we only want the remainder. So we use the modulo operator, `%%`, discussed in the Other Useful Functions section.

```
1504 %% 100
#> [1] 4
```

Now, we can combine the hours (multiplied by 60 to convert them to minutes) and minutes to get the number of minutes after midnight.

```
1504 %/ 100 * 60 + 1504 %% 100
#> [1] 904
```

There is one remaining issue. Midnight is represented by 2400, which would correspond to 1440 minutes since midnight, but it should correspond to 0. After converting all the times to minutes after midnight, `x %% 1440` will convert 1440 to zero, while keeping all the other times the same.

Putting it all together, the following code creates a new data frame `flights_times` with the new columns, `dep_time_mins` and `sched_dep_time_mins` which convert `dep_time` and `sched_dep_time`, respectively, to minutes since midnight.

```
flights_times <- mutate(flights,
  dep_time_mins = (dep_time %/ 100 * 60 + dep_time %% 100) %% 1440,
  sched_dep_time_mins = (sched_dep_time %/ 100 * 60 +
    sched_dep_time %% 100) %% 1440
)
# view only relevant columns
select(flights_times, 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
```

Looking ahead to the Functions chapter, this is precisely the sort of situation in which it would make sense to write a function to avoid copying and pasting code. We could define a function `time2mins()`, which converts a vector of times in from the format used in `flights` to minutes since midnight.

```
time2mins <- function(x) {
  (x %/ 100 * 60 + x %% 100) %% 1440
}
```

Using `time2mins`, the previous code simplifies to the following.

```
flights_times <- mutate(flights,
  dep_time_mins = time2mins(dep_time),
  sched_dep_time_mins = time2mins(sched_dep_time))
# show only the relevant columns
select(flights_times, 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
```

Exercise 5.5.2

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?

I would expect that the air time is the difference between the arrival and departure times, `air_time = arr_time - dep_time`.

To check this, I need to first convert the times to a form more amenable to arithmetic using the same calculations as in the previous exercise.

```
flights_airtime <-
  mutate(flights,
    dep_time_min = (dep_time %/% 100 * 60 + dep_time %% 100) %% 1440,
    arr_time_min = (arr_time %/% 100 * 60 + arr_time %% 100) %% 1440,
    air_time_diff = air_time - arr_time + dep_time)
```

Is my expectation correct? Does `air_time = arr_time - dep_time`?

```
filter(flights_airtime, air_time_diff != 0)
#> # A tibble: 326,128 x 22
#>   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 3.261e+05 more rows, and 15 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_time_min <dbl>, arr_time_min <dbl>, air_time_diff <dbl>
```

No. So why not? Apart from data error, I can think of two reasons why `air_time` may not equal `arr_time - dep_time`.

1. The flight passes midnight, so `arr_time < dep_time`. This will result in times that are off by 24 hours (1,440 minutes). incorrect negative flight times.
2. The flight crosses time zones, and the total air time will be off by hours (multiples of 60). Additionally, all these discrepancies should be positive. All the flights in the `nycflights13` data departed from New York City and are domestic (within the US), meaning that flights will all be to the same or more westerly time zones.

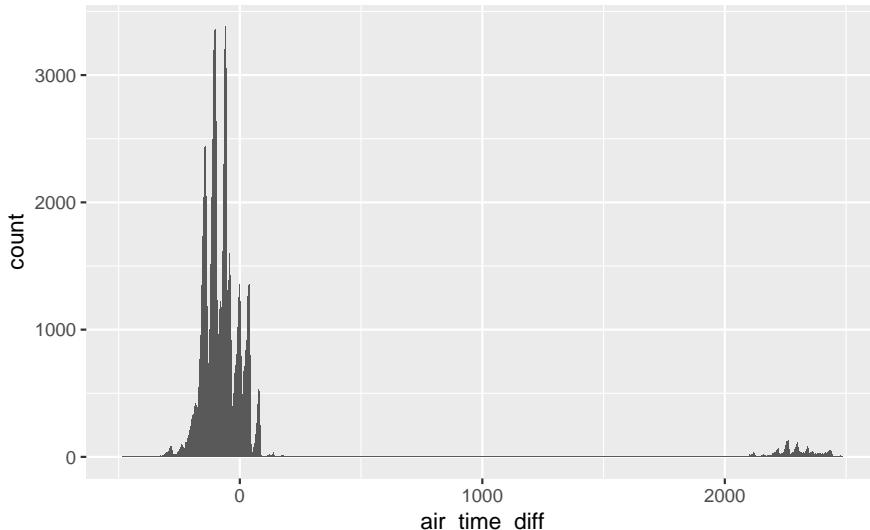
Both of these explanations have clear patterns that I would expect to see if they were true. In particular, in both cases all differences should be divisible by 60. However, there are many flights in which the difference between `arr_time` and `dest_time` is not divisible by 60.

```
filter(flights_airtime, air_time_diff %% 60 == 0)
#> # A tibble: 6,823 x 22
#>   year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>     <int>           <int>     <dbl>     <int>
#> 1 2013     1     1      608             600       8     807
#> 2 2013     1     1      746             746       0    1119
#> 3 2013     1     1      857             900      -3    1516
#> 4 2013     1     1      903             820      43    1045
```

```
#> 5 2013 1 1 908 910 -2 1020
#> 6 2013 1 1 1158 1200 -2 1256
#> # ... with 6,817 more rows, and 15 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_time_min <dbl>,
#> # arr_time_min <dbl>, air_time_diff <dbl>
```

I'll try plotting the data to see if that is more informative.

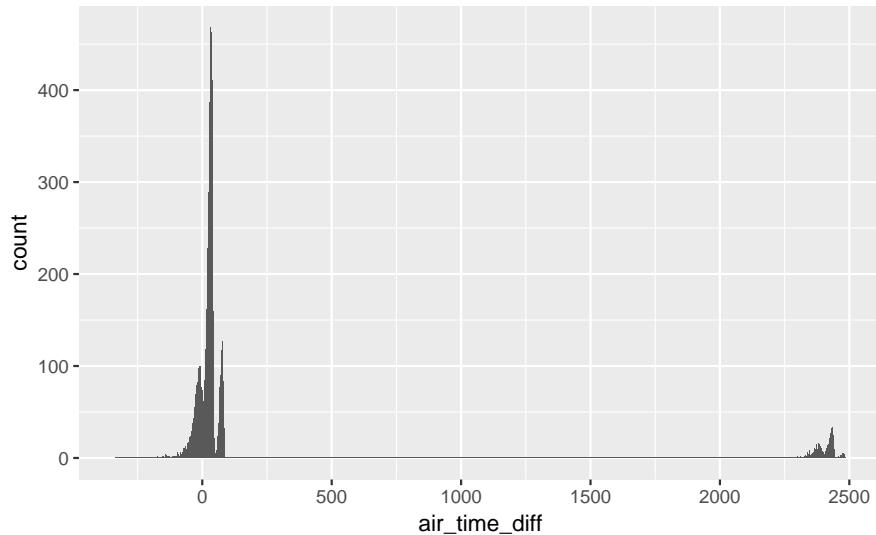
```
ggplot(flights_airtime, aes(x = air_time_diff)) +
  geom_histogram(binwidth = 1)
#> Warning: Removed 9430 rows containing non-finite values (stat_bin).
```



The distribution is bimodal, which with one mode comprising discrepancies up to several hours, suggesting the time-zone problem, and a second node around 24 hours, suggesting the overnight flights. However, in both cases, the discrepancies are not all at values divisible by 60.

I can also confirm my guess about time zones by looking at discrepancies from flights to a destinations in another air zone (or even all flights to different time zones using the time zone of the airport from the `airports` data frame). In this case, I'll look at the distribution of the discrepancies for flights to Los Angeles (LAX).

```
ggplot(filter(flights_airtime, dest == "LAX"), aes(x = air_time_diff)) +
  geom_histogram(binwidth = 1)
#> Warning: Removed 148 rows containing non-finite values (stat_bin).
```



So what else might be going on? There seem to be too many “problems” for this to be a data issue, so I’m probably missing something. So I’ll reread the documentation to make sure that I understand the definitions of `arr_time`, `dep_time`, and `air_time`. The documentation contains a link to the source of the `flights` data, https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236. Reading the page at that link, I see that there are some other variables: `TaxiIn`, `TaxiOff`, `WheelsIn`, `WheelsOff` that are not included in `flights`. The `air_time` variable refers to flight time, which must be defined as the time between wheels off (take-off) and wheels in (landing). Thus `air_time` does not include the time spent on the runway taxiing to and from gates. With this new understanding of the data, I now know that the relationship between `air_time`, `arr_time`, and `dep_time` is `air_time <= arr_time - dep_time` once `arr_time` and `dep_time` are corrected for differing time zones and dates.

Exercise 5.5.3

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

I would expect the departure delay (`dep_time`) to be equal to the difference between scheduled departure time (`sched_dep_time`), and actual departure time (`dep_time`), $\text{dep_time} - \text{sched_dep_time} = \text{dep_delay}$.

As with the previous question, the first step is to convert all times to the number of minutes since midnight. The column, `dep_delay_diff` will be the difference between `dep_delay` and departure delay calculated from the scheduled and actual departure times.

```
flights_deptime <-  
  mutate(flights,  
    dep_time_min = (dep_time %/% 100 * 60 + dep_time %% 100) %% 1440,  
    sched_dep_time_min = (sched_dep_time %/% 100 * 60 +  
                          sched_dep_time %% 100) %% 1440,  
    dep_delay_diff = dep_delay - dep_time_min + sched_dep_time_min)
```

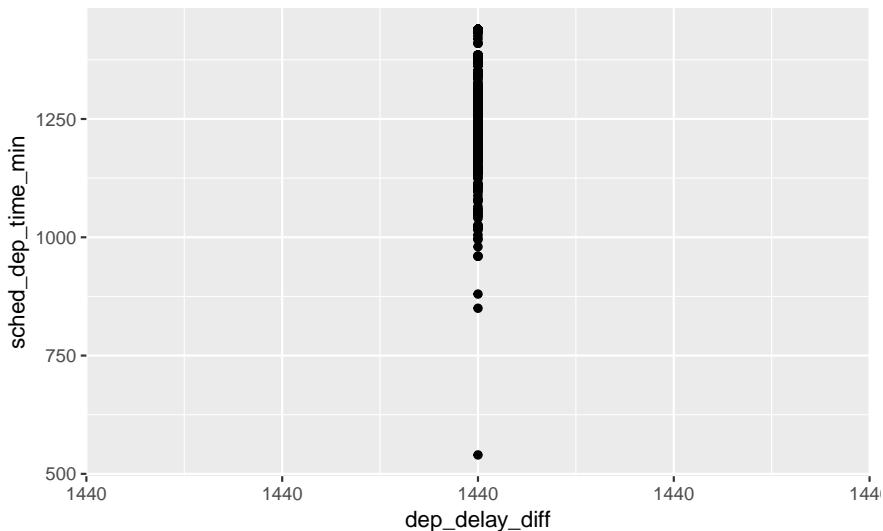
Does `dep_delay_diff` equal zero for all rows?

```
filter(flights_deptime, dep_delay_diff != 0)  
#> # A tibble: 1,236 x 22  
#>   year month   day dep_time sched_dep_time dep_delay arr_time  
#>   <int> <int> <int>     <int>           <int>      <dbl>     <int>  
#> 1  2013     1     1      848          1835       853     1001  
#> 2  2013     1     2       42          2359       43      518
```

```
#> 3 2013 1 2 126 2250 156 233
#> 4 2013 1 3 32 2359 33 504
#> 5 2013 1 3 50 2145 185 203
#> 6 2013 1 3 235 2359 156 700
#> # ... with 1,230 more rows, and 15 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_time_min <dbl>,
#> # sched_dep_time_min <dbl>, dep_delay_diff <dbl>
```

No. Unlike the last question, time zones are not an issue since we are only considering departure times.¹ However, the discrepancies could be because a flight was scheduled to depart before midnight, but was delayed after midnight. All of these discrepancies are exactly equal to 1440 (24 hours), and the flights with these discrepancies were scheduled to depart later in the day.

```
ggplot(filter(flights_deptime, dep_delay_diff > 0),
       aes(y = sched_dep_time_min, x = dep_delay_diff)) +
  geom_point()
```



Thus the only cases in which the departure delay is not equal to the difference in scheduled departure and actual departure times is due to a quirk in how these columns were stored.

Exercise 5.5.4

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 have the same value and are the most delayed, we would say they are tied for first, not tied for third or second.

```
flights_delayed <- mutate(flights, dep_delay_rank = min_rank(-dep_delay))
flights_delayed <- filter(flights_delayed, dep_delay_rank <= 20)
arrange(flights_delayed, dep_delay_rank)
#> # A tibble: 20 x 20
```

¹Except for flights daylight savings started (March 10) or ended (November 3). But since daylight savings goes into effect at 02:00, and generally flights are not scheduled to depart between midnight and 2 am, the only flights which would be scheduled to depart in Eastern Daylight Savings Time (Eastern Standard Time) time but departed in Eastern Standard Time (Daylight Savings Time), would have been scheduled before midnight, meaning they were delayed across days.

```
#>   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 14 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>
```

Exercise 5.5.5

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
```

Exercise 5.5.6

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:

```
x <- seq(-3, 7, by = 1 / 2)
cos(pi * x)
#> [1] -1.00e+00  3.06e-16  1.00e+00 -1.84e-16 -1.00e+00  6.12e-17  1.00e+00
#> [8]  6.12e-17 -1.00e+00 -1.84e-16  1.00e+00  3.06e-16 -1.00e+00 -4.29e-16
#> [15] 1.00e+00  5.51e-16 -1.00e+00 -2.45e-15  1.00e+00 -9.80e-16 -1.00e+00
cos(pi * x)
#> [1] -1.00e+00  3.06e-16  1.00e+00 -1.84e-16 -1.00e+00  6.12e-17  1.00e+00
#> [8]  6.12e-17 -1.00e+00 -1.84e-16  1.00e+00  3.06e-16 -1.00e+00 -4.29e-16
#> [15] 1.00e+00  5.51e-16 -1.00e+00 -2.45e-15  1.00e+00 -9.80e-16 -1.00e+00
tan(pi * x)
#> [1] 3.67e-16 -3.27e+15  2.45e-16 -5.44e+15  1.22e-16 -1.63e+16  0.00e+00
#> [8] 1.63e+16 -1.22e-16  5.44e+15 -2.45e-16  3.27e+15 -3.67e-16  2.33e+15
#> [15] -4.90e-16  1.81e+15 -6.12e-16  4.08e+14 -7.35e-16 -1.02e+15 -8.57e-16
```

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

```
cospi(x)
#> [1] -1 0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1
cos(x)
#> [1] -0.9900 -0.8011 -0.4161  0.0707  0.5403  0.8776  1.0000  0.8776
#> [9]  0.5403  0.0707 -0.4161 -0.8011 -0.9900 -0.9365 -0.6536 -0.2108
#> [17]  0.2837  0.7087  0.9602  0.9766  0.7539
tan(x)
#> [1]  0.143  0.747  2.185 -14.101 -1.557 -0.546  0.000  0.546
#> [9]  1.557 14.101 -2.185 -0.747 -0.143  0.375  1.158  4.637
#> [17] -3.381 -0.996 -0.291  0.220  0.871
```

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

```
x <- seq(-1, 1, by = 1 / 4)
acos(x)
#> [1] 3.142 2.419 2.094 1.823 1.571 1.318 1.047 0.723 0.000
asin(x)
#> [1] -1.571 -0.848 -0.524 -0.253  0.000  0.253  0.524  0.848  1.571
atan(x)
#> [1] -0.785 -0.644 -0.464 -0.245  0.000  0.245  0.464  0.644  0.785
```

The function `atan2()` is the angle between the x-axis and 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()`

Exercise 5.6.1

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?

What this question gets at is a fundamental question of data analysis: the cost function. As analysts, the reason we are interested in flight delay because it is costly to passengers. But it is worth thinking carefully about how it is costly and use that information in ranking and measuring these scenarios.

In many scenarios, arrival delay is more important. Presumably being late on arriving is more costly to the passenger since it could disrupt the next stages of their travel, such as connecting flights or meetings.

If the departure is delayed without affecting the arrival time and the passenger arrived at the same time, this delay will not affect future plans nor does it affect the total time spent traveling. The delay could be a positive, if less time is spent on the airplane itself, or a negative, if that extra time is spent on the plane in the runway.

Variation in arrival time is worse than consistency. If a flight is always 30 minutes late and that delay is known, then it is as if the arrival time is that delayed time. The traveler could easily plan for this. If the delay of the flight is more variable, then it is harder for the traveler to plan for it.

TODO (Add a better explanation and some examples)

Exercise 5.6.2

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()`).

The data frame `not_canceled` is defined in the chapter as,

```
not_canceled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))
```

`Count` will group a dataset on the given variable and then determine the number of instances within each group. This can be done by first grouping by the given variable, and then finding the number of observations in each group. The number of observations in each group can be found by calling the `length()` function on any variable. To make the result match `count()`, the value should go in a new column `n`.

```
not_canceled %>%
  group_by(dest) %>%
  summarise(n = length(dest))
#> # A tibble: 104 x 2
#>   dest      n
#>   <chr> <int>
#> 1 ABQ     254
#> 2 ACK     264
#> 3 ALB     418
#> 4 ANC      8
#> 5 ATL    16837
#> 6 AUS     2411
#> # ... with 98 more rows
```

A more concise way to get the number of observations in a data frame, or a group, is the function `n()`,

```
not_canceled %>%
  group_by(dest) %>%
  summarise(n = n())
#> # A tibble: 104 x 2
#>   dest      n
#>   <chr> <int>
#> 1 ABQ     254
#> 2 ACK     264
#> 3 ALB     418
#> 4 ANC      8
#> 5 ATL    16837
#> 6 AUS     2411
#> # ... with 98 more rows
```

For a weighted count, take the sum of the weight variable in each group.

```
not_canceled %>%
  group_by(tailnum) %>%
  summarise(n = sum(distance))
#> # A tibble: 4,037 x 2
#>   tailnum      n
#>   <chr> <dbl>
#> 1 D942DN    3418
#> 2 NOEGMQ  239143
#> 3 N10156  109664
#> 4 N102UW   25722
```

```
#> 5 N103US 24619
#> 6 N104UW 24616
#> # ... with 4,031 more rows
```

Alternatively, we could have used `group_by()` followed by `tally()`, since `count()` itself is a shortcut for calling `group_by()` then `tally()`,

```
not_canceled %>%
  group_by(tailnum) %>%
  tally()
#> # A tibble: 4,037 x 2
#>   tailnum     n
#>   <chr>    <int>
#> 1 D942DN      4
#> 2 NOEGMQ    352
#> 3 N10156    145
#> 4 N102UW     48
#> 5 N103US    46
#> 6 N104UW    46
#> # ... with 4,031 more rows
```

and

```
not_canceled %>%
  group_by(tailnum) %>%
  tally(distance)
#> # A tibble: 4,037 x 2
#>   tailnum     n
#>   <chr>    <dbl>
#> 1 D942DN    3418
#> 2 NOEGMQ  239143
#> 3 N10156  109664
#> 4 N102UW   25722
#> 5 N103US  24619
#> 6 N104UW  24616
#> # ... with 4,031 more rows
```

Exercise 5.6.3

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 never departs, then it won't arrive. A flight could also depart and not arrive if it crashes, or if it is redirected and lands in an airport other than its intended destination.

The more important column is `arr_delay`, which indicates the amount of delay in arrival.

```
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
```

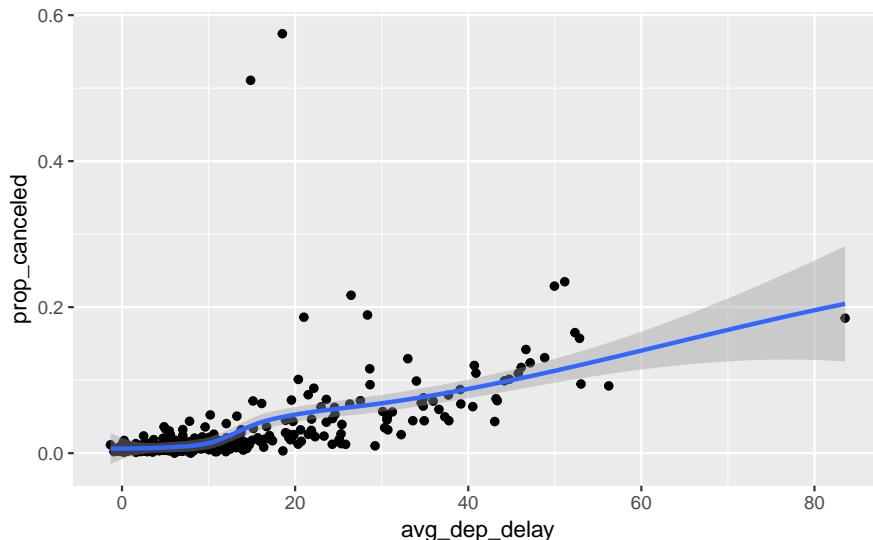
```
#> 4    1807    2251    2103    29    NA
#> 5    1939     29    2151    59    NA
#> 6    1952    2358    2207    22    NA
#> # ... with 1,169 more rows
```

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?

Exercise 5.6.4

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' and formula 'y ~ x'
```



Exercise 5.6.5

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
```

What airline corresponds to the "F9" carrier code?

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

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.

Exercise 5.6.6

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 run `count()` followed by `arrange()`.

5.7 Grouped mutates (and filters)

Exercise 5.7.1

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.

Exercise 5.7.2

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

The question does not define the on-time record. I will use the proportion of flights not delayed or canceled. This metric does not differentiate between the amount of delay, but has the benefit of easily incorporating canceled flights.

```
flights %>%
  # unknown why flights have sched_arr_time, arr_time but missing arr_delay.
  filter(!is.na(arr_delay)) %>%
  mutate(canceled = is.na(arr_time),
        late = !canceled & arr_delay > 0) %>%
```

```

group_by(tailnum) %>%
  summarise(on_time = mean(!late)) %>%
  filter(min_rank(on_time) <= 1)
#> # A tibble: 104 x 2
#>   tailnum on_time
#>   <chr>     <dbl>
#> 1 N121DE      0
#> 2 N136DL      0
#> 3 N143DA      0
#> 4 N17627      0
#> 5 N240AT      0
#> 6 N26906      0
#> # ... with 98 more rows

```

However, there are many planes that have *never* flown an on-time flight.

Another alternative is to rank planes by the mean of minutes delayed.

```

flights %>%
  group_by(tailnum) %>%
  summarise(arr_delay = mean(arr_delay)) %>%
  filter(min_rank(desc(arr_delay)) <= 1)
#> # A tibble: 1 x 2
#>   tailnum arr_delay
#>   <chr>     <dbl>
#> 1 N844MH     320

```

Exercise 5.7.3

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)) %>%
  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

```

Exercise 5.7.4

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(arr_delay_total = sum(arr_delay),
        arr_delay_prop = arr_delay / arr_delay_total)
#> # 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>,
#> #   arr_delay_total <dbl>, arr_delay_prop <dbl>

```

The key to answering this question is when calculating the total delay and proportion of delay we only consider only delayed flights, and ignore on-time or early flights.

Exercise 5.7.5

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.

This calculates the departure delay of the preceding flight from the same airport.

```

lagged_delays <- flights %>%
  arrange(origin, year, month, day, dep_time) %>%
  group_by(origin) %>%
  mutate(dep_delay_lag = lag(dep_delay)) %>%
  filter(!is.na(dep_delay), !is.na(dep_delay_lag))

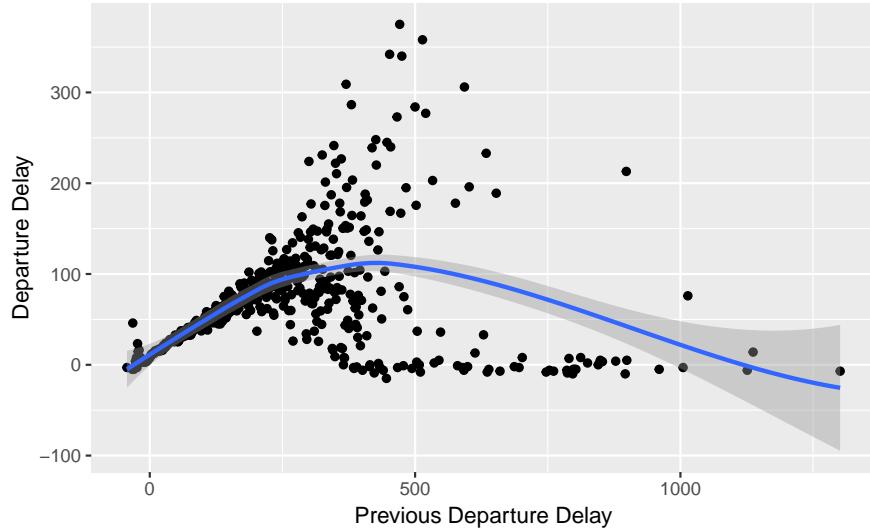
```

This plots the relationship between the mean delay of a flight for all values of the previous flight.

```

lagged_delays %>%
  group_by(dep_delay_lag) %>%
  summarise(dep_delay_mean = mean(dep_delay)) %>%
  ggplot(aes(y = dep_delay_mean, x = dep_delay_lag)) +
  geom_point() +
  geom_smooth() +
  labs(y = "Departure Delay", x = "Previous Departure Delay")

```



We can summarize this relationship by the average difference in delays:

```
lagged_delays %>%
  summarise(delay_diff = mean(dep_delay - dep_delay_lag), na.rm = TRUE)
#> # A tibble: 3 x 3
#>   origin delay_diff na.rm
#>   <chr>     <dbl> <lgl>
#> 1 EWR      0.148  TRUE
#> 2 JFK     -0.0319 TRUE
#> 3 LGA      0.209  TRUE
```

Exercise 5.7.6

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 of a flight relative to the shortest flight to that destination. Which flights were most delayed in the air?

When calculating this answer we should only compare flights within the same origin, destination pair.

A common approach to finding unusual observations would be to calculate the z-score of observations each flight.

```
flights_with_zscore <- flights %>%
  filter(!is.na(air_time)) %>%
  group_by(dest, origin) %>%
  mutate(air_time_mean = mean(air_time),
        air_time_sd = sd(air_time),
        n = n()) %>%
  ungroup() %>%
  mutate(z_score = (air_time - air_time_mean) / air_time_sd)
```

Possible unusual flights are the Lets print out the 10 flights with the largest

```
flights_with_zscore %>%
  arrange(desc(abs(z_score))) %>%
  select() %>%
  print(n = 15)
#> # A tibble: 327,346 x 0
```

Now that we've identified potentially bad observations, we would like to distinguish between the real problems and

One potential issue with the way that we calculated z-scores is that the mean and standard deviation used to calculate it include the unusual observations that we are looking for. Since the mean and standard deviation are sensitive to outliers, that means that an outlier could affect the mean and standard deviation calculations enough that it does not look like one. We would want to calculate the z-score of each observation using the mean and standard deviation based on all other flights to that origin and destination. This will be more of an issue if the number of observations is small. Thankfully, there are easy methods to update the mean and variance by removing an observation, but for now, we won't use them.²

Another way to improve this calculation is to use the same method used in box plots (see `geom_boxplot()`) to screen outliers. That method uses the median and inter-quartile range, and thus is less sensitive to outliers. Adjust the previous code and see if it makes a difference.

All of these answers have relied on the distribution of comparable observations (flights from the same origin to the same destination) to flag unusual observations. Apart from our knowledge that flights from the same origin to the same destination should have similar air times, we have not used any domain specific knowledge. But actually know much more about this problem. We know that aircraft have maximum speeds. So could use the time and distance of each flight to calculate the average speed of each flight and find any clearly impossibly fast flights.

Exercise 5.7.7

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

To restate this question, we are asked to rank airlines by the number of destinations that they fly to, considering only those airports that are flown to by two or more airlines.

We will calculate this ranking in two parts. First, find all airports serviced by two or more carriers.

```
dest_2carriers <- flights %>%
  # keep only unique carrier,dest pairs
  select(dest, carrier) %>%
  group_by(dest, carrier) %>%
  filter(row_number() == 1) %>%
  # count carriers by destination
  group_by(dest) %>%
  mutate(n_carrier = n_distinct(carrier)) %>%
  filter(n_carrier >= 2)
```

Second, rank carriers by the number of these destinations that they service.

```
carriers_by_dest <- dest_2carriers %>%
  group_by(carrier) %>%
  summarise(n_dest = n()) %>%
  arrange(desc(n_dest))
head(carriers_by_dest)
#> # A tibble: 6 x 2
#>   carrier n_dest
#>   <chr>    <int>
#> 1 EV        51
#> 2 9E        48
#> 3 UA        42
```

²In most interesting data analysis questions, no answer ever “right”. With infinite time and money, an analysis could almost always improve their answer with more data or better methods. The difficulty in real life is finding the quickest, simplest method that works “good enough”.

```
#> 4 DL      39
#> 5 B6      35
#> 6 AA      19
```

The carrier "EV" flies to the most destinations , considering only airports flown to by two or more carriers. What is airline does the "EV" carrier code correspond to?

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

Unless you know the airplane industry, it is likely that you don't recognize ExpressJet; I certainly didn't. It is a regional airline that partners with major airlines to fly from hubs (larger airports) to smaller airports. This means that many of the shorter flights of major carriers are actually operated by ExpressJet. This business model explains why ExpressJet services the most destinations.

Exercise 5.7.8

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

```
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
```


Chapter 6

Workflow: scripts

6.1 Running code

No exercises

6.2 RStudio diagnostics

No exercises

6.3 Practice

Exercise 6.3.1

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

Here's the current timeline of @rstudiotips.

Tweets by rstudiotips

Exercise 6.3.2

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

You should read that page, but 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

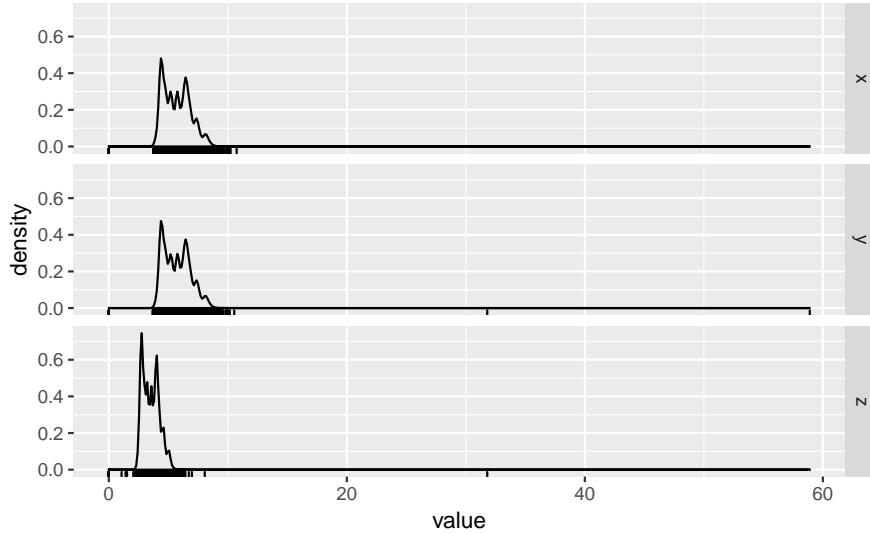
7.3 Variation

Exercise 7.3.1

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)

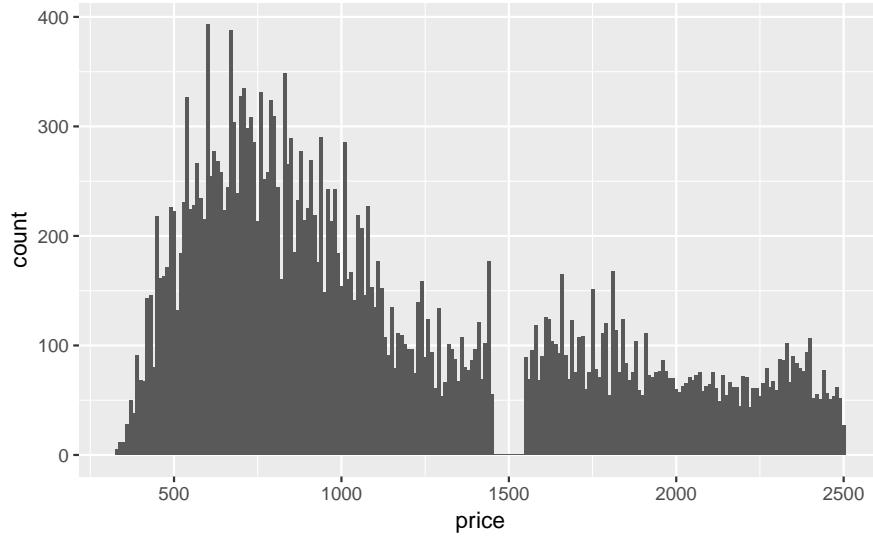
According to the documentation for `diamonds`: x is length, y is width, and z is depth. I don't know if I would have figured that out before; maybe if there was data on the type of cuts.

Exercise 7.3.2

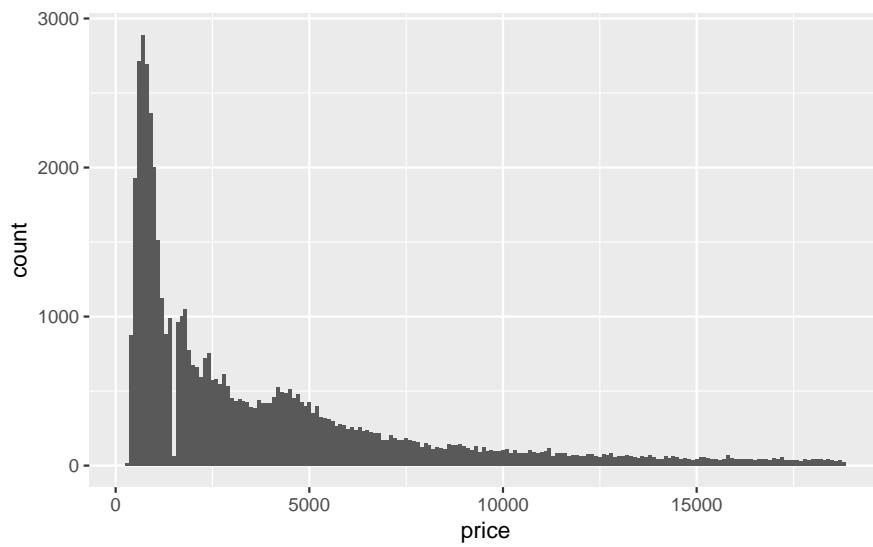
Explore the distribution of price. Do you discover anything unusual or surprising? (Hint: Carefully think about the `binwidth` and make sure you try a wide range of values.)

- 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 \$750.

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

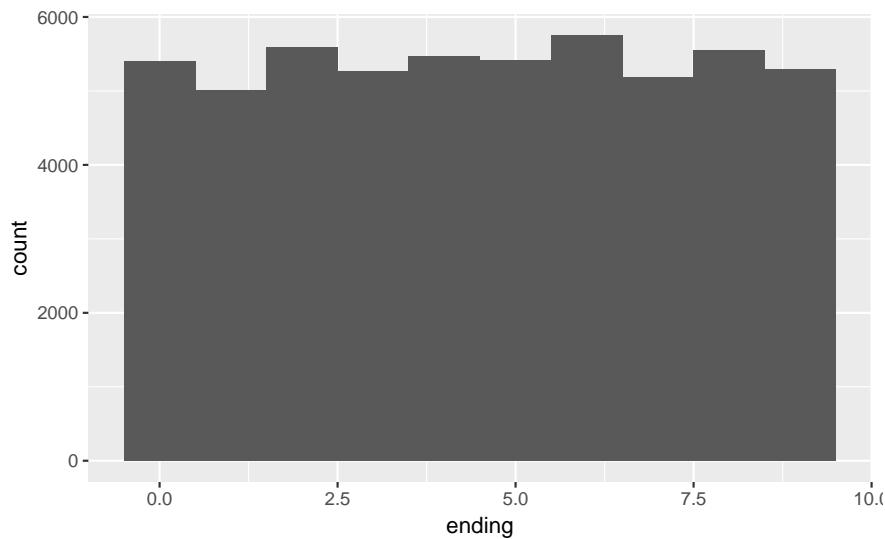


```
ggplot(filter(diamonds), aes(x = price)) +  
  geom_histogram(binwidth = 100, center = 0)
```

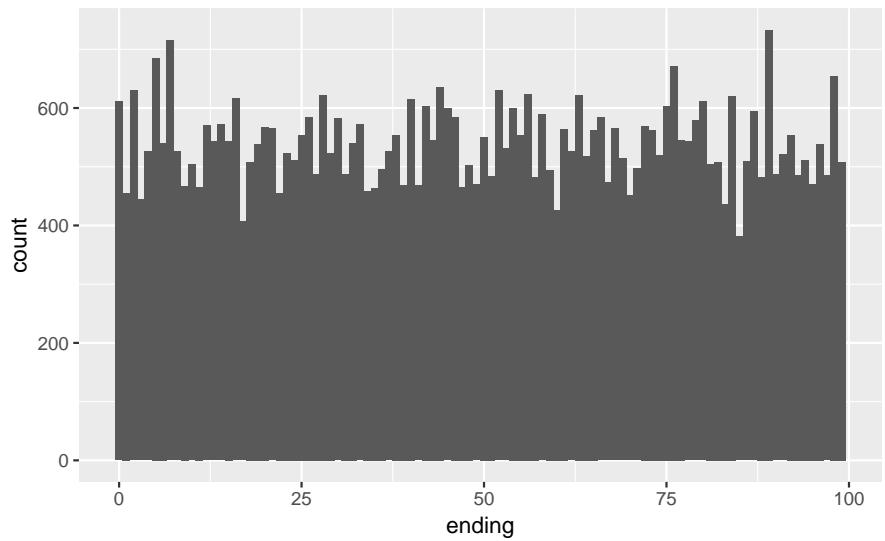


Distribution of last digit

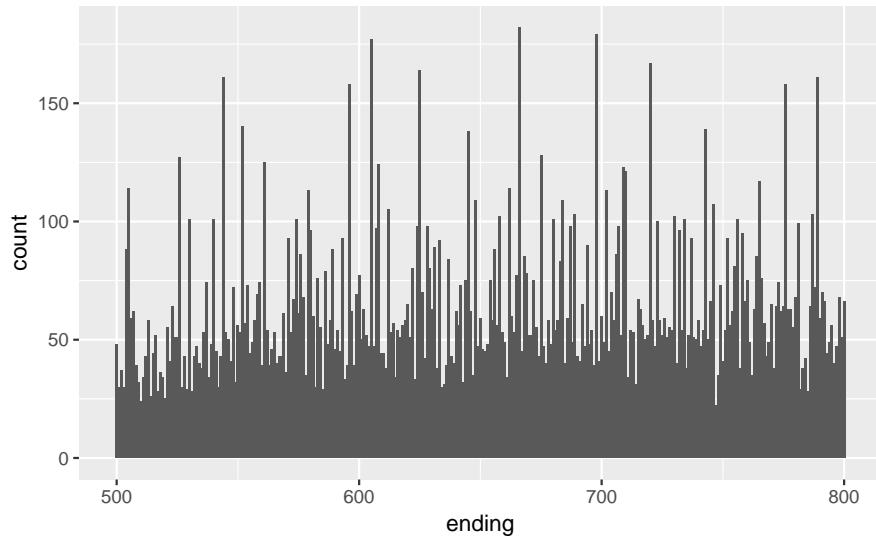
```
diamonds %>%  
  mutate(ending = price %% 10) %>%  
  ggplot(aes(x = ending)) +  
  geom_histogram(binwidth = 1, center = 0) +  
  geom_bar()
```



```
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()
```



Exercise 7.3.3

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 0.99     23
#> 2 1        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 0.9      1485
#> 2 0.91     570
#> 3 0.92     226
#> 4 0.93     142
#> 5 0.94      59
#> 6 0.95      65
#> 7 0.96     103
#> 8 0.97      59
#> 9 0.98      31
```

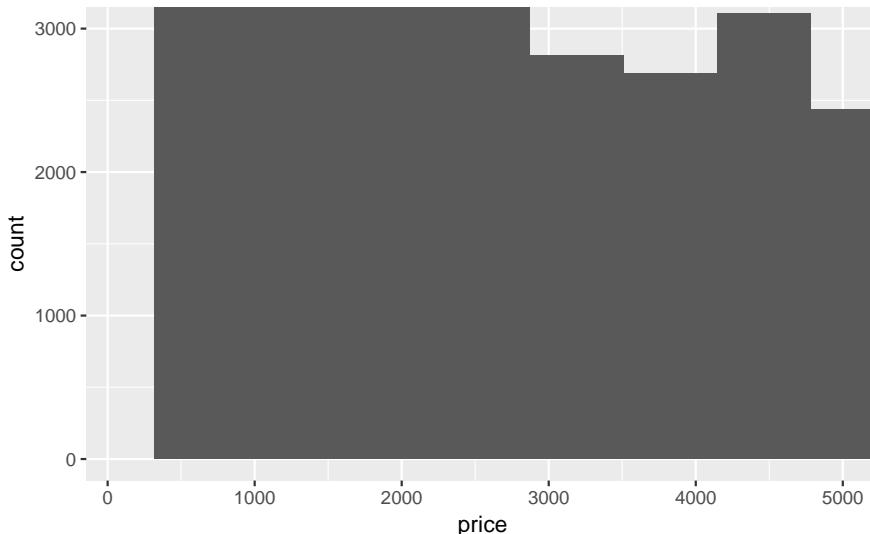
```
#> 10 0.99    23
#> 11 1      1558
#> 12 1.01   2242
#> 13 1.02   883
#> 14 1.03   523
#> 15 1.04   475
#> 16 1.05   361
#> 17 1.06   373
#> 18 1.07   342
#> 19 1.08   246
#> 20 1.09   287
#> 21 1.1    278
```

Exercise 7.3.4

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?

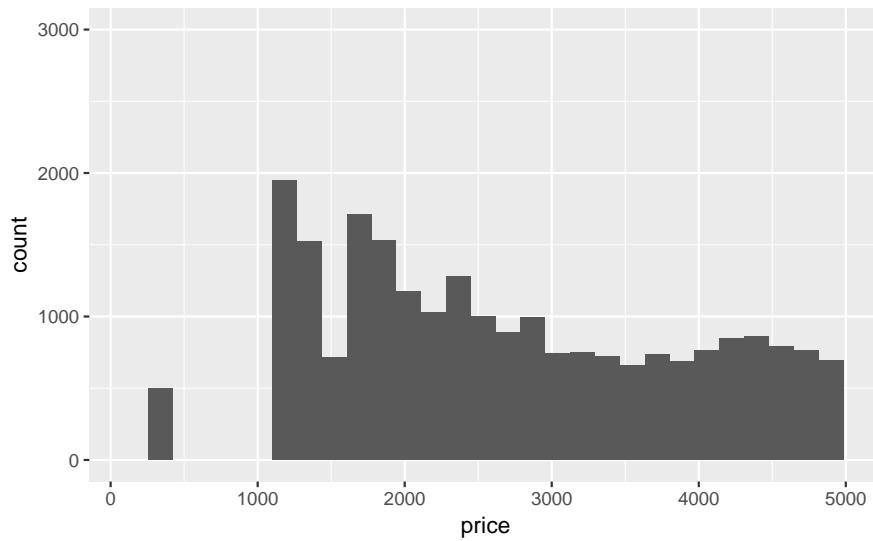
The `coord_cartesian()` function zooms in on the area specified by the limits, after having calculated and drawn the geoms. Since the histogram bins have already been calculated, it is unaffected.

```
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 influence actions before the calculation of the stats related to the histogram. Thus, any values outside the x- and y-limits are dropped before calculating bin widths and counts. This can influence how the histogram looks.

```
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 6 rows containing missing values (geom_bar).
```



7.4 Missing Values

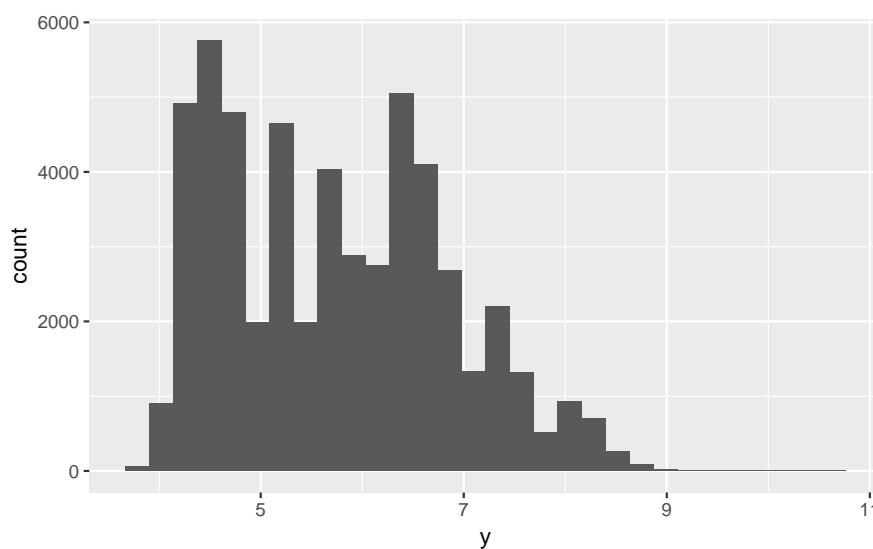
Exercise 7.4.1

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)`

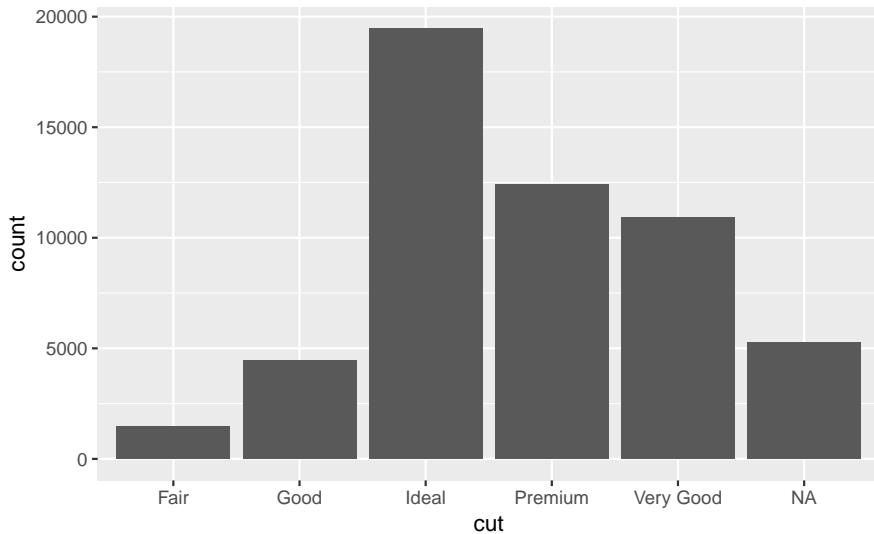
```
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))

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 the `geom_bar()` function, `NA` is treated as another category. The `x` aesthetic in `geom_bar()` requires a discrete (categorical) variable, and missing values act like 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.

Exercise 7.4.2

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

Exercise 7.5.1.1

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

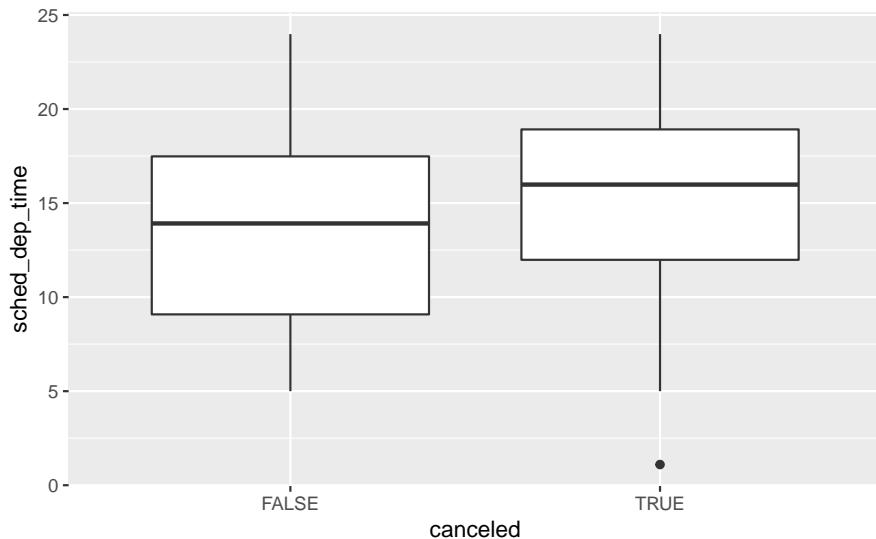
Instead of a `freqplot` use a box-plot

```
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))

```



Exercise 7.5.1.2

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

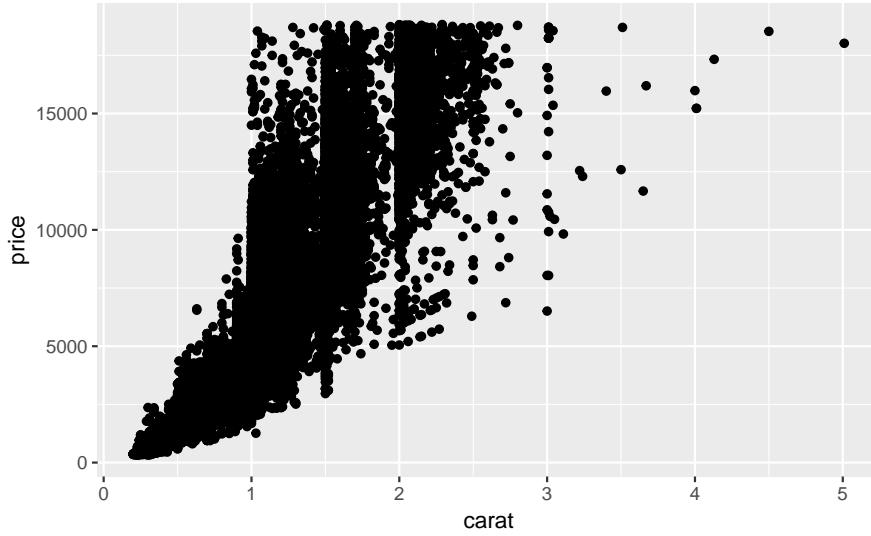
What are the general relationships of each variable with the price of the diamonds? I will consider the variables: `carat`, `clarity`, `color`, and `cut`. I ignore the dimensions of the diamond since `carat` measures size, and thus incorporates most of the information contained in these variables.

Both `price` and `carat` are continuous variables, so I will use scatterplot visualize their relationship.

```

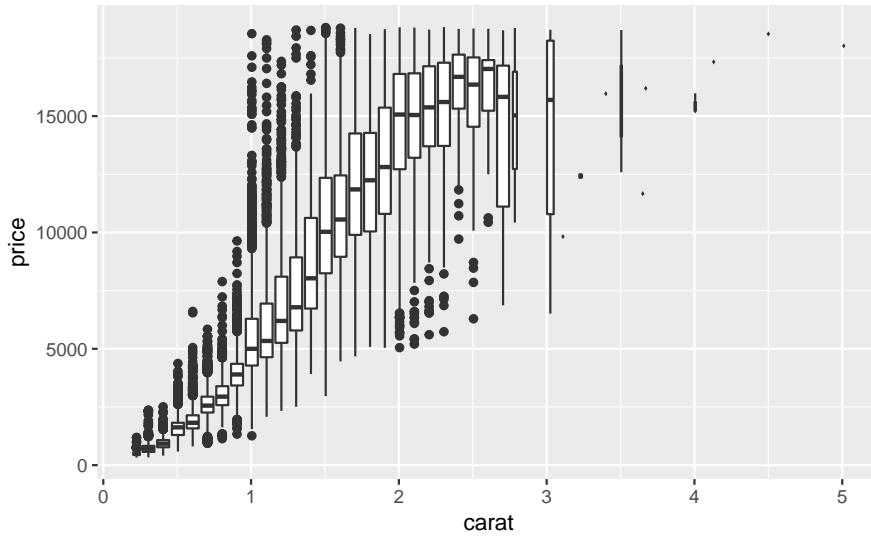
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point()

```



However, since there is a large number of points in the data, I will use a boxplot by binning `carat` (as suggested in the chapter).

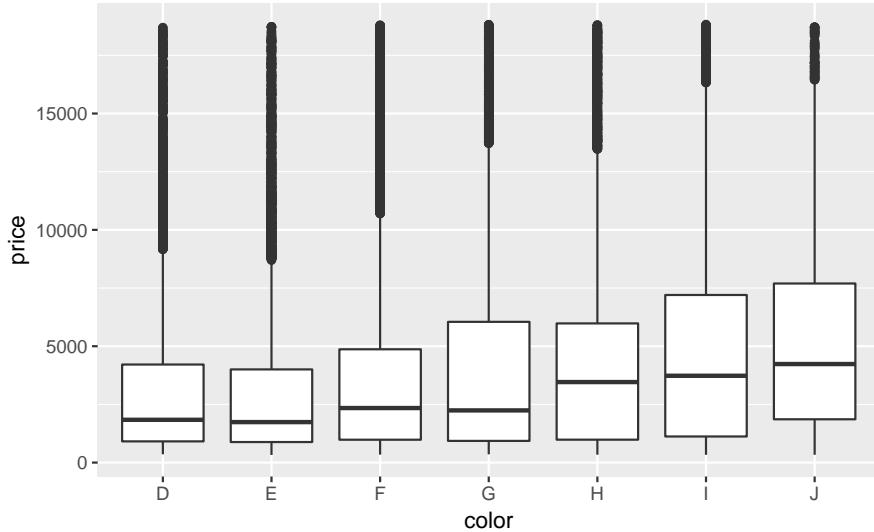
```
ggplot(data = diamonds, mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_width(carat, 0.1)))
```



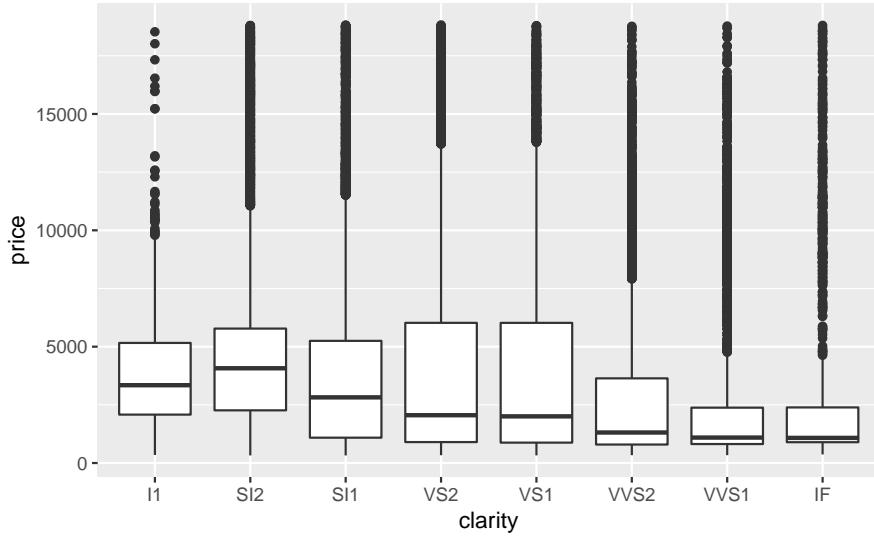
Note that the choice of the binning width is important, as if it were too large it would obscure any relationship, and if it were too small, the values in the bins could be too variable to reveal underlying trends.

The variables `color` and `clarity` are ordered categorical variables. The chapter suggests visualizing a categorical and continuous variable using frequency polygons or boxplots. In this case, I will use a box plot since it will better show a relationship over the variables.

```
ggplot(diamonds, aes(x = color, y = price)) +
  geom_boxplot()
```



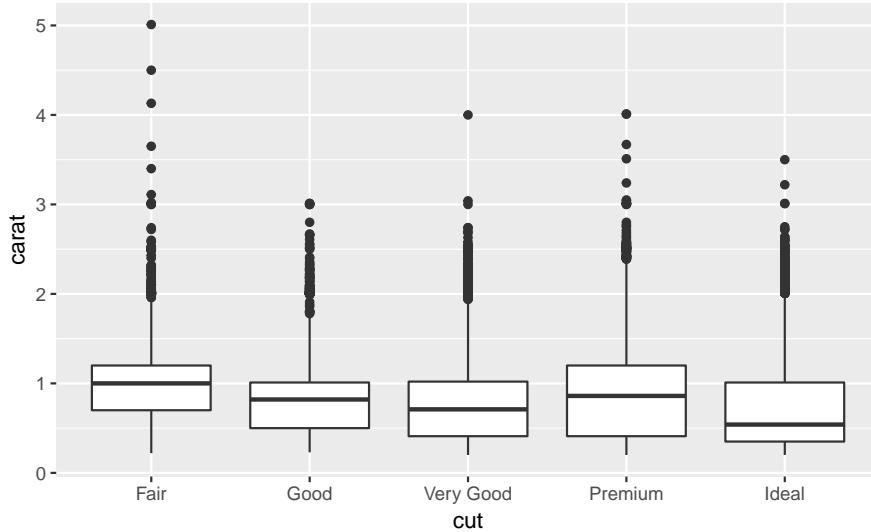
```
ggplot(data = diamonds) +
  geom_boxplot(mapping = aes(x = clarity, y = price))
```



There is a strong relationship between `carat` and `price`. There is a weak positive relationship between `color` and `price`, and, surprisingly, a weak negative relationship between `clarity` and `price`. For both `clarity` and `color`, there is a large amount of variation within each category, which overwhelms the between category trend. Carat is clearly the best predictor of its price.

Now that we have established that carat appears to be the best predictor of price, what is the relationship between it and cut? Since this is an example of a continuous (carat) and categorical (cut) variable, it can be visualized with a box plot.

```
ggplot(diamonds, aes(x = cut, y = carat)) +
  geom_boxplot()
```



There is a lot of variability in the distribution of carat sizes within each cut category. There is a slight negative relationship between carat and cut. Noticeably, the largest carat diamonds have a cut of “Fair” (the lowest).

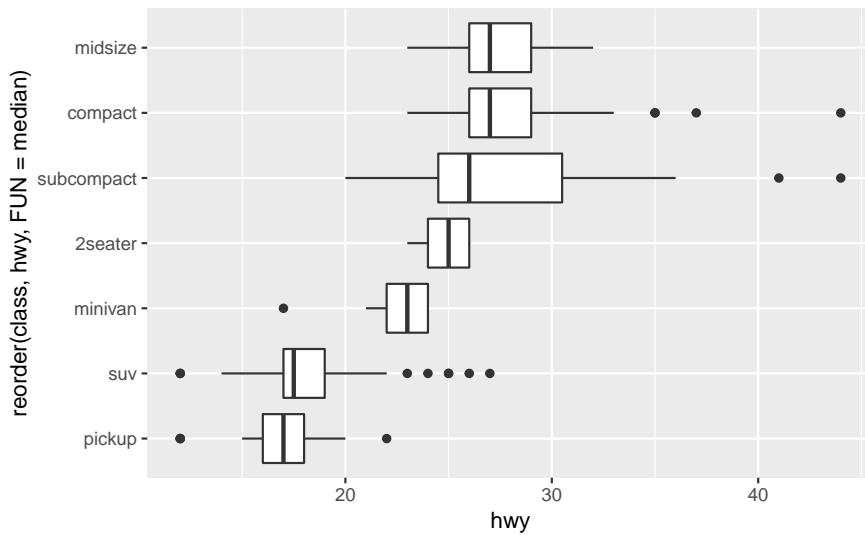
This negative relationship can be due to the way in which diamonds are selected for sale. A larger diamond can be profitably sold with a lower quality cut, while a smaller diamond requires a better cut.

Exercise 7.5.1.3

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

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

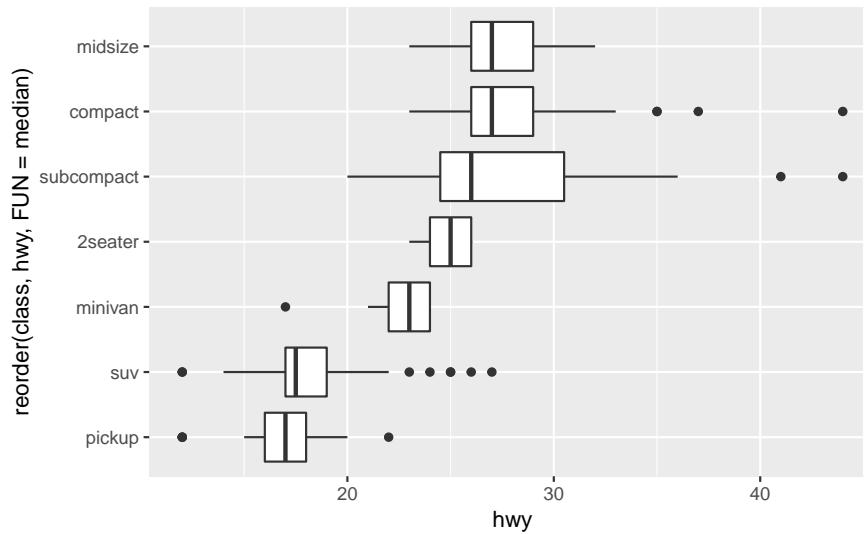
```
ggplot(data = mpg) +
  geom_boxplot(mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) +
  coord_flip()
```



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

```
library("ggstance")

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



Exercise 7.5.1.4

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?

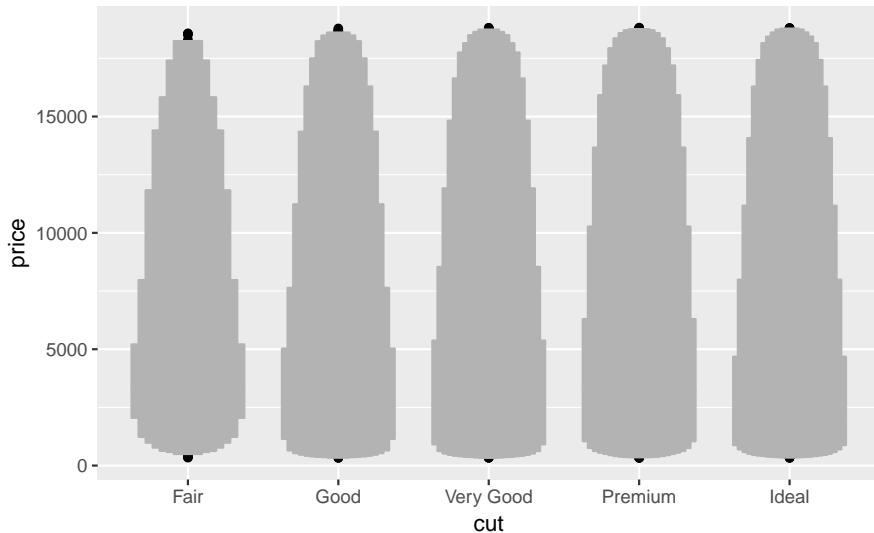
How do you interpret the plots?

Like box-plots, the boxes of the letter-value plot correspond to quantiles. However, they incorporate far more quantiles than box-plots. 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).

```
library("lvplot")

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



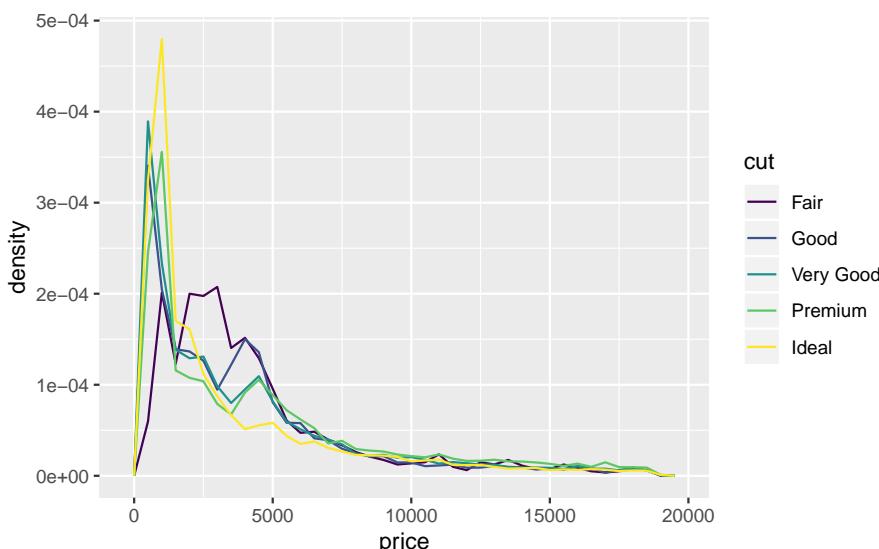
The letter-value plot is described in Hofmann, Wickham, and Kafadar (2017).

Exercise 7.5.1.5

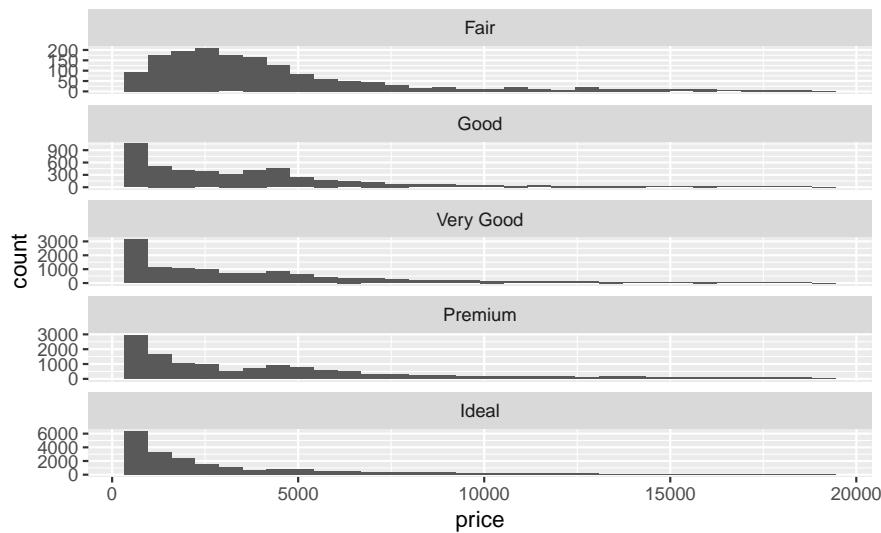
Compare and contrast `geom_violin()` with a faceted `geom_histogram()`, or a colored `geom_freqpoly()`. What are the pros and cons of each method?

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, it is 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.

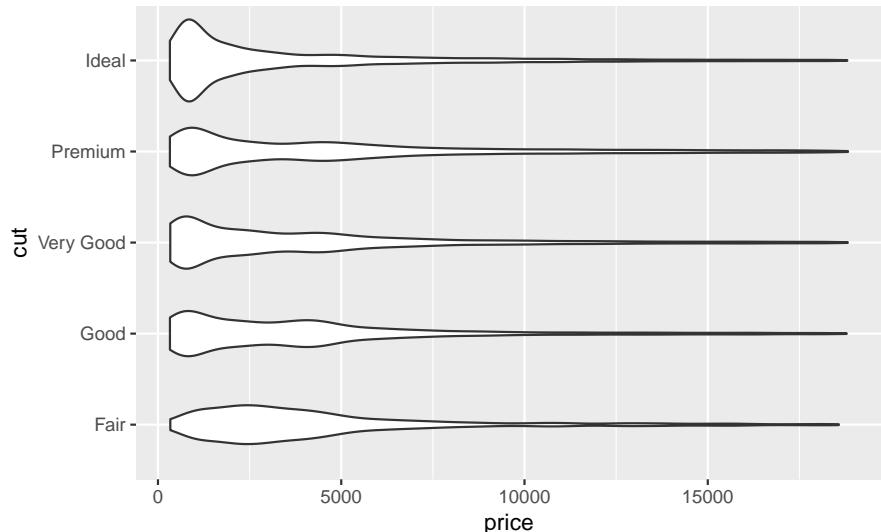
```
ggplot(data = diamonds, mapping = aes(x = price, y = ..density..)) +
  geom_freqpoly(mapping = aes(color = cut), binwidth = 500)
```



```
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 and Nelson (1998).

Exercise 7.5.1.6

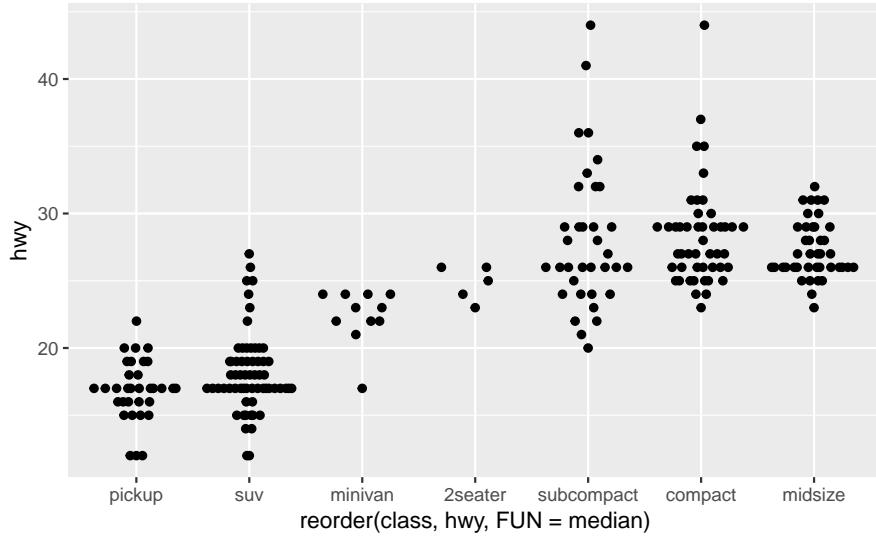
If you have a small dataset, it's sometimes useful to use `geom_jitter()` to see the relationship between a continuous and categorical variable. The `ggbeeswarm` package provides a number of methods similar to `geom_jitter()`. List them and briefly describe what each one does.

There are two methods:

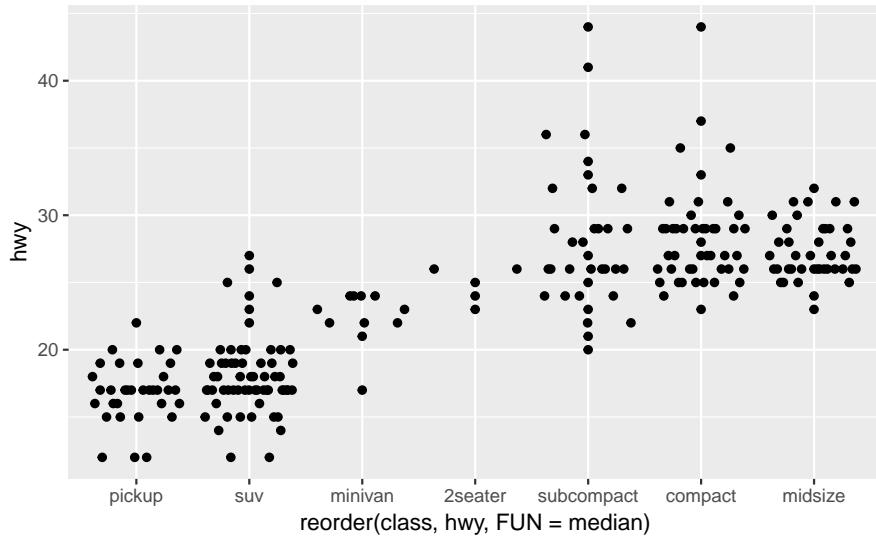
- `geom_quasirandom()` produces plots that are a mix of jitter and violin plots. There are several different methods that determine exactly how the random location of the points is generated.
- `geom_beeswarm()` produces a plot similar to a violin plot, but by offsetting the points.

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

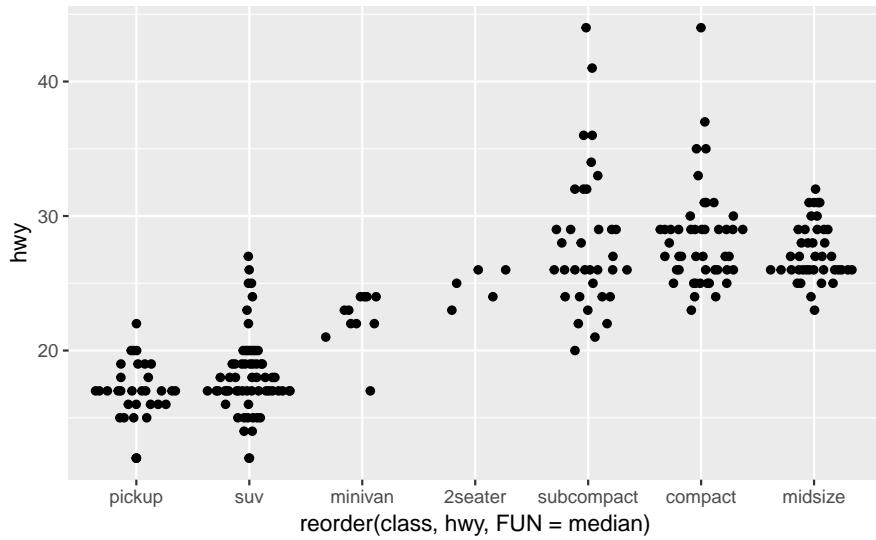
```
library("ggbeeswarm")
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                  y = hwy))
```



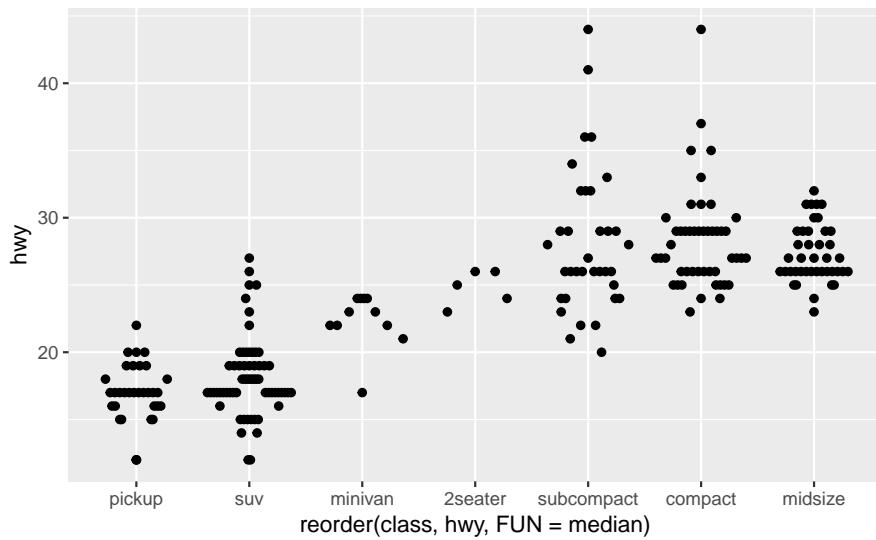
```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                  y = hwy),
                    method = "tukey")
```



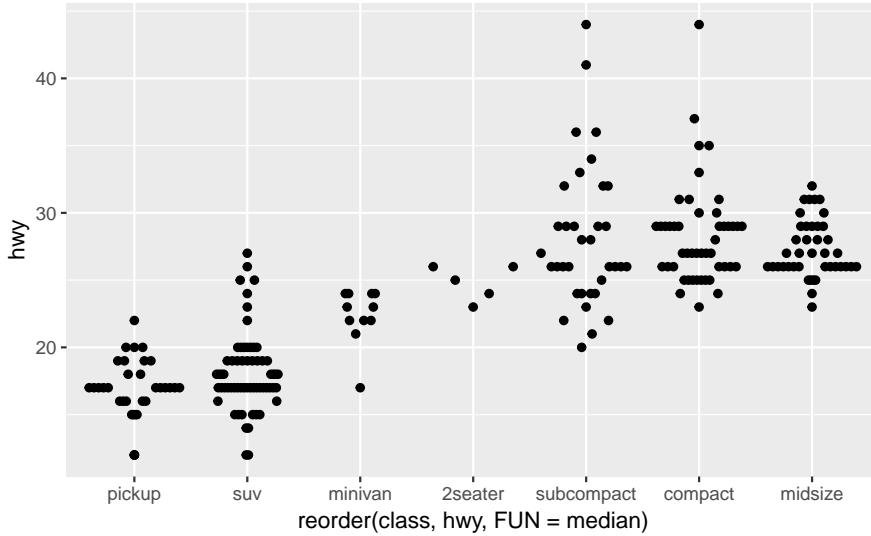
```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                  y = hwy),
                    method = "tukeyDense")
```



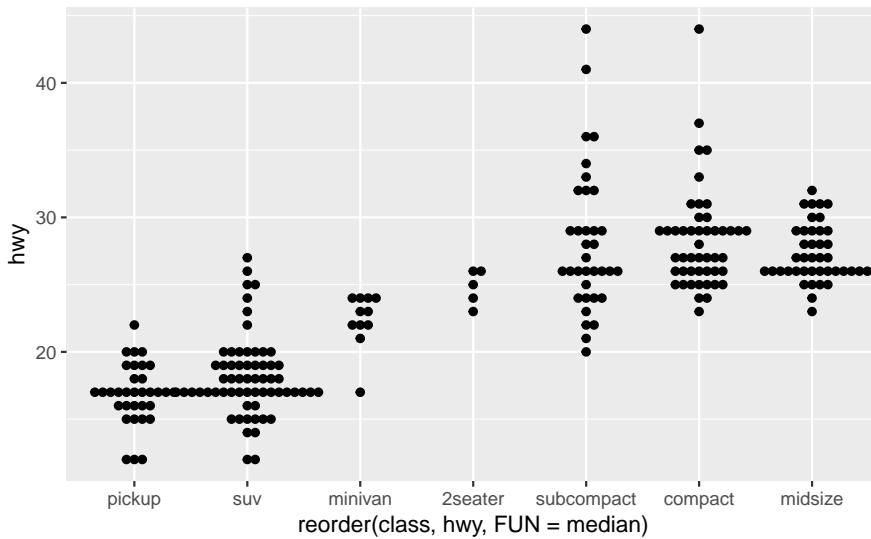
```
ggplot(data = mpg) +
  geom_quasirandom(mapping = aes(x = reorder(class, hwy, FUN = median),
                                  y = hwy),
                    method = "frowney")
```



```
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

Exercise 7.5.2.1

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.

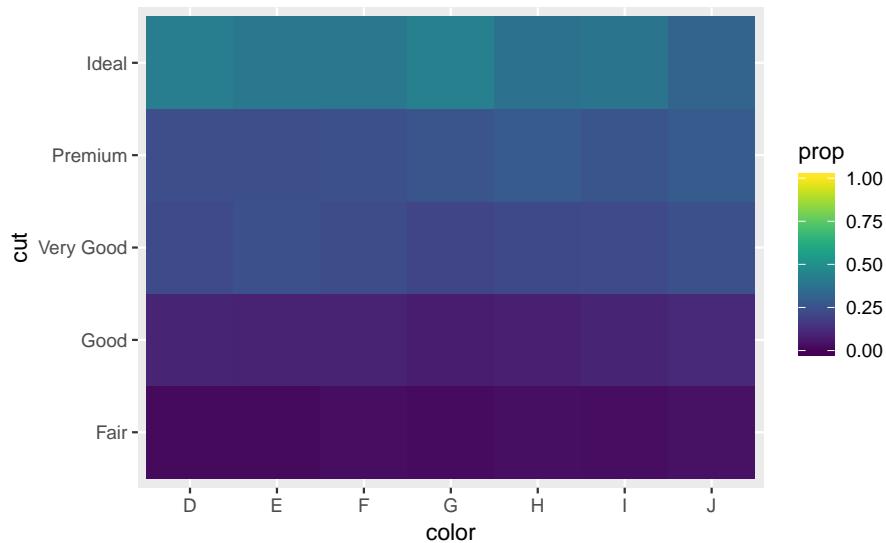
```
library(viridis)

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)) #from the viridis colour palette library

```

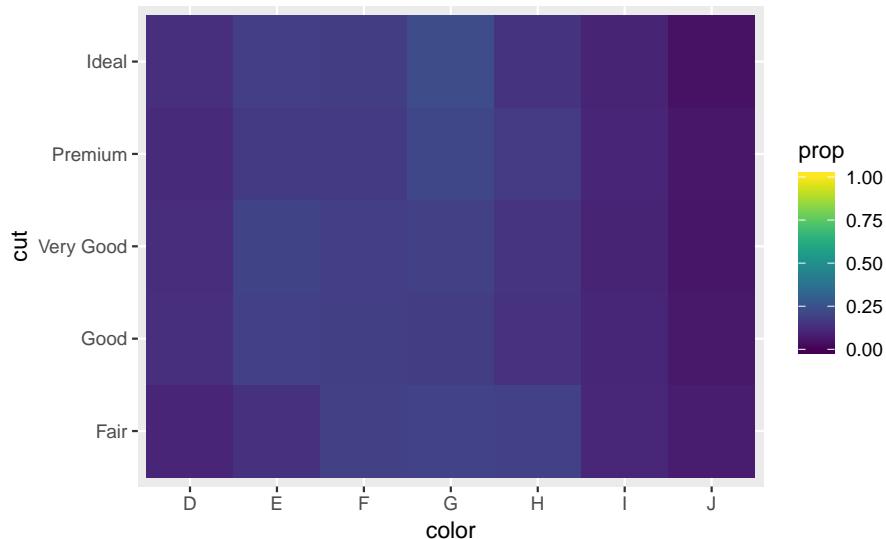


Similarly, to scale by the distribution of `color` within `cut`,

```

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))

```

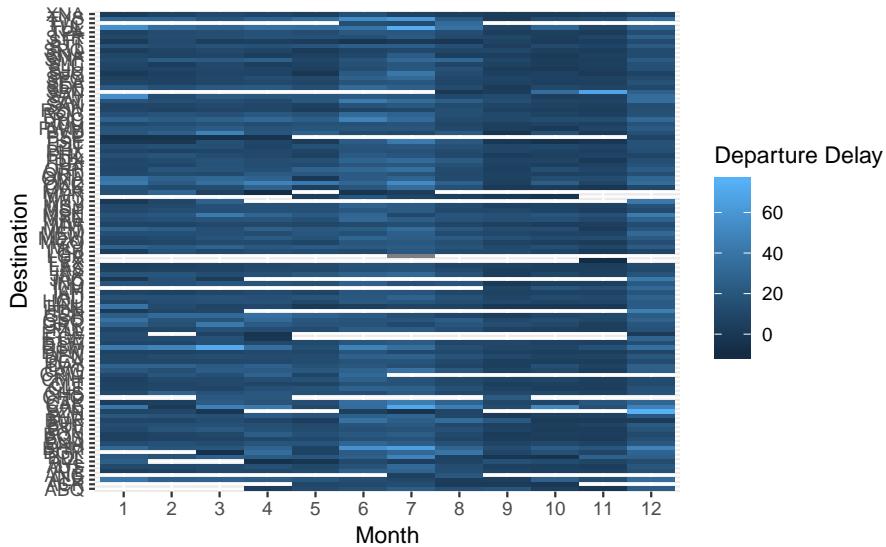


I add `limit = c(0, 1)` to put the color scale between (0, 1). These are the logical boundaries of proportions. This makes it possible to compare each cell to its actual value, and would improve comparisons across multiple plots. However, it ends up limiting the colors and makes it harder to compare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

Exercise 7.5.2.2

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")
```

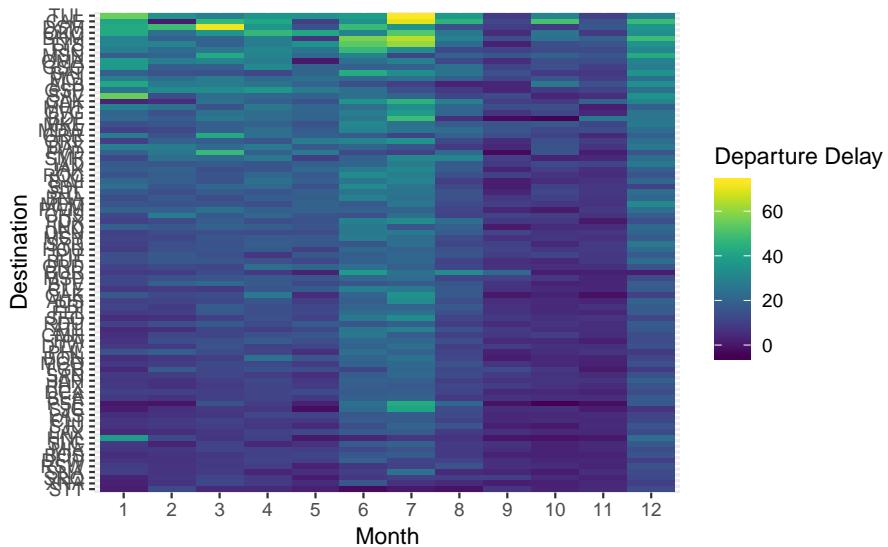


There are several things that could be done to improve it,

- 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 = 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")
```



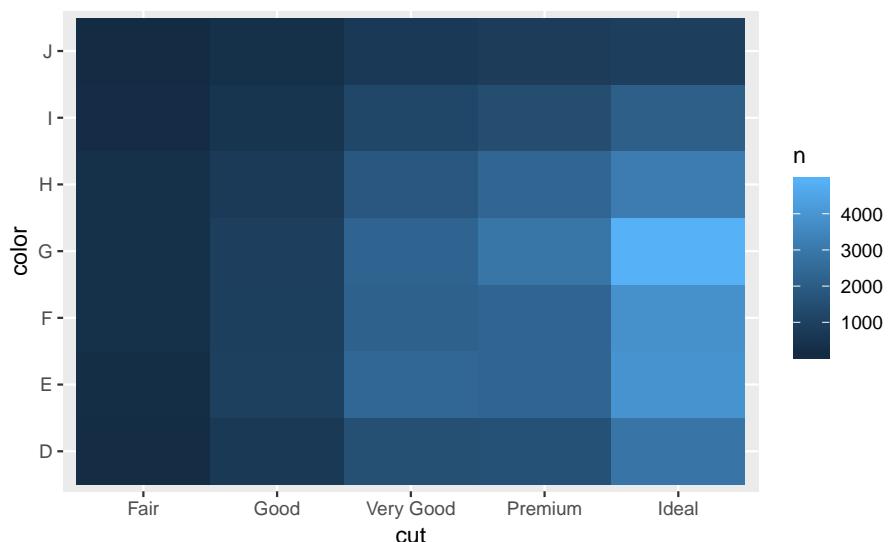
Exercise 7.5.2.3

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

Exercise 7.5.3.1

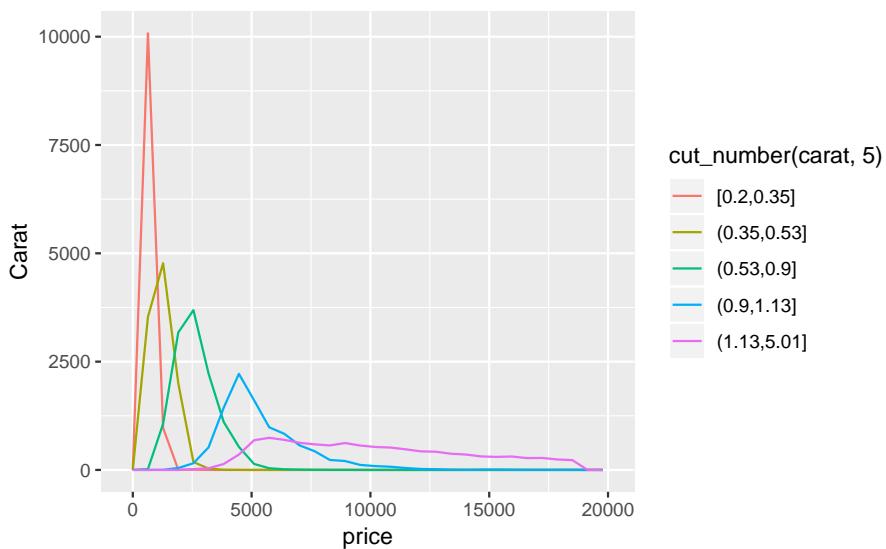
Instead of summarizing the conditional distribution with a box plot, you could use a frequency polygon. What do you need to consider when using `cut_width()` vs `cut_number()`? How does that impact a visualization of the 2d distribution of `carat` and `price`?

Both `cut_width()` and `cut_number()` split a variable into groups. When using `cut_width()`, we need to choose the width. When using `cut_number()`, we only need to specify the number of groups, and the width will be calculated.

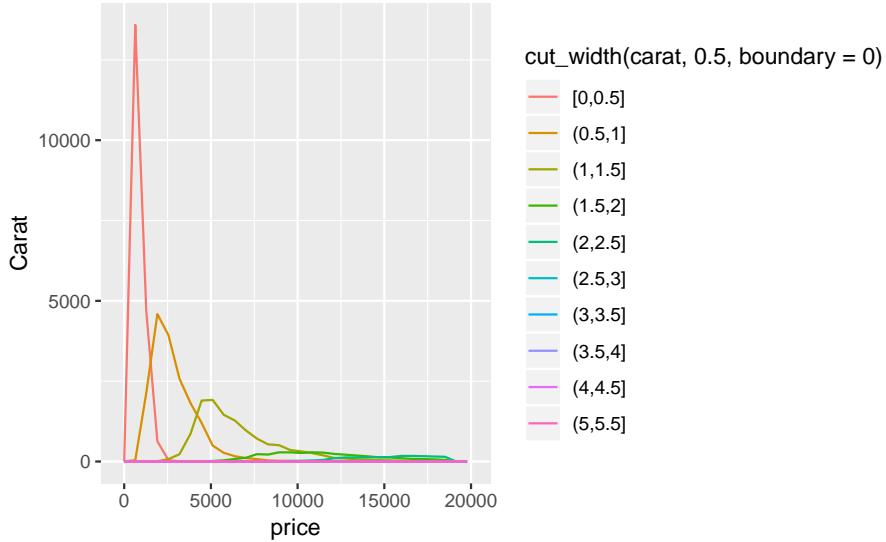
In either case, we need to choose the number or width of bins to be large enough to aggregate observations enough to remove noise but not so much as to remove all patterns.

The number of bins in the frequency plots will generally need to be less than those in the box plots in order to be interpretable.

```
ggplot(data = diamonds,
       mapping = aes(color = cut_number(carat, 5), x = price)) +
  geom_freqpoly() +
  ylab("Carat")
```



```
ggplot(data = diamonds,
       mapping = aes(color = cut_width(carat, 0.5, boundary = 0), x = price)) +
  geom_freqpoly() +
  ylab("Carat")
```

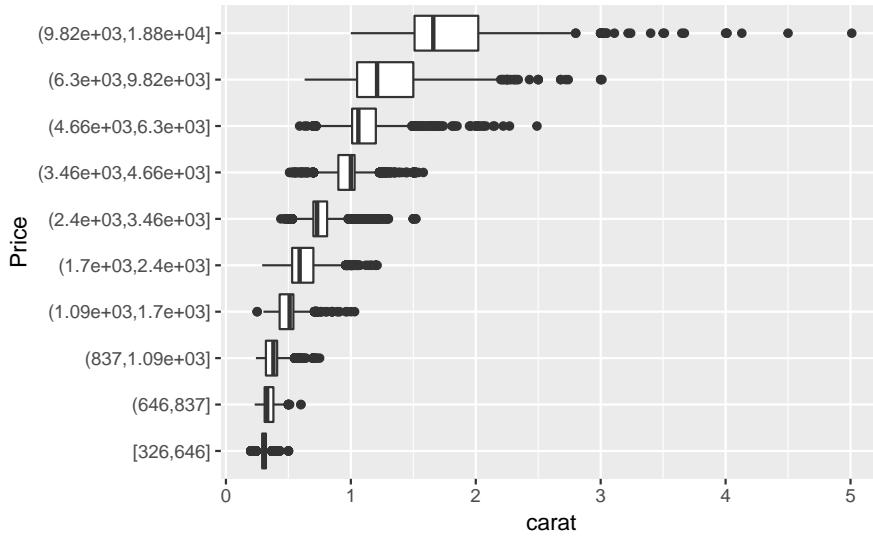


Exercise 7.5.3.2

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

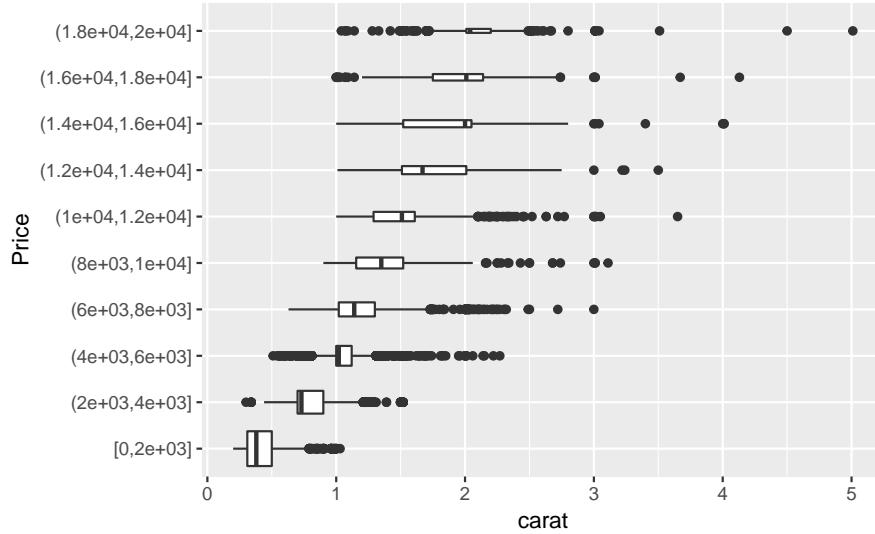
With a box plot, partitioning into an 10 bins with the same number of observations:

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



With a box plot, partitioning into 10 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")
```



Exercise 7.5.3.3

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

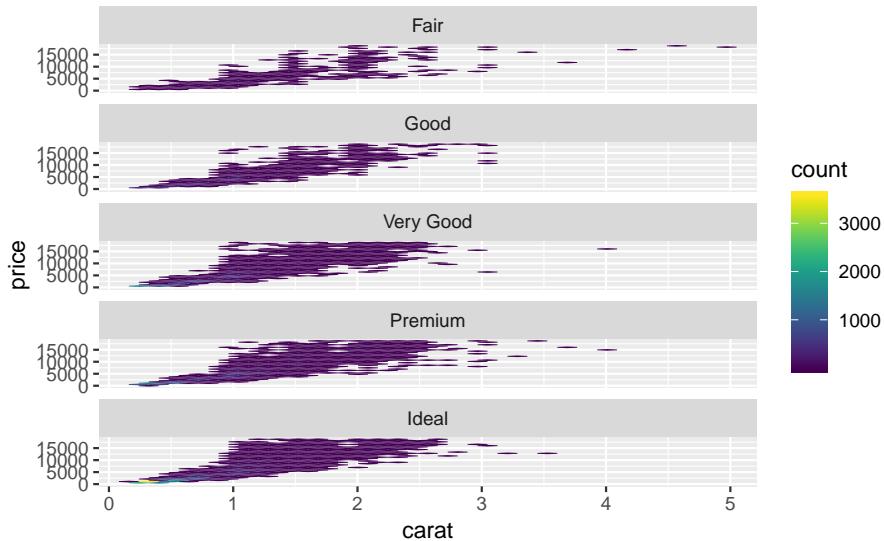
The distribution of very large diamonds is more variable. I am not surprised, since I knew little about diamond prices. After the fact, it does not seem surprising (as many things do). I would guess that this is due to the way in which diamonds are selected for retail sales. Suppose that someone selling a diamond only finds it profitable to sell it if some combination of size, cut, clarity, and color are above a certain threshold. The smallest diamonds are only profitable to sell if they are exceptional in all the other factors (cut, clarity, and color), so the small diamonds sold have similar characteristics. However, larger diamonds may be profitable regardless of the values of the other factors. Thus we will observe large diamonds with a wider variety of cut, clarity, and color and thus more variability in prices.

Exercise 7.5.3.4

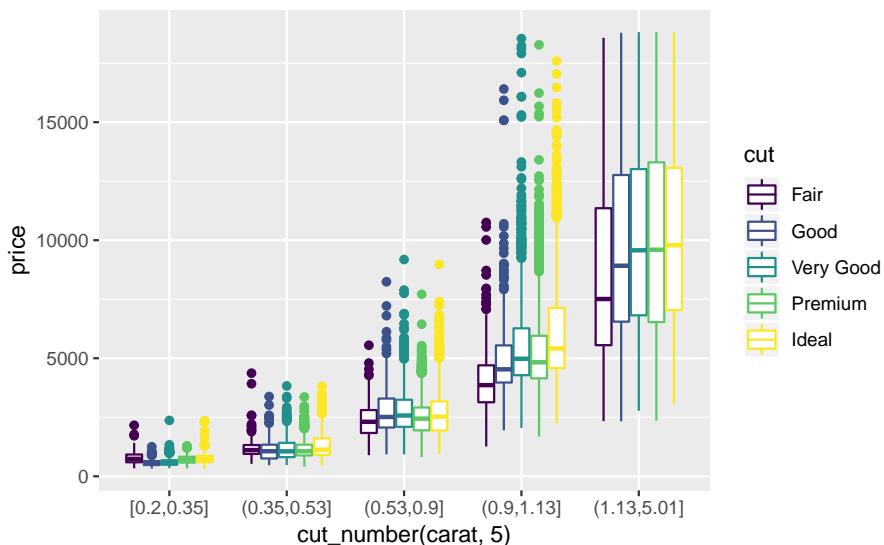
Combine two of the techniques you've learned to visualize the combined distribution of cut, carat, and price.

There are many options to try, so your solutions may vary from mine. Here are a few options that I tried.

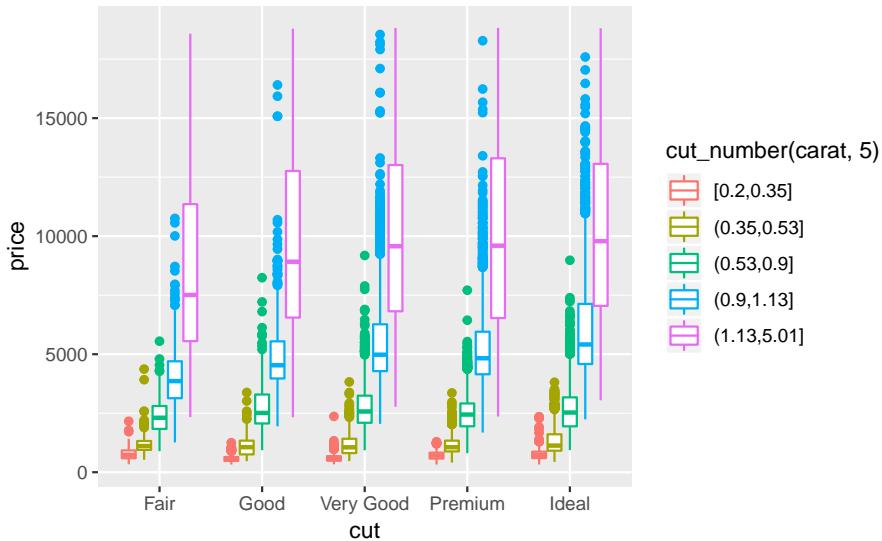
```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_hex() +
  facet_wrap(~ cut, ncol = 1) +
  scale_fill_viridis()
```



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



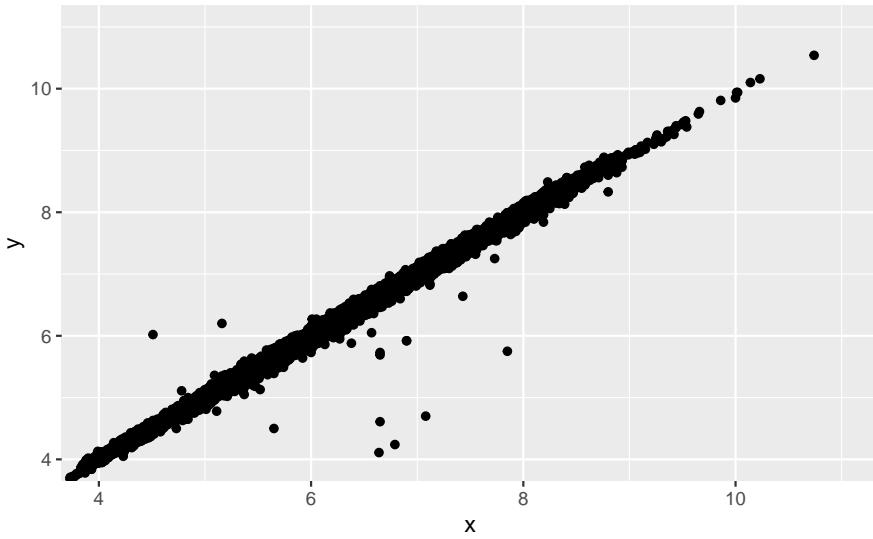
```
ggplot(diamonds, aes(colour = cut_number(carat, 5), y = price, x = cut)) +
  geom_boxplot()
```



Exercise 7.5.3.5

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

Introduction

No exercises.

Chapter 10

Tibbles

10.1 Introduction

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

10.2 Creating Tibbles

No exercises

10.3 Tibbles vs. data.frame

No exercises

10.4 Subsetting

No exercises

10.5 Interacting with older code

No exercises

10.6 Exercises

Exercise 10.6.1

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

When we print `mtcars`, it prints all the columns.

```
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 5.25 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
```

But when we first convert `mtcars` to a tibble using `as_tibble()`, it prints on the first ten observations. There are also some other differences in formatting of the printed data frame.

```
as_tibble(mtcars)
#> # A tibble: 32 x 11
#>   mpg   cyl  disp  hp drat    wt  qsec vs am gear carb
#> * <dbl> <dbl>
#> 1 21     6 160. 110 3.9  2.62 16.5  0  1    4    4
#> 2 21     6 160. 110 3.9  2.88 17.0  0  1    4    4
#> 3 22.8   4 108.  93 3.85 2.32 18.6  1  1    4    1
#> 4 21.4   6 258. 110 3.08 3.22 19.4  1  0    3    1
#> 5 18.7   8 360. 175 3.15 3.44 17.0  0  0    3    2
#> 6 18.1   6 225. 105 2.76 3.46 20.2  1  0    3    1
#> # ... with 26 more rows
```

You can use the function `is_tibble()` to check whether a data frame is a tibble or not. The `mtcars` data frame is not a tibble.

```
is_tibble(mtcars)
#> [1] FALSE
```

But the `diamonds` and `flights` data are tibbles.

```
is_tibble(ggplot2::diamonds)
#> [1] TRUE
is_tibble(nycflights13::flights)
#> [1] TRUE
is_tibble(as_tibble(mtcars))
#> [1] TRUE
```

More generally, you can use the `class()` function to find out the class of an object. Tibbles has the classes `c("tbl_df", "tbl", "data.frame")`, while old data frames will only have the class `"data.frame"`.

```
class(mtcars)
#> [1] "data.frame"
class(ggplot2::diamonds)
#> [1] "tbl_df"      "tbl"        "data.frame"
class(nycflights13::flights)
#> [1] "tbl_df"      "tbl"        "data.frame"
```

If you are interested in reading more on R's classes, read the chapters on object oriented programming in Advanced R.

Exercise 10.6.2

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.

Exercise 10.6.3

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`.

Exercise 10.6.4

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.

For this example, I'll create a dataset called `annoying` with columns named 1 and 2.

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

1. To extract the variable called 1 run

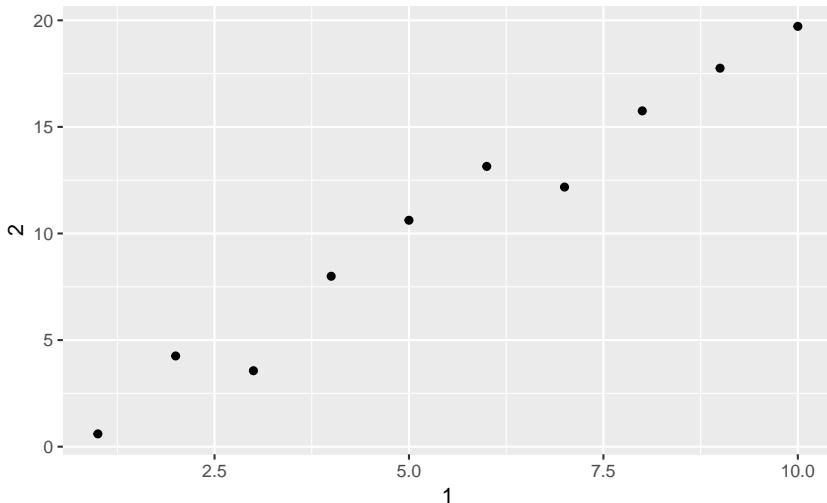
```
annoying[["1"]]
#> [1] 1 2 3 4 5 6 7 8 9 10
```

or

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

2. To create a scatter plot of 1 vs. 2 run

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



3. To add a new column 3 which is 2 divided by 1 run

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

or

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

4. To rename the columns to `one`, `two`, and `three`, run

```
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
```

Exercise 10.6.5

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

The function `tibble::enframe()` converts named vectors to a data frame with names and values

```
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
```

Exercise 10.6.6

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

The help page for the `print()` method of tibble objects is discussed in `?print.tbl_df`. The `n_extra` argument determines the number of extra columns to print information for.

Chapter 11

Data import

11.1 Introduction

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

11.2 Getting started

Exercise 11.2.1

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

Use the `read_delim()` function with the argument `delim="|"`.

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

Exercise 11.2.2

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.

Exercise 11.2.3

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.

Exercise 11.2.4

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'"
```

For `read_delim()`, we will will need to specify a delimiter, in this case `,`, and a quote argument.

```
x <- "x,y\n1,'a,b'"
read_delim(x, ", ", quote = "")
```

#> # A tibble: 1 x 2
#> x y
#> <int> <chr>
#> 1 1 a,b

However, this question is out of date. `read_csv()` now supports a quote argument, so the following code works.

```
read_csv(x, quote = "")
```

#> # A tibble: 1 x 2
#> x y
#> <int> <chr>
#> 1 1 a,b

Exercise 11.2.5

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 is 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
```

11.3 Parsing a vector

Exercise 11.3.1

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

The `locale` object has arguments to set the following:

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

Exercise 11.3.2

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
```

Exercise 11.3.3

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.

Exercise 11.3.4

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

Read the help page for `locale()` using `?locale` to learn about the different variables that can be set.

As an example, consider Australia. Most of the defaults values are valid, except that the date format is “(d)mm/yyyy”, meaning that January 2, 2006 is written as 02/01/2006.

However, default locale will parse that date as February 1, 2006.

```
parse_date("02/01/2006")
#> Warning: 1 parsing failure.
#> row # A tibble: 1 x 4 col      row    col expected     actual   expected <int> <int> <chr>
#> [1] NA
```

To correctly parse Australian dates, define a new `locale` object.

```
au_locale <- locale(date_format = "%d/%m/%Y")
```

Using `parse_date()` with the `au_locale` as its locale will correctly parse our example date.

```
parse_date("02/01/2006", locale = au_locale)
#> [1] "2006-01-02"
```

Exercise 11.3.5

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).

div>

Exercise 11.3.6

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 a good list of encodings 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 see the following sources.

- 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/>.

Programs that identify the encoding of text include

- `guess_encoding()` in the `reader` package
- `str_enc_detect()` in the `stringi` package
- `iconv`
- `chardet` (Python)

Exercise 11.3.7

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
```

11.4 Parsing a file

No exercises

11.5 Writing to a file

No exercises

11.6 Other Types of Data

No code

Chapter 12

Tidy Data

12.1 Introduction

```
library(tidyverse)
```

12.2 Tidy data

Exercise 12.2.1

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
```

Exercise 12.2.2

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?

In order to calculate cases per person, we need to divide cases by population for each country, year. This is easiest if country and population are two columns in a data frame with country and year rows.

For Table 2, we need to first create separate tables for cases and population and ensure that they are sorted in the same order.

```
t2_cases <- filter(table2, type == "cases") %>%
  rename(cases = count) %>%
  arrange(country, year)
```

```
t2_population <- filter(table2, type == "population") %>%
  rename(population = count) %>%
  arrange(country, year)
```

Calculate the cases per capita in a separate data frame.

```
t2_cases_per_cap <- t2_cases %>%
  mutate(population = t2_population$population,
        cases_per_cap = (cases / population) * 10000) %>%
  select(country, year, cases_per_cap)
```

Since the question asks us to store it back in the appropriate location, we will add new rows with `type = "cases_per_cap"` to `table2` and then sort by country, year, and variable type as in the original table.

```
t2_cases_per_cap <- t2_cases_per_cap %>%
  mutate(type = "cases_per_cap") %>%
  rename(count = cases_per_cap)

bind_rows(table2, t2_cases_per_cap) %>%
  arrange(country, year, type, count)
#> # A tibble: 18 x 4
#>   country     year type           count
#>   <chr>      <int> <chr>         <dbl>
#> 1 Afghanistan 1999 cases          745
#> 2 Afghanistan 1999 cases_per_cap  0.373
#> 3 Afghanistan 1999 population  19987071
#> 4 Afghanistan 2000 cases          2666
#> 5 Afghanistan 2000 cases_per_cap  1.29
#> 6 Afghanistan 2000 population  20595360
#> # ... with 12 more rows
```

Note that after adding the `cases_per_cap` rows, the type of `count` is coerced to `numeric` (double) because `cases_per_cap` is not an integer.

For `table4a` and `table4b`, we will create a separate table for cases per capita (`table4c`), with country rows and year columns.

```
table4c <-
  tibble(country = table4a$country,
        `1999` = table4a[["1999"]] / table4b[["1999"]] * 10000,
        `2000` = table4a[["2000"]] / table4b[["2000"]] * 10000)
table4c
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#>   <chr>      <dbl>   <dbl>
#> 1 Afghanistan 0.373    1.29
#> 2 Brazil       2.19    4.61
#> 3 China        1.67    1.67
```

Neither table is particularly easy to work with. Since `table2` has separate rows for cases and population we needed to generate a table with columns for cases and population where we could calculate cases per capita. `table4a` and `table4b` split the cases and population variables into different tables which made it easy to divide cases by population. However, we had to repeat this calculation for each row.

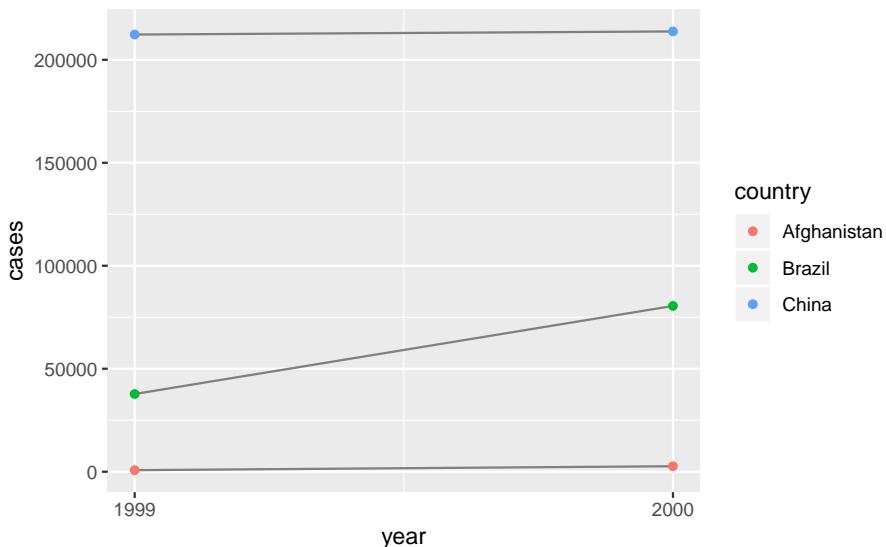
The ideal format of a data frame to answer this question is one with columns `country`, `year`, `cases`, and `population`. Then problem could be answered with a single `mutate()` call.

Exercise 12.2.3

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

Before creating the plot with change in cases over time, we need to filter the data frame to only include rows representing cases of TB.

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



12.3 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")
```

Exercise 12.3.1

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(`2015`:`2016`, key = "year", value = "return")
#> # A tibble: 4 x 3
#>   half year  return
#>   <dbl> <chr> <dbl>
#> 1     1 2015  1.88
#> 2     2 2015  0.59
#> 3     1 2016  0.92
#> 4     2 2016  0.17
```

The functions `spread()` and `gather()` are not perfectly symmetrical because column type information is not transferred between them. When we use `gather()` on a data frame, it throws away all the information about the original column types. Additionally, it has to coerce all the variables that are being gathered into a single type, since they are going into a single vector. Later if we `spread()` that data frame, the `spread()` function has no way to know what the original data types of the columns that were earlier gathered.

In this example, in the original table, the column `year` was numeric. After running `spread()` and `gather()` it is a character vector. Variable names are always converted to a character vector by `gather()`.

The functions `spread()` and `gather()` can be closer to symmetrical if we use the `convert` argument. It will try to convert character vectors to the appropriate type using `type.convert()`.

```
stocks %>%
  spread(key = "year", value = "return") %>%
  gather(`2015`:`2016`, key = "year", value = "return", convert = TRUE)
#> # A tibble: 4 x 3
#>   half year  return
#>   <dbl> <int> <dbl>
#> 1     1 2015  1.88
#> 2     2 2015  0.59
#> 3     1 2016  0.92
#> 4     2 2016  0.17
```

However, since `convert = TRUE` is guessing the appropriate type it still may not work.

Exercise 12.3.2

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
```

Exercise 12.3.3

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”. If we added another column with an indicator for the number observation it is, the code will work.

```
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
```

Exercise 12.3.4

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
)
```

To tidy `preg`, we need to use `gather()`. The variables in this data are

- `sex` (“female”, “male”)
- `pregnant` (“yes”, “no”)
- `count`, which is a non-negative integer representing the number of observations.

The observations in this data are unique combinations of sex and pregnancy status.

```
preg_tidy <- preg %>%
  gather(male, female, key = "sex", value = "count", na.rm = TRUE)
preg_tidy
#> # A tibble: 3 x 3
#>   pregnant sex     count
#> * <chr>    <chr>   <dbl>
#> 1 no       male     20
#> 2 yes      female   10
#> 3 no       female   12
```

However, we should consider the missing value in the male, non-pregnant row. Is it missing due to missing data, or missing due to structural reasons? This will be discussed in the upcoming section on Missing Values. Supposing that these data represent observations from a species in which it is impossible for males to get pregnant (not seahorses), then that missing value is structural. In the non-tidy data frame we had to include that structural missing value explicitly with an `NA` entry. However, in the tidy version we can drop that row since it is an impossible combination. We can do that by adding the argument `na.rm = TRUE` to `gather()`.

```
preg_tidy2 <- preg %>%
  gather(male, female, key = "sex", value = "count", na.rm = TRUE)
preg_tidy2
#> # A tibble: 3 x 3
#>   pregnant sex     count
#> * <chr>    <chr>   <dbl>
#> 1 no       male     20
#> 2 yes      female   10
#> 3 no       female   12
```

Though not necessary, there is one more way in which we can improve this data. If a variable takes two values, like `pregnant` and `sex` do, it is often preferable to store them as logical vectors. I will

```
preg_tidy3 <- preg_tidy2 %>%
  mutate(female = sex == "female",
         pregnant = pregnant == "yes") %>%
  select(female, pregnant, count)
preg_tidy3
#> # A tibble: 3 x 3
#>   female pregnant count
#>   <lgl>   <lgl>   <dbl>
#> 1 FALSE   FALSE     20
#> 2 TRUE    TRUE      10
#> 3 TRUE    FALSE     12
```

Note that I named the logical variable representing the sex `female`, not `sex`. This makes the meaning of the variable self-documenting. If the variable were named `sex` with values `TRUE` and `FALSE`, without reading the documentation, we wouldn’t know whether `TRUE` means male or female.

Apart from some minor memory savings, representing these variables as logical vectors results in more

clear and concise code. Compare the `filter()` calls to select non-pregnant females from `preg_tidy2` and `preg_tidy`.

```
filter(preg_tidy2, sex == "female", pregnant == "no")
#> # A tibble: 1 x 3
#>   pregnant sex     count
#>   <chr>    <chr>   <dbl>
#> 1 no      female     12
filter(preg_tidy3, female, !pregnant)
#> # A tibble: 1 x 3
#>   female pregnant count
#>   <lgl>   <lgl>   <dbl>
#> 1 TRUE    FALSE     12
```

12.4 Separating and Uniting

Exercise 12.4.1

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
```

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 those in `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
```

Exercise 12.4.2

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.

Exercise 12.4.3

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 `separate()`, splits columns a column into multiple groups using `by` separator, if the `sep` argument is a character vector, or character positions, if `sep` is numeric.

```
# example with separators
tibble(x = c("X_1", "X_2", "AA_1", "AA_2")) %>%
  separate(x, c("variable", "into"), sep = "_")
#> # A tibble: 4 x 2
#>   variable     into
#>   <chr>       <chr>
#> 1 X           1
#> 2 X           2
#> 3 AA          1
#> 4 AA          2

# example with position
tibble(x = c("X1", "X2", "Y1", "Y2")) %>%
  separate(x, c("variable", "into"), sep = c(1))
#> # A tibble: 4 x 2
#>   variable     into
#>   <chr>       <chr>
#> 1 X           1
#> 2 X           2
#> 3 Y           1
#> 4 Y           2
```

The function `extract()` uses a regular expression to specify groups in character vector and split that single character vector into multiple columns. This is more flexible than `separate()` because it does not require a common separator or specific column positions.

```
# example with separators
tibble(x = c("X_1", "X_2", "AA_1", "AA_2")) %>%
  extract(x, c("variable", "id"), regex = "([A-Z])_(\d+)")
#> # A tibble: 4 x 2
#>   variable id
#>   <chr>    <chr>
#> 1 X        1
#> 2 X        2
#> 3 AA      1
#> 4 AA      2
```

```
# example with position
tibble(x = c("X1", "X2", "Y1", "Y2")) %>%
  extract(x, c("variable", "id"), regex = "([A-Z])([0-9])")
#> # A tibble: 4 x 2
#>   variable id
#>   <chr>    <chr>
#> 1 X        1
#> 2 X        2
#> 3 Y        1
#> 4 Y        2

# example that separate could not parse
tibble(x = c("X1", "X20", "AA11", "AA2")) %>%
  extract(x, c("variable", "id"), regex = "([A-Z]+)([0-9]+)")
#> # A tibble: 4 x 2
#>   variable id
#>   <chr>    <chr>
#> 1 X        1
#> 2 X        20
#> 3 AA      11
#> 4 AA      2
```

Both `separate()` and `extract()` convert a single column to many columns. However, `unite()` converts many columns to one, with a choice of a separator to include between column values.

```
tibble(variable = c("X", "X", "Y", "Y"), id = c(1, 2, 1, 2)) %>%
  unite(x, variable, id, sep = "_")
#> # A tibble: 4 x 1
#>   x
#>   <chr>
#> 1 X_1
#> 2 X_2
#> 3 Y_1
#> 4 Y_2
```

In other words, with `extract()` and `separate()` only one column can be chosen, but there are many choices how to split that single column into different columns. With `unite()`, there are many choices as to which columns to include, but only choice as to how to combine their contents into a single vector.

12.5 Missing Values

Exercise 12.5.1

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

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

Exercise 12.5.2

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").

12.6 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", ...
#>   $ 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
```

Exercise 12.6.1

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
```

Exercise 12.6.2

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

The `separate()` function emits the warning “too few values”. If we check the rows for keys beginning with “`newrel_`”, we see that `sexage` is missing, 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
```

Exercise 12.6.3

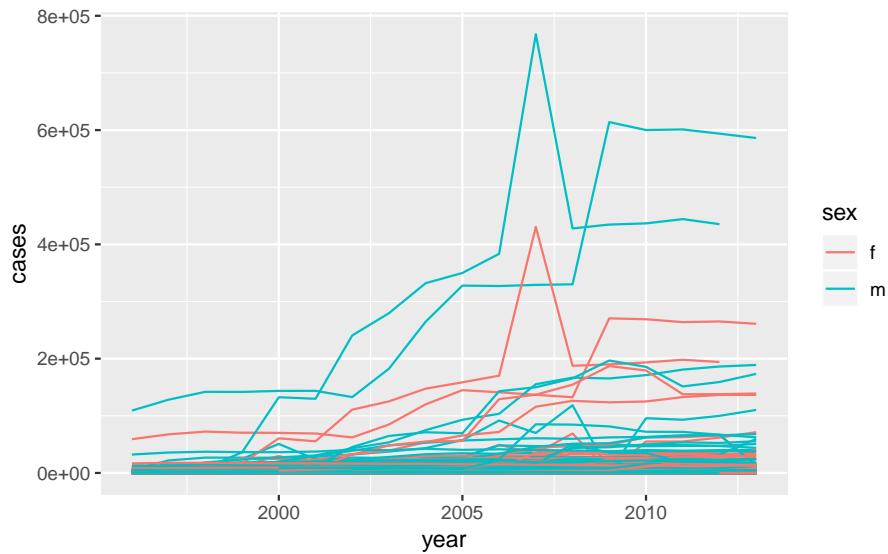
I claimed that `iso2` and `iso3` were redundant with `country`. Confirm this claim.

```
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>
```

Exercise 12.6.4

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.

12.7 Non-Tidy Data

No exercises

Chapter 13

Relational data

13.1 Introduction

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

The package `datamodelr` is used to draw database schema:

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

Exercise 13.1.1

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
- join `flights` with `airports` twice. The first join adds the location of the origin airport (`origin`). The second join adds the location of destination airport (`dest`).

Exercise 13.1.2

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`.

Exercise 13.1.3

`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).

Exercise 13.1.4

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’.

13.2 Keys

Exercise 13.2.1

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, ...
#> $ month          <int> 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, ...
#> $ 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, -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", ...
#> $ 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, 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, ...
```

Exercise 13.2.2

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 answer to each part follows.

1. 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
```

2. The primary key for `babynames::babynames` is `year, sex, name`. It is not 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
```

3. 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
```

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

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

5. There is no primary key for `ggplot2::diamonds`. The number of distinct rows in the dataset is less than the total number of rows, which implies that there is no combination of variables uniquely identifies the observations.

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

Exercise 13.2.3

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 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.

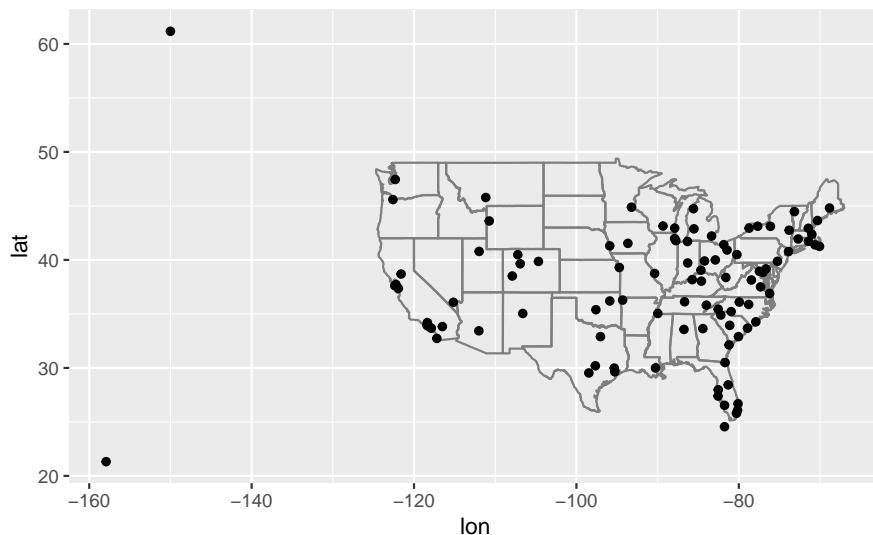
13.3 Mutating Joins

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

Exercise 13.3.1

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:

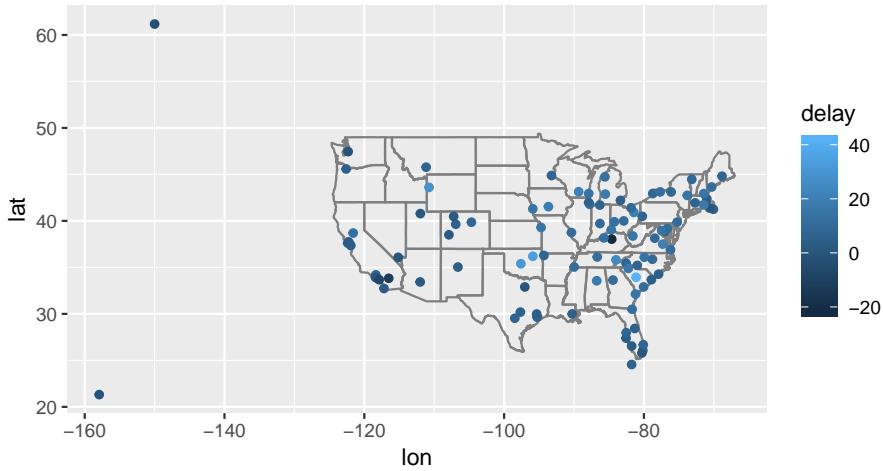
```
airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  ggplot(aes(lon, lat)) +
  borders("state") +
  geom_point() +
  coord_quickmap()
```



(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.

Exercise 13.3.2

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

You can perform one join after another. If duplicate variables are found, by default, dplyr will distinguish the two by adding `.x`, and `.y` to the ends of the variable names to solve naming conflicts.

```
airport_locations <- airports %>%
  select(faa, lat, lon)

flights %>%
  select(year:day, hour, origin, dest) %>%
  left_join(
    airport_locations,
    by = c("origin" = "faa")
  ) %>%
  left_join(
    airport_locations,
    by = c("dest" = "faa")
  )
#> # A tibble: 336,776 x 10
#>   year month day hour origin dest lat.x lon.x lat.y lon.y
#>   <int> <int> <int> <dbl> <chr> <chr> <dbl> <dbl> <dbl> <dbl>
#> 1  2013     1     1      5 EWR     IAH     40.7 -74.2  30.0 -95.3
```

```
#> 2 2013 1 1 5 LGA IAH 40.8 -73.9 30.0 -95.3
#> 3 2013 1 1 5 JFK MIA 40.6 -73.8 25.8 -80.3
#> 4 2013 1 1 5 JFK BQN 40.6 -73.8 NA NA
#> 5 2013 1 1 6 LGA ATL 40.8 -73.9 33.6 -84.4
#> 6 2013 1 1 5 EWR ORD 40.7 -74.2 42.0 -87.9
#> # ... with 3.368e+05 more rows
```

This default can be over-ridden using the `suffix` argument.

```
airport_locations <- airports %>%
  select(faa, lat, lon)

flights %>%
  select(year:day, hour, origin, dest) %>%
  left_join(
    airport_locations,
    by = c("origin" = "faa")
  ) %>%
  left_join(
    airport_locations,
    by = c("dest" = "faa"),
    suffix = c("_origin", "_dest")
  ) %>%
  # existing lat and lon variables in tibble gain the _origin suffix
  # new lat and lon variables are given _dest suffix
)
#> # A tibble: 336,776 x 10
#>   year month day hour origin dest lat_origin lon_origin lat_dest
#>   <int> <int> <int> <dbl> <chr> <chr> <dbl> <dbl> <dbl>
#> 1 2013     1     1     5 EWR  IAH    40.7   -74.2   30.0
#> 2 2013     1     1     5 LGA  IAH    40.8   -73.9   30.0
#> 3 2013     1     1     5 JFK  MIA    40.6   -73.8   25.8
#> 4 2013     1     1     5 JFK  BQN    40.6   -73.8   NA
#> 5 2013     1     1     6 LGA  ATL    40.8   -73.9   33.6
#> 6 2013     1     1     5 EWR  ORD    40.7   -74.2   42.0
#> # ... with 3.368e+05 more rows, and 1 more variable: lon_dest <dbl>
```

It's always good practice to have clear variable names.

Exercise 13.3.3

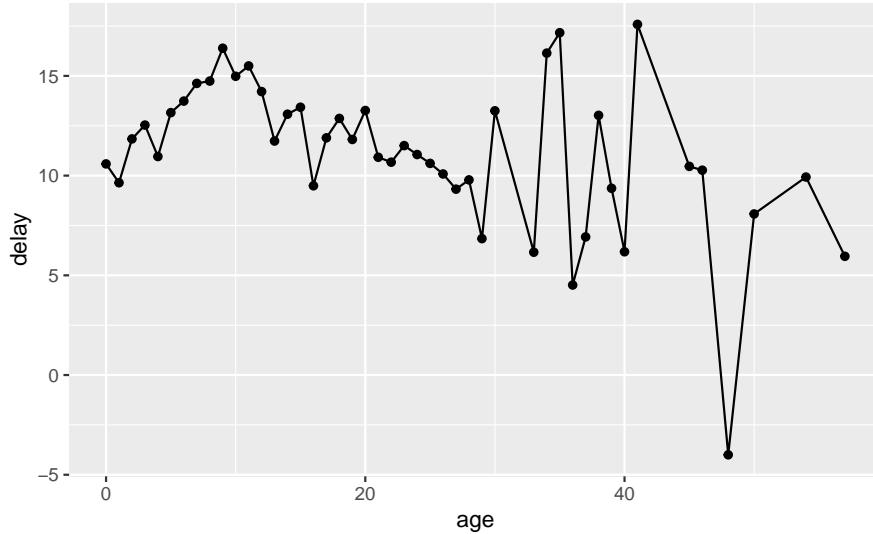
Is there a relationship between the age of a plane and its delays?

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).
```



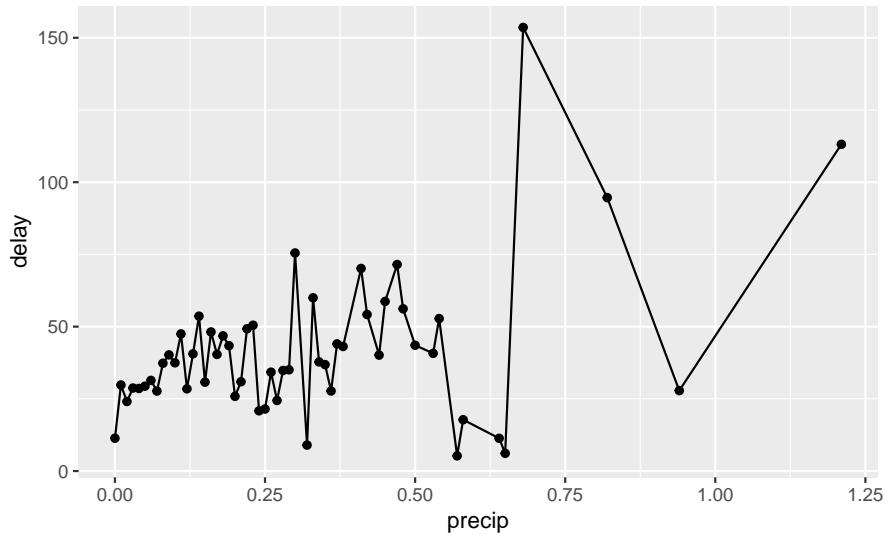
Exercise 13.3.4

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()
```



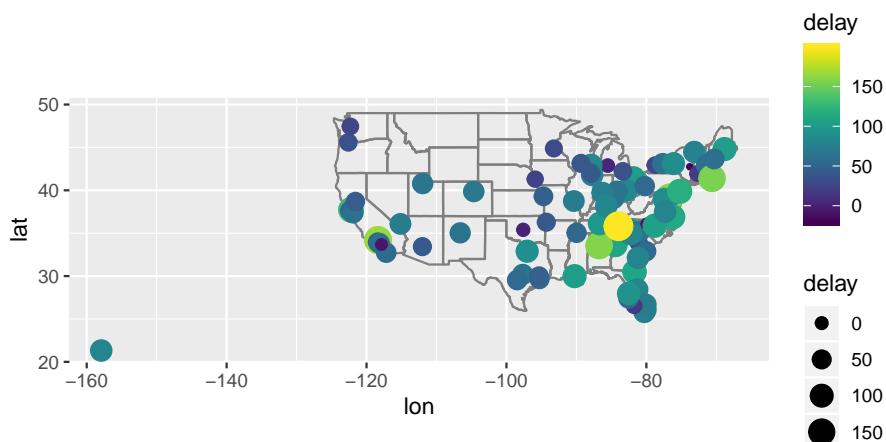
Exercise 13.3.5

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)
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_colour_viridis()
#> Warning: Removed 3 rows containing missing values (geom_point).
```



13.4 Filtering Joins

Exercise 13.4.1

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
```

Exercise 13.4.2

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>
```

Exercise 13.4.3

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, 4, 4, 6, 4, 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", "Regu...
#> $ 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, 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   1.6 Regu~  28   22
#> 2 1834 Acura Integ~ 1986 Subc- Manu~ Fron~    4   1.6 Regu~  28   23
#> 3 3037 Acura Integ~ 1987 Subc- Auto~ Fron~    4   1.6 Regu~  28   22
#> 4 3038 Acura Integ~ 1987 Subc- Manu~ Fron~    4   1.6 Regu~  28   23
#> 5 4183 Acura Integ~ 1988 Subc- Auto~ Fron~    4   1.6 Regu~  27   22
#> 6 4184 Acura Integ~ 1988 Subc- Manu~ Fron~    4   1.6 Regu~  28   23
#> # ... with 1.452e+04 more rows
```

Exercise 13.4.4

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
#>   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
```

Exercise 13.4.5

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.

Exercise 13.4.6

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. 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:

```
multi_carrier_planes <- 
  flights %>%
  filter(!is.na(tailnum)) %>%
  count(tailnum, carrier) %>%
  count(tailnum) %>%
  filter(nn > 1)
multi_carrier_planes
#> # A tibble: 17 x 2
#>   tailnum    nn
#>   <chr>     <int>
#> 1 N146PQ      2
#> 2 N153PQ      2
#> 3 N176PQ      2
#> 4 N181PQ      2
#> 5 N197PQ      2
#> 6 N200PQ      2
#> # ... with 11 more rows
```

There are 17 airplanes in this dataset that have had more than one carrier.

To see which carriers these planes have been associated, filter the `flights` by `tailnum` in `multi_carrier_planes`, and extract the unique combinations of `tailnum` and `carrier`.

```
multi_carrier_planes <-
  flights %>%
  semi_join(multi_carrier_planes, by = "tailnum") %>%
  select(tailnum, carrier) %>%
  distinct() %>%
  arrange(tailnum)
```

```
multi_carrier_planes
#> # A tibble: 34 x 2
#>   tailnum carrier
#>   <chr>    <chr>
#> 1 N146PQ  9E
#> 2 N146PQ  EV
#> 3 N153PQ  9E
#> 4 N153PQ  EV
#> 5 N176PQ  9E
#> 6 N176PQ  EV
#> # ... with 28 more rows
```

The names of airlines are easier to understand than the two-letter carrier codes. Join the multi-airline table with the associated airline in `airlines` using the `carrier` column. The spread the data so it has columns `carrier_1`, `carrier_2`, and so on. This is not tidy, but it is more easier to display.

```
carrier_transfer_tbl <-
  multi_carrier_planes %>%
  group_by(tailnum) %>%
  mutate(
    carrier_num = seq_along(tailnum),
    carrier_num = paste0("carrier_", carrier_num)
  ) %>%
  left_join(airlines, by = "carrier") %>%
  select(-carrier) %>%
  spread(carrier_num, name)
carrier_transfer_tbl
#> # A tibble: 17 x 3
#> # Groups:   tailnum [17]
#>   tailnum carrier_1      carrier_2
#>   <chr>    <chr>       <chr>
#> 1 N146PQ  Endeavor Air Inc. ExpressJet Airlines Inc.
#> 2 N153PQ  Endeavor Air Inc. ExpressJet Airlines Inc.
#> 3 N176PQ  Endeavor Air Inc. ExpressJet Airlines Inc.
#> 4 N181PQ  Endeavor Air Inc. ExpressJet Airlines Inc.
#> 5 N197PQ  Endeavor Air Inc. ExpressJet Airlines Inc.
#> 6 N200PQ  Endeavor Air Inc. ExpressJet Airlines Inc.
#> # ... with 11 more rows
```

13.5 Join problems

No exercises

13.6 Set operations

No exercises

Chapter 14

Strings

14.1 Introduction

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

14.2 String Basics

Exercise 14.2.1

In code that doesn't `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"
```

Exercise 14.2.2

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.

Exercise 14.2.3

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"
```

Exercise 14.2.4

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.

Exercise 14.2.5

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 "
```

Exercise 14.2.6

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.

This function needs to handle four cases.

1. `n == 0`: an empty string, e.g. `""`.
2. `n == 1`: the original vector, e.g. `"a"`.
3. `n == 2`: return the two elements separated by `"and"`, e.g. `"a and b"`.
4. `n > 2`: return the first `n - 1` elements separated by commas, and the last element separated by a comma and `"and"`, e.g. `"a, b, and c"`.

```
str_commasep <- function(x, delim = ",") {
  n <- length(x)
  if (n == 0) {
    ""
  } else if (n == 1) {
    x
  } else if (n == 2) {
    # no comma before and when n == 2
    str_c(x[[1]], "and", x[[2]], sep = " ")
  } else {
    # commas after all n - 1 elements
    not_last <- str_c(x[seq_len(n - 1)], delim)
    # prepend "and" to the last element
    last <- str_c("and", x[[n]], sep = " ")
    # combine parts with spaces
    str_c(c(not_last, last), collapse = " ")
  }
}
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"`

```
str_commasep(c("a", "b", "c", "d"))
```

`#> [1] "a, b, c, and d"`

14.3 Matching Patterns and Regular Expressions

14.3.1 Basic Matches

Exercise 14.3.1.1

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.

Exercise 14.3.1.2

How would you match the sequence "'\\" ?

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

Exercise 14.3.1.3

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("\\.\\.\\.\\..\\.."))
```

14.3.2 Anchors

Exercise 14.3.2.1

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

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

Exercise 14.3.2.2

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.

The answer to each part follows.

1. The words that start with “y” are:

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

2. End with “x”

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

3. Are exactly three letters long are

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

4. The words that have seven letters or more are

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

14.3.3 Character classes and alternatives

Exercise 14.3.3.1

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`.

The answer to each part follows.

1. Words starting with vowels

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

2. 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.

3. Words that end with “-ed” but not ending in “-eed”. This handles the special case of “-ed”, as well as words with a length great than two.

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

4. Words ending in `ing` or `ise`:

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

Exercise 14.3.3.2

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()` 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)"))
```

Exercise 14.3.3.3

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)
```

Exercise 14.3.3.4

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.

Exercise 14.3.3.5

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")
```

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.

14.3.4 Repetition**Exercise 14.3.4.1**

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

| Pattern | <code>{m,n}</code> | Meaning |
|---------|--------------------|-----------------|
| ? | <code>{0,1}</code> | Match at most 1 |
| + | <code>{1,}</code> | Match 1 or more |
| * | <code>{0,}</code> | Match 0 or more |

For example, let's repeat the let's rewrite the ?, +, and * examples using `{,}`.

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")

str_view(x, "CC{0,1}")

str_view(x, "CC+")

str_view(x, "CC{1,}")
```

Exercise 14.3.4.2

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}"`

The answer to each part follows.

1. `^.*$` will match any string. For example: `^.*$`: `c("dog", "$1.23", "lorem ipsum")`.
2. `"\\{.+\\"}` will match any string with curly braces surrounding at least one character. For example: `"\\{.+\\"}`: `c("{a}", "{abc}")`.
3. `\d{4}-\d{2}-\d{2}` will match 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”). For example: `\d{4}-\d{2}-\d{2}`: `2018-01-11`
4. `"\\\\{4}"` is `\\{4}`, which will match four backslashes. For example: `"\\\\{4}"`: `"\\\\\\\\\\\\\\\\"`.

Exercise 14.3.4.3

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.

The answer to each part follows.

1. This regex finds all words starting with three consonants.

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

2. This regex finds three or more vowels in a row:

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

3. This regex finds two or more vowel-consonant pairs in a row.

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

Exercise 14.3.4.4

Solve the beginner regexp crosswords at <https://regexecrossword.com/challenges/>

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

14.3.5 Grouping and backreferences

Exercise 14.3.5.1

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"`

The answer to each part follows.

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".

Exercise 14.3.5.2

Construct regular expressions to match words that:

1. Start and end with the same character.
2. Contain a repeated pair of letters (e.g. `church`'s contains `sch` repeated twice.)
3. Contain one letter repeated in at least three places (e.g. `eleven`'s contains `thre`e's.)

The answer to each part follows.

1. This regular expression matches words that start and end with the same character.

```
str_view(stringr::words, "^(.)((.*\\1$)|\\1?\\$)", match = TRUE)
```

2. This regex matches words that contain a repeated pair of letters.

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

These patterns checks for any pair of repeated “letters”.

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

```
str_view(words, "([[:letter:]])..\1")
```

Note that these patterns are case sensitive. Use the case insensitive flag if you want to check for repeated pairs of letters with different capitalization.

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

3. This regex matches words that contain one letter repeated in at least three places.

```
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"
```

14.4 Tools

14.4.1 Detect matches

No exercises

14.4.2 Exercises

Exercise 14.4.2.1

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

The answer to each part follows.

1. Words that start or end with x?

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

2. Words starting with vowel and ending with consonant.

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

3. There is not a simple regular expression to match words that contain at least one of each vowel. The regular expression would need to consider all possible orders in which the vowels could occur.

```
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.
```

To check that this pattern works, test it on a pattern that should match

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

Using multiple `str_detect()` calls, one pattern for each vowel, produces a much simpler and readable answer.

```
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.

4. The word with the highest number of vowels is

```
vowels <- str_count(words, "[aeiou]")
words[which(vowels == max(vowels))]
#> [1] "appropriate" "associate"   "available"    "colleague"   "encourage"
#> [6] "experience"  "individual"  "television"
```

The word with the highest proportion of vowels is

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

14.4.3 Extract Matches

Exercise 14.4.3.1

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)
```

Exercise 14.4.3.2

From the Harvard sentences data, extract:

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

The answer to each part follows.

1. Finding the first word in each sentence requires defining what a pattern constitutes a word. For the purposes of this question, I'll consider a word any contiguous set of letters.

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

2. This pattern finds 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"
```

3. Finding all plurals cannot be correctly accomplished with regular expressions alone. Finding plural words would at least require morphological information about words in the language. See WordNet for a resource that would do that. However, identifying words that end in an “s” and with more than three characters, in order to remove “as”, “is”, “gas”, etc., is a reasonable heuristic.

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

14.4.4 Grouped Matches

Exercise 14.4.4.1

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."
```

Exercise 14.4.4.2

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"
```

14.4.5 Replacing Matches

Exercise 14.4.5.1

Replace all forward slashes in a string with backslashes.

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

Exercise 14.4.5.2

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"))
```

Exercise 14.4.5.3

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"
```

14.4.6 Splitting

Exercise 14.4.6.1

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"
```

Exercise 14.4.6.2

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

Splitting by `boundary("word")` is a more sophisticated method to split a string into words. It recognizes non-space punctuation that splits words, and also removes punctuation while retaining internal non-letter characters that are parts of the word, e.g., “can’t” See the ICU website for a description of the set of rules that are used to determine word boundaries.

Consider this sentence from the official Unicode Report on word boundaries,

```
sentence <- "The quick ("brown") fox can't jump 32.3 feet, right?"
```

Splitting the string on spaces considers will group the punctuation with the words,

```
str_split(sentence, " ")
#> [[1]]
#> [1] "The"        "quick"       "("(brown)")" "fox"        "can't"      "jump"
#> [7] "32.3"       "feet,"      "right?"
```

However, splitting the string using `boundary("word")` correctly removes punctuation, while not separating “32.2” and “can’t”,

```
str_split(sentence, boundary("word"))
#> [[1]]
#> [1] "The"     "quick"   "brown"   "fox"     "can't"   "jump"   "32.3"   "feet"    "right"
```

Exercise 14.4.6.3

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.

14.4.7 Find matches

No exercises

14.5 Other types of patterns

Exercise 14.5.1

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"
```

Exercise 14.5.2

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
```

14.6 Other uses of regular expressions

No exercises

14.7 stringi

Exercise 14.7.1

Find the `stringi` functions that:

1. Count the number of words.
2. Find duplicated strings.
3. Generate random text.

The answer to each part follows.

1. To count the number of words use `stri_count_words()`.
2. To find duplicated strings use `stri_duplicated()`.

3. To generate random text the *stringi* package contains 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 (characters) in the text.

Exercise 14.7.2

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

You can set a locale to use when sorting with either `stri_sort(..., opts_collator=stri_opts_collator(locale = ...))` or `stri_sort(..., locale = ...)`.

Chapter 15

Factors

15.1 Introduction

Functions and packages:

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

15.2 Creating Factors

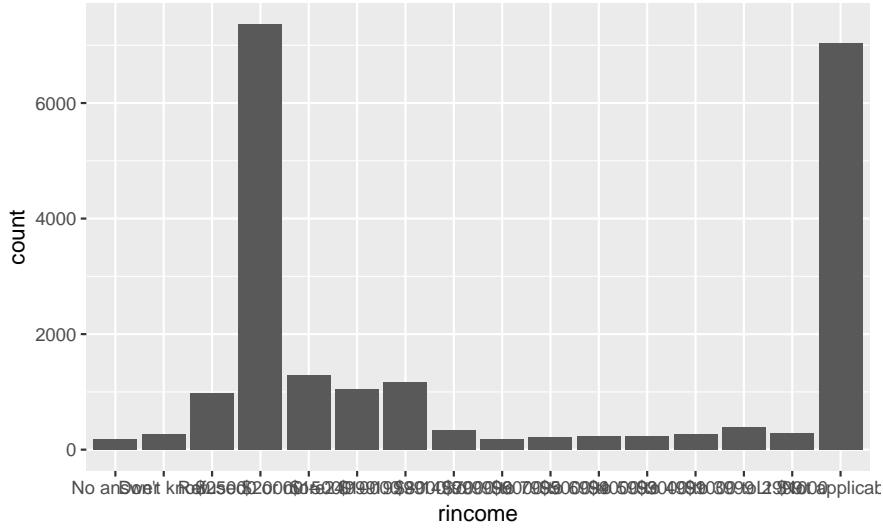
No exercises

15.3 General Social Survey

Exercise 15.3.1

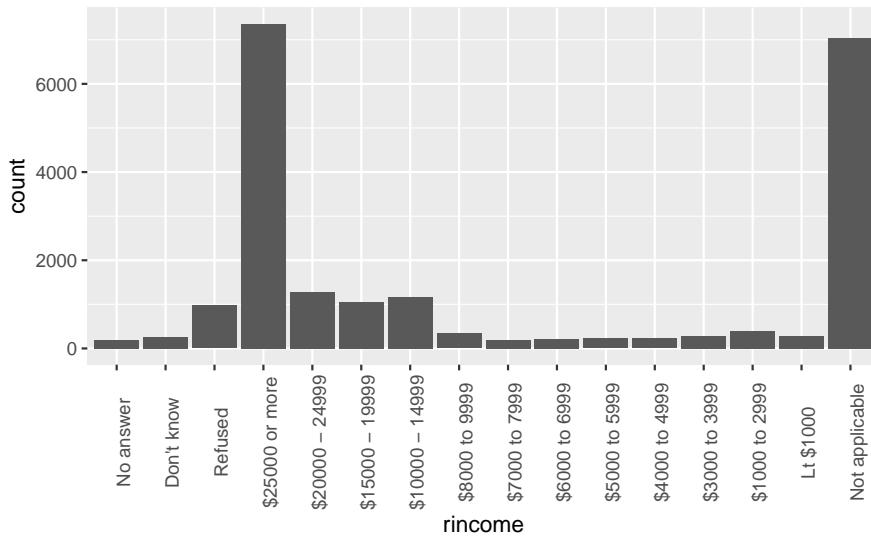
Explore the distribution of `rincome` (reported income). What makes the default bar chart hard to understand? How could you improve the plot?

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



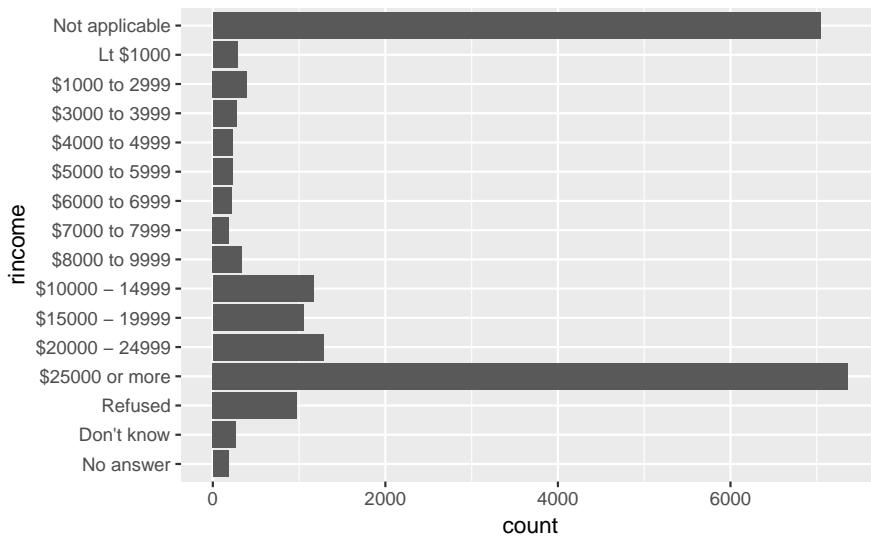
The default bar chart labels are too squished to read. One solution is to change the angle of the labels,

```
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.

Exercise 15.3.2

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
```

Exercise 15.3.3

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"
```

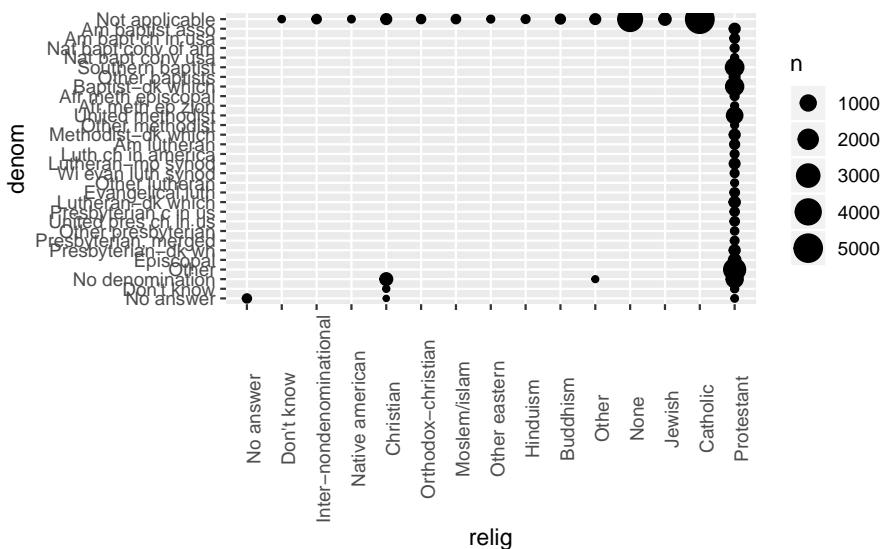
```
#> [7] "Presbyterian, merged" "Other presbyterian"
#> [10] "Presbyterian c in us" "Lutheran-dk which"
#> [13] "Other lutheran" "Wi evan luth synod"
#> [16] "Luth ch in america" "Am lutheran"
#> [19] "Other methodist" "United methodist"
#> [22] "Afr meth episcopal" "Baptist-dk which"
#> [25] "Southern baptist" "Nat bapt conv usa"
#> [28] "Am bapt ch in usa" "Am baptist asso"
#> [31] "Episcopal" "United pres ch in us"
#> [34] "Evangelical luth" "Lutheran-mo synod"
#> [37] "Methodist-dk which" "Methodist-dk which"
#> [40] "Afr meth ep zion" "Other baptists"
#> [43] "Other baptists" "Nat bapt conv of am"
#> [46] "Nat bapt conv usa" "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))
```



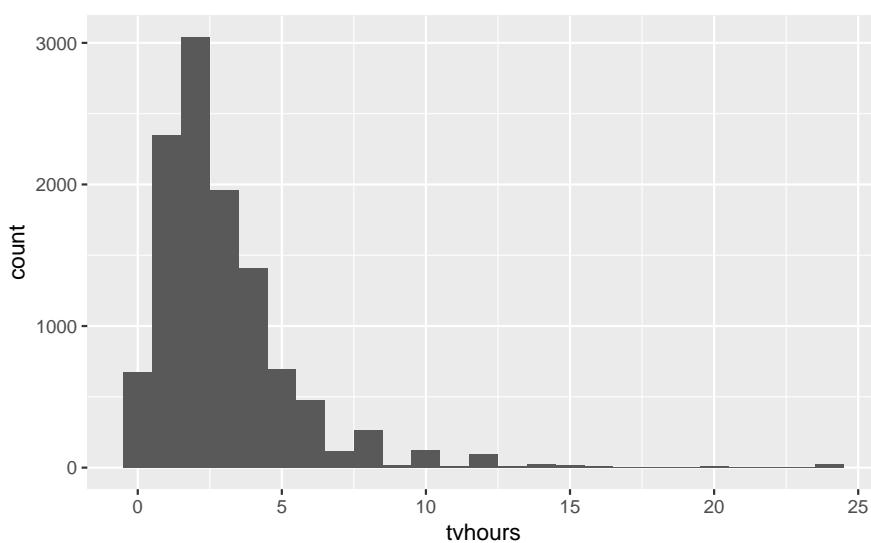
15.4 Modifying factor order

Exercise 15.4.1

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.

Exercise 15.4.2

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.

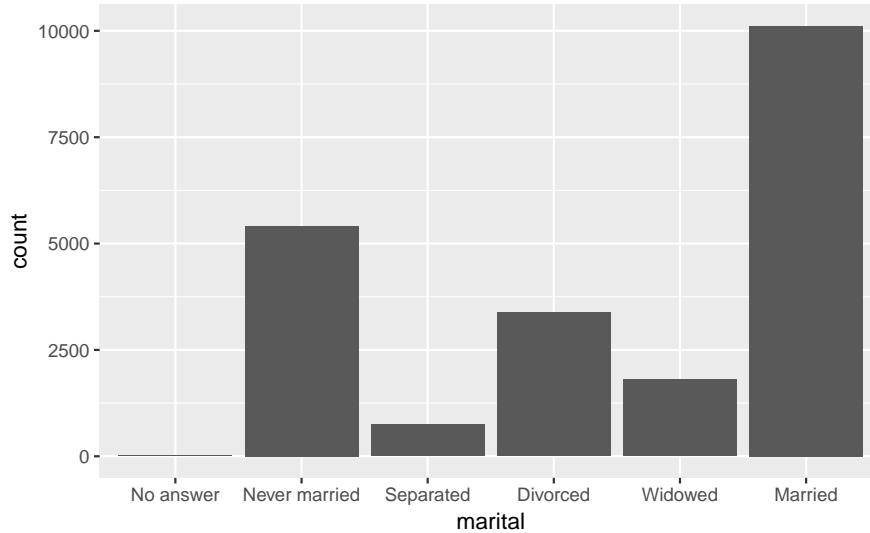
```
keep(gss_cat, is.factor) %>% names()
#> [1] "marital" "race"    "rincome" "partyid" "relig"   "denom"
```

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.

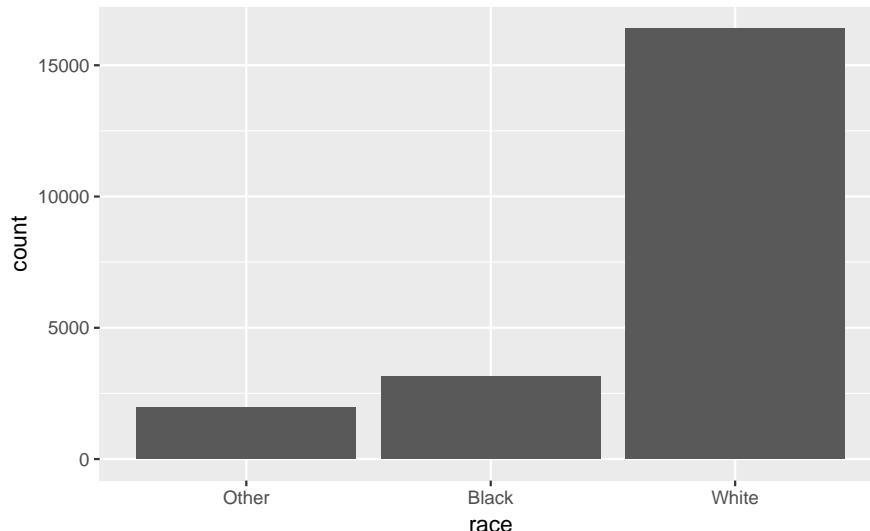
```
levels(gss_cat[["marital"]])
#> [1] "No answer"   "Never married" "Separated"    "Divorced"
#> [5] "Widowed"    "Married"
```

```
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.

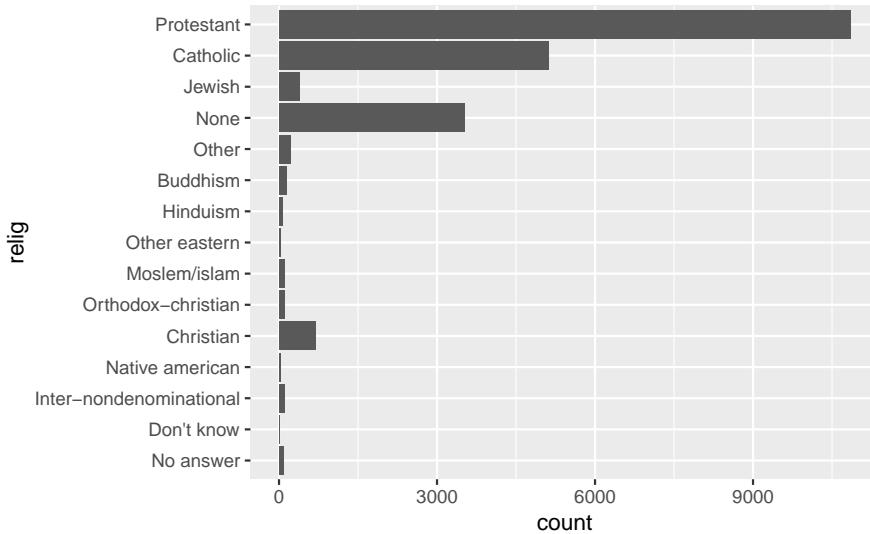
```
levels(gss_cat$rincome)
#> [1] "No answer"      "Don't know"     "Refused"        "$25000 or more"
```

```
#> [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"
```

Exercise 15.4.3

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.

15.5 Modifying factor levels

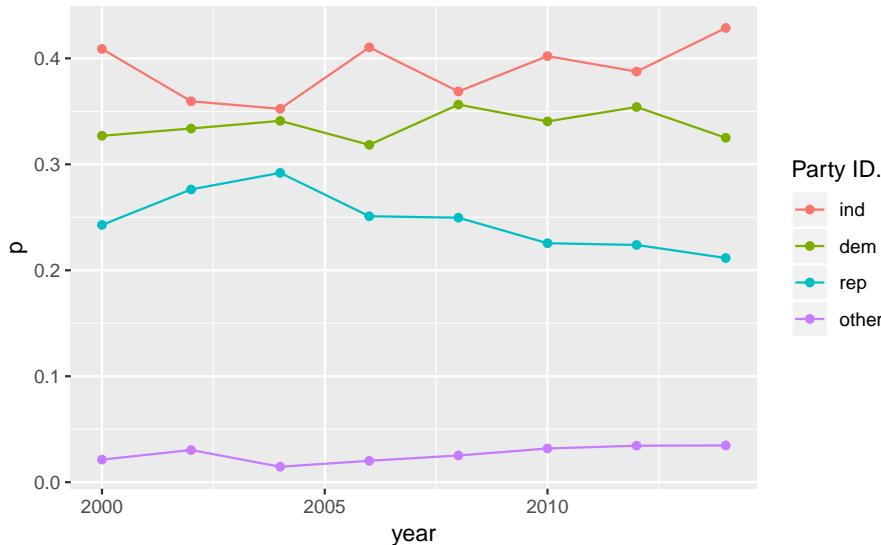
Exercise 15.5.1

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.")
```



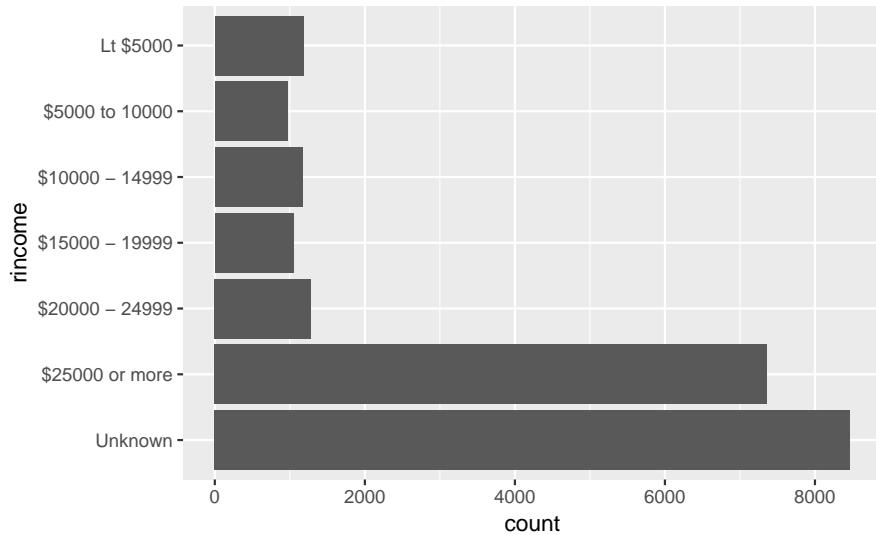
Exercise 15.5.2

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 would not use `fct_lump()`.⁴

```
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 16

Dates and times

16.1 Introduction

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

16.2 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"))
```

Exercise 16.2.1

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.

Exercise 16.2.2

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.

Exercise 16.2.3

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"
```

16.3 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.

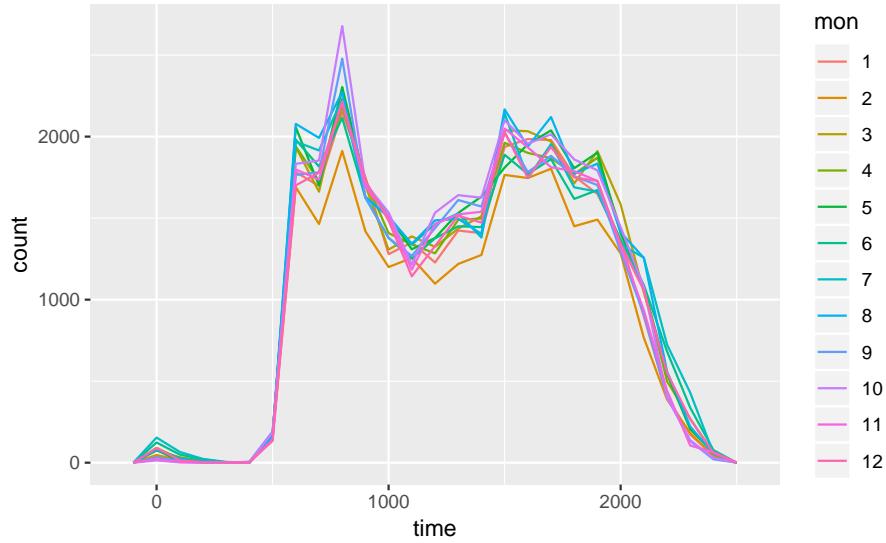
Exercise 16.3.1

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

Let's try plotting this by month:

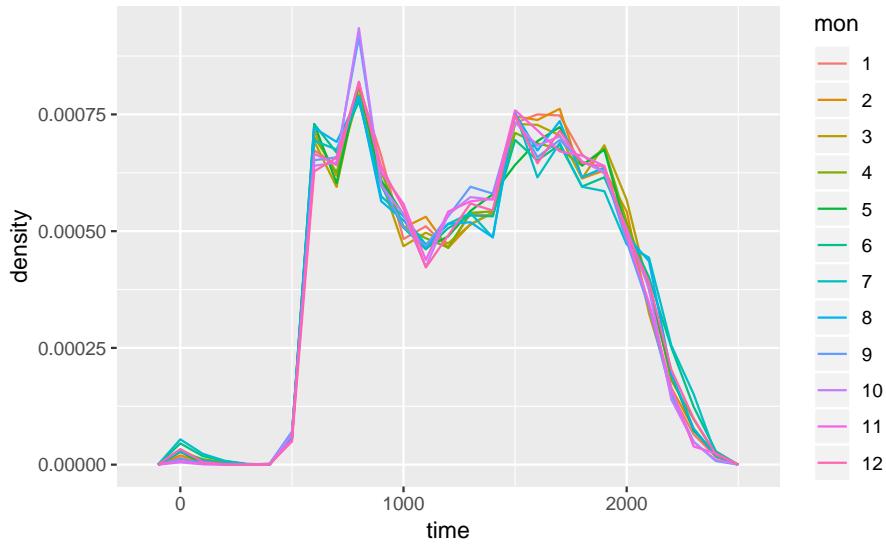
```
flights_dt %>%
  mutate(time = hour(dep_time) * 100 + minute(dep_time),
        mon = as.factor(month
                         (dep_time))) %>%
```

```
ggplot(aes(x = time, group = mon, colour = 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, colour = 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.

Exercise 16.3.2

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.

Exercise 16.3.3

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
```

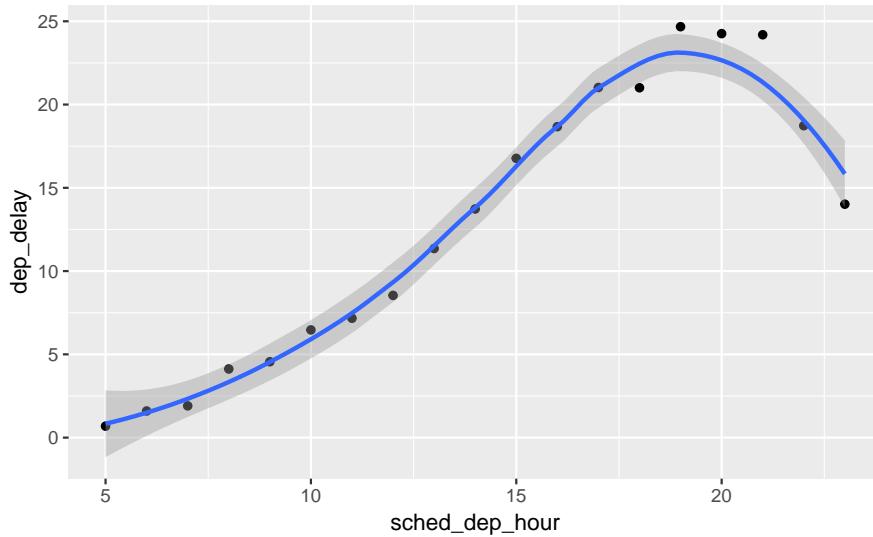
Exercise 16.3.4

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' and formula 'y ~ x'
```



Exercise 16.3.5

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

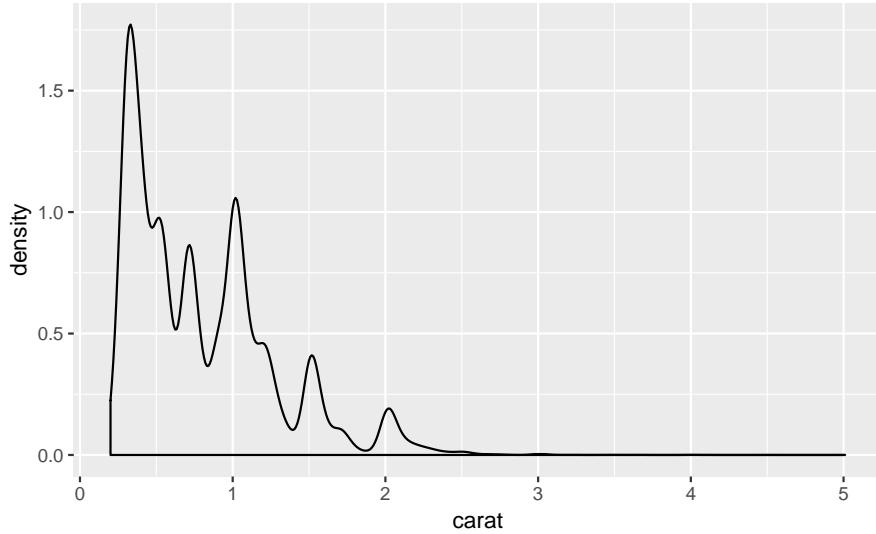
Sunday has the lowest average departure delay time and the lowest average arrival delay time.

```
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
```

Exercise 16.3.6

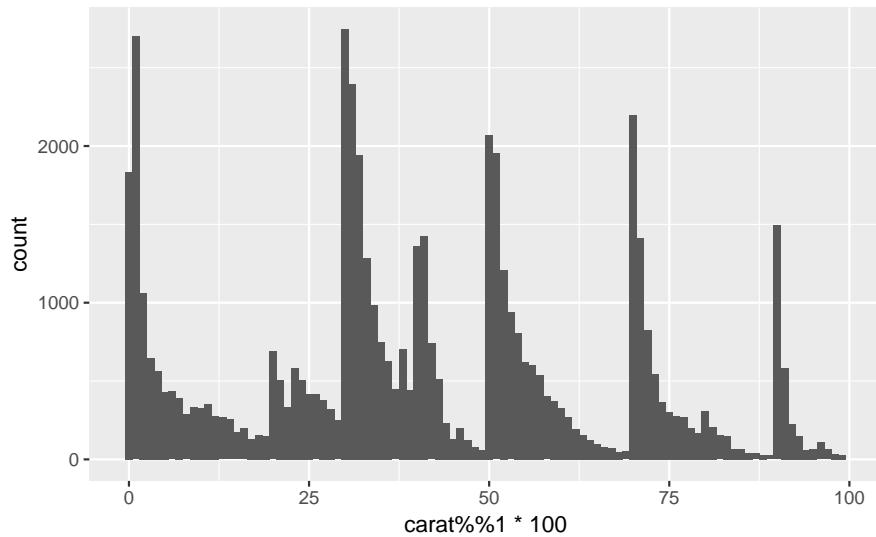
What makes the distribution of diamonds\$carat and flights\$sched_dep_time similar?

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



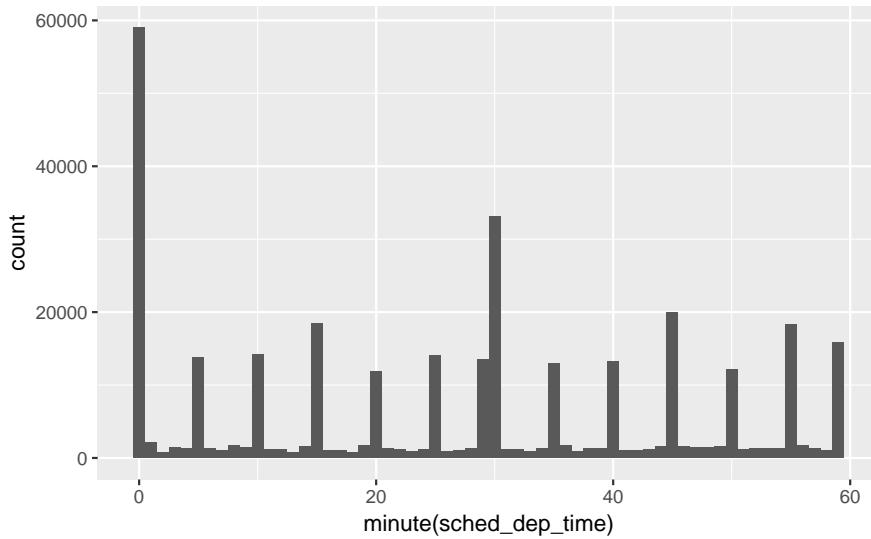
In both `carat` and `sched_dep_time` there are abnormally large numbers of values are at nice “human” numbers. In `sched_dep_time` it is at 00 and 30 minutes. In carats, it is at 0, 1/3, 1/2, 2/3,

```
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)
```

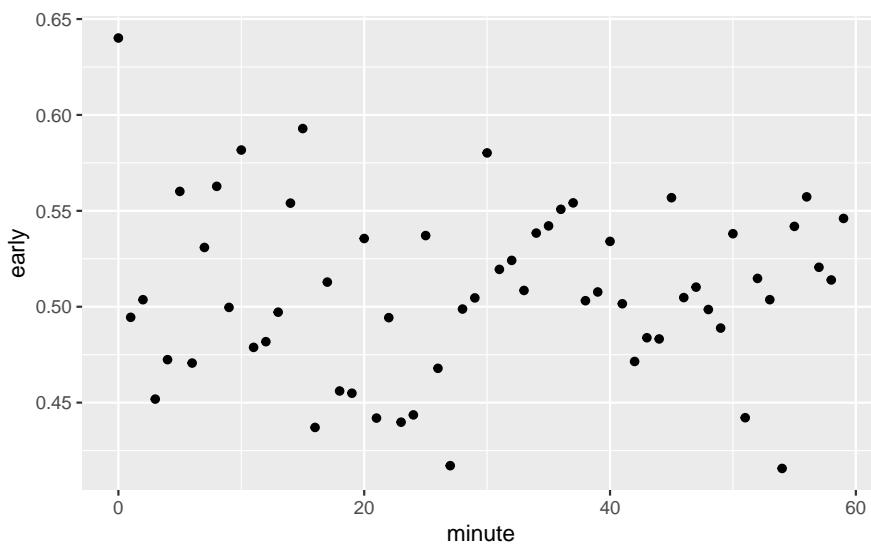


Exercise 16.3.7

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:

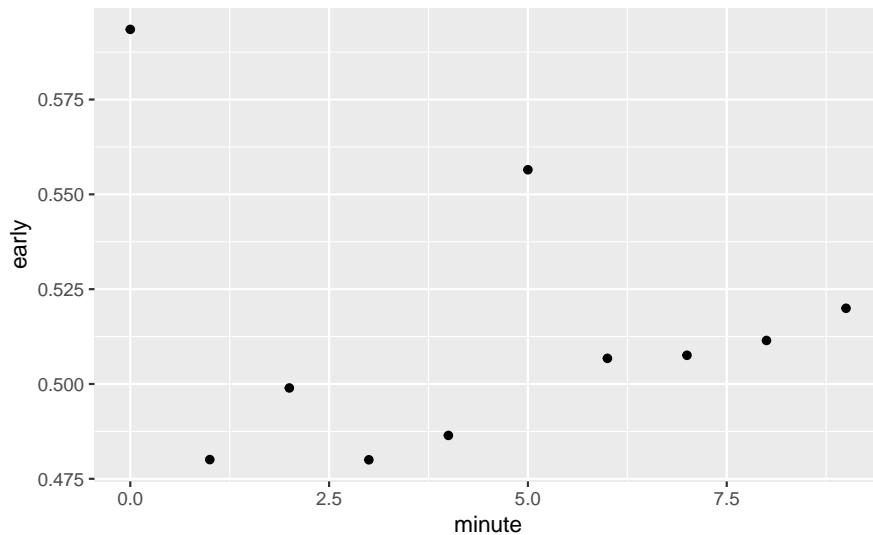
```
flights_dt %>%
  mutate(early = dep_delay < 0,
        minute = minute(sched_dep_time)) %>%
  group_by(minute) %>%
  summarise(early = mean(early)) %>%
  ggplot(aes(x = minute, y = early)) +
  geom_point()
```



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()
```



16.4 Time Spans

Exercise 16.4.1

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, June, 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.

Exercise 16.4.2

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.

Exercise 16.4.3

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"
```

Exercise 16.4.4

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
```

Exercise 16.4.5

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
```

16.5 Time Zones

No exercises.

Part III

Program

Chapter 17

Introduction

Chapter 18

Pipes

No exercises in this chapter.

Chapter 19

Functions

19.1 Introduction

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

19.2 When should you write a function?

Exercise 19.2.1

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 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
rescale01_alt(c(NA, 1:5), finite = TRUE)
#> [1] NA 0.00 0.25 0.50 0.75 1.00
```

Exercise 19.2.2

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

```

Exercise 19.2.3

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 code calculates the proportion of NA values in a vector.

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

I will write it as a function named `prop_na()` that takes a single argument `x`, and returns a single numeric value between 0 and 1.

```

prop_na <- function(x) {
  mean(is.na(x))
}

prop_na(c(0, 1, 2, NA, 4, NA))
#> [1] 0.333

```

This code standardizes a vector so that it sums to 1.

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

I'll write a function named `sum_to_one()`, which is a function of a single argument, `x`, the vector to standardize, and an optional argument `na.rm`. The optional argument, `na.rm`, makes the function more expressive, since it can handle NA values in two ways (returning NA or dropping them). Additionally, this makes `sum_to_one()` consistent with `sum()`, `mean()`, and many other R functions which have a `na.rm` argument. While the example code had `na.rm = TRUE`, I set `na.rm = FALSE` by default in order to make the function behave the same as the built-in functions like `sum()` and `mean()` in its handling of missing values.

```

sum_to_one <- function(x, na.rm = FALSE) {
  x / sum(x, na.rm = na.rm)
}

# no missing values
sum_to_one(1:5)
#> [1] 0.0667 0.1333 0.2000 0.2667 0.3333
# if any missing, return all missing
sum_to_one(c(1:5, NA))
#> [1] NA NA NA NA NA NA
# drop missing values when standarizing
sum_to_one(c(1:5, NA), na.rm = TRUE)

```

```
#> [1] 0.0667 0.1333 0.2000 0.2667 0.3333      NA
```

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

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

I'll write a function named `coef_variation()`, which takes a single argument `x`, and an optional `na.rm` argument.

```
coef_variation <- function(x, na.rm = FALSE) {
  sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)
}
coef_variation(1:5)
#> [1] 0.527
coef_variation(c(1:5, NA))
#> [1] NA
coef_variation(c(1:5, NA), na.rm = TRUE)
#> [1] 0.527
```

Exercise 19.2.4

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, na.rm = TRUE) {
  n <- length(x)
  m <- mean(x, na.rm = TRUE)
  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 one of the most commonly used is the following:(Doane and Seward 2011)

$$\text{skewness}(x) = \frac{n}{(n-1)(n-2)} \sum_{i=1}^n \left(\frac{x_i - \bar{x}}{s} \right)^3.$$

where \bar{x} is the sample mean and

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

is the sample standard deviation. The corresponding function is:

```
skewness <- function(x, na.rm = FALSE) {
  n <- length(x)
  m <- mean(x, na.rm = na.rm)
  s <- sd(x, na.rm = na.rm)
  n * sum(((x - m) / s)^ 3) / (n - 1) / (n - 2)
}
skewness(c(1, 2, 5, 100))
#> [1] 1.99
```

Exercise 19.2.5

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
```

Exercise 19.2.6

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.

Exercise 19.2.7

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)
```

19.3 Functions are for humans and computers

Exercise 19.3.1

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
```

Good names would include `recycle()` (R's name for this behavior), or `expand()`.

Exercise 19.3.2

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.

Exercise 19.3.3

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

`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 in `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. However, both functions are internally consistent. It would not be good practice to have `mu` and `sd` as arguments or `mean` and `Sigma` as arguments.

Exercise 19.3.4

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()`, the naming convention groups functions by their distribution.

If named `rnorm()`, and `dnorm()`, the naming convention groups functions by the action they perform.

- `r*` functions always sample from distributions: for example, `rnorm()`, `rbinom()`, `runif()`, and `rexp()`.
- `d*` functions calculate the probability density or mass of a distribution: For example, `dnorm()`, `dbinom()`, `dunif()`, and `dexp()`.

R distributions use this latter naming convention.

19.4 Conditional execution

Exercise 19.4.1

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.

Exercise 19.4.2

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 <- lubridate::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 morning"
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"
```

Exercise 19.4.3

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)) {
    "fizzbuzz"
  } else if (!(x %% 3)) {
    "fizz"
  } else if (!(x %% 5)) {
    "buzz"
  } else {
    x
  }
}
fizzbuzz(6)
#> [1] "fizz"
fizzbuzz(10)
#> [1] "buzz"
fizzbuzz(15)
#> [1] "fizzbuzz"
fizzbuzz(2)
#> [1] 2
```

Exercise 19.4.4

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 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.

Exercise 19.4.5

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

In `switch(n, ...)`, if `n` is numeric, it will return the `n`th argument from `...`. This means that if `n = 1`, `switch()` will return the first argument in `...`, if `n = 2`, the second, and so on. For example,

```

switch(1, "apple", "banana", "cantaloupe")
#> [1] "apple"
switch(2, "apple", "banana", "cantaloupe")
#> [1] "banana"

```

If you use a non-integer number for the first argument of `switch()`, it will ignore the non-integer part.

```

switch(1.2, "apple", "banana", "cantaloupe")
#> [1] "apple"
switch(2.8, "apple", "banana", "cantaloupe")
#> [1] "banana"

```

Note that `switch()` truncates the numeric value, it does not round to the nearest integer. While it is possible to use non-integer numbers with `switch()`, you should avoid it

Exercise 19.4.6

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

```

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

```

Experiment, then carefully read the documentation.

First, let's write a function `switcheroo()`, and see what it returns for different values of `x`.

```

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")
switcheroo("f")
```

The `switcheroo()` function returns "ab" for `x = "a"` or `x = "b"`, "cd" for `x = "c"` or `x = "d"`, and `NULL` for `x = "e"` or any other value of `x` not in `c("a", "b", "c", "d")`.

How does this work? The `switch()` function returns the first non-missing argument value for the first name it matches. Thus, when `switch()` encounters an argument with a missing value, like `a = ,`, it will return the value of the next argument with a non missing value, which in this case is `b = "ab"`. If `object` in `switch(object=)` is not equal to the names of any of its arguments, `switch()` will return either the last (unnamed) argument if one is present or `NULL`. Since "e" is not one of the named arguments in `switch()` (`a, b, c, d`), and no other unnamed default value is present, this code will return `NULL`.

The code in the question is shorter way of writing the following.

```
switch(x,
  a = "ab",
  b = "ab",
  c = "cd",
  d = "cd",
  NULL # value to return if x not matched
)
```

19.5 Function arguments

Exercise 19.5.1

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)  
}
```

Exercise 19.5.2

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?

This is the definition of the rule function from the chapter.

```
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 ---
```

You can currently supply multiple characters to the `pad` argument, but the output is will not be the desired width. The `rule()` function duplicates `pad` a number of times equal to the desired width minus the length of the title and five extra characters. This implicitly assumes that `pad` is only one character. If `pad` were two character, the output will be almost twice as long.

One way to handle this is to use `stringr::str_trunc()` to truncate the string, and `stringr::str_length()` to calculate the number of characters in the `pad` argument.

```
rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <-getOption("width") - nchar(title) - 5
  padding <- stringr::str_dup(pad,
    ceiling(width / stringr::str_length(title))) %>%
    stringr::str_trunc(width)
  cat(title, " ", padding, "\n", sep = "")
}
rule("Important output")
#> Important output -----
rule("Valuable output", pad = "-+")
#> Valuable output -+-----+
rule("Vital output", pad = "-+-")
#> Vital output -+-+-+-----+
```

Note that in the second output, there is only a single - at the end.

Exercise 19.5.3

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

The `trim` argument 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.

Exercise 19.5.4

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.

19.6 Return values

No Exercises

19.7 Environment

No Exercises

Chapter 20

Vectors

20.1 Introduction

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

20.2 Vector Basics

No exercises

20.3 Important Types of Atomic Vector

Exercise 20.3.1

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

To find out, try the functions on a numeric vector that includes at least one number and the four 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
```

The `is.finite()` function considers non-missing numeric values to be finite, and missing (`NA`), not a number (`NaN`), and positive (`Inf`) and negative infinity (`-Inf`) to not be finite. The `is.infinite()` behaves slightly differently. It considers `Inf` and `-Inf` to be infinite, and everything else, including non-missing numbers, `NA`, and `NaN` to not be infinite. See Table 20.1.

Table 20.1: Results of `is.finite()` and `is.infinite()` for numeric and special values.

| | <code>is.finite()</code> | <code>is.infinite()</code> |
|----|--------------------------|----------------------------|
| 1 | TRUE | FALSE |
| NA | FALSE | FALSE |

| | is.finite() | is.infinite() |
|-----|-------------|---------------|
| NaN | FALSE | FALSE |
| Inf | FALSE | TRUE |

Exercise 20.3.2

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
#> }
#> <bytecode: 0x7fdd581b1928>
#> <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.

Exercise 20.3.3

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.

For integers vectors, R uses a 32-bit representation. This means that it can represent up to 2^{32} different values with integers. One of these values is set aside for `NA_integer_`. From the help for `integer`.

Note that current implementations of R use 32-bit integers for integer vectors, so the range of representable integers is restricted to about $+/ - 2^{31}$: doubles can hold much larger integers exactly.

The range of integers values that R can represent in an integer vector is $\pm 2^{31} - 1$,

```
.Machine$integer.max
#> [1] 2147483647
```

The maximum integer is $2^{31} - 1$ rather than 2^{32} because 1 bit is used to represent the sign (+, -) and one value is used to represent `NA_integer_`.

If you try to represent an integer greater than that value, R will return `NA` values.

```
.Machine$integer.max + 1L
#> Warning in .Machine$integer.max + 1L: NAs produced by integer overflow
#> [1] NA
```

However, you can represent that value (exactly) with a numeric vector at the cost of about two times the memory.

```
as.numeric(.Machine$integer.max) + 1
#> [1] 2.15e+09
```

The same is true for the negative of the integer max.

```
-Machine$integer.max - 1L
#> Warning in -Machine$integer.max - 1L: NAs produced by integer overflow
#> [1] NA
```

For double vectors, R uses a 64-bit representation. This means that they can hold up to 2^{64} values exactly. However, some of those values are allocated to special values such as `-Inf`, `Inf`, `NA_real_`, and `NaN`. From the help for `double`:

All R platforms are required to work with values conforming to the IEC 60559 (also known as IEEE 754) standard. This basically works with a precision of 53 bits, and represents to that precision a range of absolute values from about $2e-308$ to $2e+308$. It also has special values `NaN` (many of them), plus and minus infinity and plus and minus zero (although R acts as if these are the same). There are also denormal(ized) (or subnormal) numbers with absolute values above or below the range given above but represented to less precision.

The details of floating point representation and arithmetic are complicated, beyond the scope of this question, and better discussed in the references provided below. The double can represent numbers in the range of about $\pm 2 \times 10^{308}$, which is provided in

```
.Machine$double.xmax
#> [1] 1.8e+308
```

Many other details for the implementation of the double vectors are given in the `.Machine` variable (and its documentation). These include the base (radix) of doubles,

```
.Machine$double.base
#> [1] 2
```

the number of bits used for the significand (mantissa),

```
.Machine$double.digits
#> [1] 53
```

the number of bits used in the exponent,

```
.Machine$double.exponent
#> [1] 11
```

and the smallest positive and negative numbers not equal to zero,

```
.Machine$double.eps
#> [1] 2.22e-16
.Machine$double.neg.eps
#> [1] 1.11e-16
```

- Computerphile, “Floating Point Numbers”
- https://en.wikipedia.org/wiki/IEEE_754
- https://en.wikipedia.org/wiki/Double-precision_floating-point_format
- “Floating Point Numbers: Why floating-point numbers are needed”
- Fabien Sanglard, “Floating Point Numbers: Visually Explained”
- James Howard, “How Many Floating Point Numbers are There?”
- GeeksforGeeks, “Floating Point Representation Basics”
- Chris Hecker, “Lets Go to the (Floating) Point”, *Game Developer*
- Chua Hock-Chuan, A Tutorial on Data Representation Integers, Floating-point Numbers, and Characters
- John D. Cook, “Anatomy of a floating point number”
- John D. Cook, “Five Tips for Floating Point Programming”

Exercise 20.3.4

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 $-100.5, -99.5, \dots, 0, \dots, 99.5, 100.5$. 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.

Exercise 20.3.5

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

The function `parse_logical()` parses logical values, which can appear as variations of TRUE/FALSE or 1/0.

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

The function `parse_integer()` parses integer values.

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

However, if there are any non-numeric characters in the string, including currency symbols, commas, and decimals, `parse_integer()` will raise an error.

```
parse_integer(c("1000", "$1,000", "10.00"))
#> Warning in rbind(names(probs), probs_f): number of columns of result is not
#> a multiple of vector length (arg 1)
#> Warning: 2 parsing failures.
#> row # A tibble: 2 x 4 col      row    col expected           actual
#> [1] 1000   NA   NA
#> attr(,"problems")
#> # A tibble: 2 x 4
#>   row    col expected           actual
#>   <int> <int> <chr>          <chr>
#> 1     2    NA an integer      $1,000
#> 2     3    NA no trailing characters .00
```

The function `parse_number()` parses integer values.

```
parse_number(c("1.0", "3.5", "$1,000.00", "NA"))
#> [1] 1.0 3.5 1000.0 NA
```

Unlike `parse_integer()`, the function `parse_number()` is very forgiving about the format of the numbers. It ignores all non-numeric characters, as with "\$1,000.00" in the example. This allows it to easily parse numeric fields that include currency symbols and comma separators in number strings without any intervention by the user.

20.4 Using atomic vectors

Exercise 20.4.1

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
```

Exercise 20.4.2

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
```

Exercise 20.4.3

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

The function `setNames()` takes two arguments, a vector to be named and a vector of names to apply to its elements.

```
setNames(1:4, c("a", "b", "c", "d"))
#> a b c d
#> 1 2 3 4
```

You can name an vector with itself if the `nm` argument is used.

```
setNames(nm = c("a", "b", "c", "d"))
#> a b c d
#> "a" "b" "c" "d"
```

The function `set_names()` has more ways to set the names than `setNames()`. The names can be specified in the same manner as `setNames()`.

```
purrr::set_names(1:4, c("a", "b", "c", "d"))
#> a b c d
#> 1 2 3 4
```

The names can also be specified as unnamed arguments,

```
purrr::set_names(1:4, "a", "b", "c", "d")
#> a b c d
#> 1 2 3 4
```

The function `set_names()` will name an object with itself if no `nm` argument is provided (the opposite of `setNames()` behavior).

```
purrr::set_names(c("a", "b", "c", "d"))
#> a b c d
#> "a" "b" "c" "d"
```

The biggest difference between `set_names()` and `setNames()` is that `set_names()` allows for using a function or formula to transform the existing names.

```
purrr::set_names(c(a = 1, b = 2, c = 3), toupper)
#> A B C
#> 1 2 3
purrr::set_names(c(a = 1, b = 2, c = 3), ~ toupper(.))
#> A B C
#> 1 2 3
```

The `set_names()` function also checks that the length of the `names` argument is the same length as the vector that is being named, and will raise an error if it is not.

```
purrr::set_names(1:4, c("a", "b"))
#> Error: `nm` must be `NULL` or a character vector the same length as `x`
```

The `setNames()` function will allow the names to be shorter than the vector being named, and will set the missing names to NA.

```
setNames(1:4, c("a", "b"))
#> a b <NA> <NA>
#> 1 2 3 4
```

Exercise 20.4.4

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.
3. Every element except the last value.
4. Only even numbers (and no missing values).

The answers to the parts follow.

1. This function find the last value in a vector.

```
last_value <- function(x) {
  # check for case with no length
  if (length(x)) {
    x[[length(x)]]
  } else {
    x
  }
}
last_value(numeric())
#> numeric(0)
```

```
last_value(1)
#> [1] 1
last_value(1:10)
#> [1] 10
```

The function uses `[[` in order to extract a single element.

2. This function returns the elements at even number positions.

```
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"
```

3. This function returns a vector with every element except the last.

```
not_last <- function(x) {
  n <- length(x)
  if (n) {
    x[-n]
  } else {
    # n == 0
    x
  }
}
not_last(1:3)
#> [1] 1 2
```

We should also confirm that the function works with some edge cases, like a vector with one element, and a vector with zero elements.

```
not_last(1)
#> numeric(0)
not_last(numeric())
#> numeric(0)
```

In both these cases, `not_last()` correctly returns an empty vector.

4. This function returns the elements of a vector that are even numbers.

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

We could improve this function by handling the cases of the special values: `NA`, `NaN`, `Inf`. However, first we need to decide how to handle them. Neither `NaN` nor `Inf` are considered even numbers. What about `NA`? Well, we don't know. The value of `NA` could be even or odd, but it is missing. So we will follow the convention of many R functions and keep the `NA` values. The revised function now handles these cases.

```
even_numbers <- function(x) {
  x[!is.infinite(x) & !is.nan(x) & (x %% 2 == 0)]
}
even_numbers(c(0:4, NA, NaN, Inf))
#> [1] 0 2 4 NA
```

Exercise 20.4.5

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

These expressions differ in the way that they treat missing values. Let's test how they work by creating a vector with positive and negative integers, and special values (`NA`, `NaN`, and `Inf`). These values should encompass all relevant types of values that these expressions would encounter.

```
x <- c(-1:1, Inf, -Inf, NaN, NA)
x[-which(x > 0)]
#> [1] -1 0 -Inf NaN NA
x[x <= 0]
#> [1] -1 0 -Inf NA NA
```

The expressions `x[-which(x > 0)]` and `x[x <= 0]` return the same values except for a `NaN` instead of a `NA` in the `which()` based expression.

So what is going on here? Let's work through each part of these expressions and see where the difference occurs. Let's start with the expression `x[x <= 0]`.

```
x <= 0
#> [1] TRUE TRUE FALSE FALSE TRUE NA NA
```

Recall how the logical relational operators (`<`, `<=`, `==`, `!=`, `>`, `>=`) treat `NA` values. Any relational operation that includes a `NA` returns an `NA`. Is `NA <= 0`? We don't know because it depends on the unknown value of `NA`, so the answer is `NA`. This same argument applies to `NaN`. Asking whether `NaN <= 0` does not make sense because you can't compare a number to "Not a Number".

Now recall how indexing treats `NA` values. Indexing can use a logical vector, and will include those elements where the logical vector is `TRUE`, and will not return those elements where the logical vector is `FALSE`. Since a logical vector can include `NA` values, what should it do for them? Well, since the value is `NA` it could be `TRUE` or `FALSE`, we don't know. Keeping elements with `NA` would treat the `NA` as `TRUE`, and dropping them would treat the `NA` as `FALSE`.

The way R decides to handle the `NA` values so that they are treated differently than `TRUE` or `FALSE` values is to include elements where the indexing vector is `NA`, but set their values to `NA`.

Now consider the expression `x[-which(x > 0)]`. As before, to understand this expression we'll work from the inside out. Consider `x > 0`.

```
x > 0
#> [1] FALSE FALSE TRUE TRUE FALSE NA NA
```

As with `x <= 0`, it returns `NA` for comparisons involving `NA` and `NaN`.

What does `which()` do?

```
which(x > 0)
#> [1] 3 4
```

The `which()` function returns the indexes for which the argument is TRUE. This means that it is not including the indexes for which the argument is FALSE or NA.

Now consider the full expression `x[-which(x > 0)]`? The `which()` function returned a vector of integers. How does indexing treat negative integers?

```
x[1:2]
#> [1] -1  0
x[-(1:2)]
#> [1]    1  Inf -Inf  NaN  NA
```

If indexing gets a vector of positive integers, it will select those indexes; if it receives a vector of negative integers, it will drop those indexes. Thus, `x[-which(x > 0)]` ends up dropping the elements for which `x > 0` is true, and keeps all the other elements and their original values, including NA and NaN.

There's one other special case that we should consider. How do these two expressions work with an empty vector?

```
x <- numeric()
x[x <= 0]
#> numeric(0)
x[-which(x > 0)]
#> numeric(0)
```

Thankfully, they both handle empty vectors the same.

This exercise is a reminder to always test your code. Even though these two expressions looked equivalent, they are not in practice. And when you do test code, consider both how it works on typical values as well as special values and edge cases, like a vector with NA or NaN or Inf values, or an empty vector. These are where unexpected behavior is most likely to occur.

Exercise 20.4.6

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?

Let's consider the named vector,

```
x <- c(a = 10, b = 20)
```

If we subset it by an integer larger than its length, it returns a vector of missing values.

```
x[3]
#> <NA>
#>    NA
```

This also applies to ranges.

```
x[3:5]
#> <NA> <NA> <NA>
#>    NA    NA    NA
```

If some indexes are larger than the length of the vector, those elements are NA.

```
x[1:5]
#>     a     b <NA> <NA> <NA>
#> 10   20    NA    NA    NA
```

Likewise, when [is provided names not in the vector's names, it will return `NA` for those elements.

```
x["c"]
#> <NA>
#> NA
x[c("c", "d", "e")]
#> <NA> <NA> <NA>
#> NA NA NA
x[c("a", "b", "c")]
#>     a     b <NA>
#> 10   20    NA
```

Though not yet discussed much in this chapter, the [[behaves differently. With an atomic vector, if [[is given an index outside the range of the vector or an invalid name, it raises an error.

```
x[["c"]]
#> Error in x[["c"]]: subscript out of bounds

x[[5]]
#> Error in x[[5]]: subscript out of bounds
```

20.5 Recursive Vectors (lists)

Exercise 20.5.1

Draw the following lists as nested sets:

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

There are a variety of ways to draw these graphs. The original diagrams in *R for Data Science* were produced with Graffle. You could also use various diagramming, drawing, or presentation software, including Adobe Illustrator, Inkscape, PowerPoint, Keynote, and Google Slides.

For these examples, I generated these diagrams programmatically using the DiagrammeR R package to render Graphviz diagrams.

1. The nested set diagram for `list(a, b, list(c, d), list(e, f))` is
2. The nested set diagram for `list(list(list(list(list(a))))))` is as follows.

Exercise 20.5.2

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 difference between a list and a `tibble` is that all the elements (columns) of a tibble must have the same length (number of rows). Lists can have vectors with different lengths as elements.

```
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
```

20.6 Attributes

No exercises

20.7 Augmented Vectors

Exercise 20.7.1

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

Exercise 20.7.2

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

If I try to create a 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
```

Exercise 20.7.3

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.

⊜ double-rounding: The built-in variable `.Machine$double.rounding` indicates the rounding method used by R. It states that the round half to even method is expected to be used, but this may differ by operating system.

Chapter 21

Iteration

21.1 Introduction

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

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

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

21.2 For Loops

Exercise 21.2.1

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 .

The answers for each part are below.

1. 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
```

2. 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.

```
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"
```

3. To compute the number of unique values in each column of the `iris` dataset.

```
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
```

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

```
# 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()` recycle the `mean` argument.

```
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
```

Exercise 21.2.2

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"
```

Since `str_c()` already works with vectors, 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))

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
```

Exercise 21.2.3

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. Generalize it to any number of people in any sleeping structure.
3. Convert the song "99 bottles of beer on the wall" to a function. Generalize to any number of any vessel containing any liquid on surface.

The answers to each part follow.

1. The lyrics for Alice the Camel are:

```
Alice the camel has five humps.
Alice the camel has five humps.
Alice the camel has five humps.
So go, Alice, go.
```

This verse is repeated, each time with one fewer hump, until there are no humps. The last verse, with no humps, is:

```
Alice the camel has no humps.
Alice the camel has no humps.
Alice the camel has no humps.
Now Alice is a horse.
```

We'll iterate from five to no humps, and print out a different last line if there are no humps.

```
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.
```

2. The lyrics for Ten in the Bed are:

Here we go!
 There were ten in the bed
 and the little one said,
 “Roll over, roll over.”
 So they all rolled over and one fell out.

This verse is repeated, each time with one fewer in the bed, until there is one left. That last verse is:

One! There was one in the bed
 and the little one said,
 “I'm lonely...”

```
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...

```

3. The lyrics of Ninety-Nine Bottles of Beer on the Wall are

99 bottles of beer on the wall, 99 bottles of beer.
Take one down, pass it around, 98 bottles of beer on the wall

This verse is repeated, each time with one fewer bottle, until there are no more bottles of beer. The last verse is

No more bottles of beer on the wall, no more bottles of beer.
 We've taken them down and passed them around; now we're drunk and passed out!

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.
```

Exercise 21.2.4

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
```

How does this affect performance? Design and execute an experiment.

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) 209 209   214    209 216 223     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) 668 2663 4705   4657 6723 8789     3

```

The pre-allocated vector is about **10** 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.

21.3 For loop variations

Exercise 21.3.1

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)

```

Exercise 21.3.2

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
```

Exercise 21.3.3

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 all variable names 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
```

Exercise 21.3.4

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"]])
```

21.4 For loops vs. functionals

Exercise 21.4.1

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.

Exercise 21.4.2

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 column 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
```

21.5 The map functions

Exercise 21.5.1

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
```

Exercise 21.5.2

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
```

Exercise 21.5.3

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
```

Exercise 21.5.4

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
```

Exercise 21.5.5

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
```

21.6 Dealing with Failure

No exercises

21.7 Mapping over multiple arguments

No exercises

21.8 Walk

No exercises

21.9 Other patterns of for loops

Exercise 21.9.1

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 them 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.

Exercise 21.9.2

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
```

Exercise 21.9.3

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 22

Introduction

Chapter 23

Model basics

23.1 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. If it is not set (the default), R will silently drop them.

23.2 A simple model

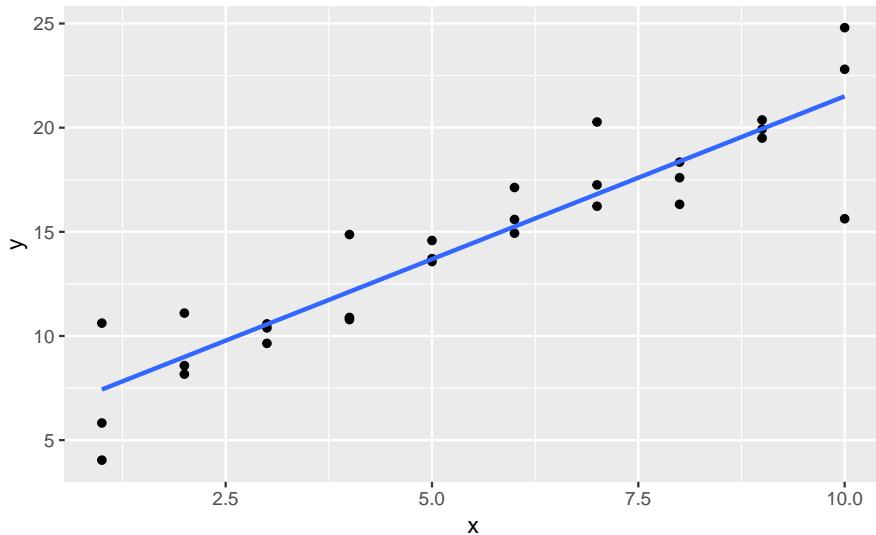
Exercise 23.2.1

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)
```

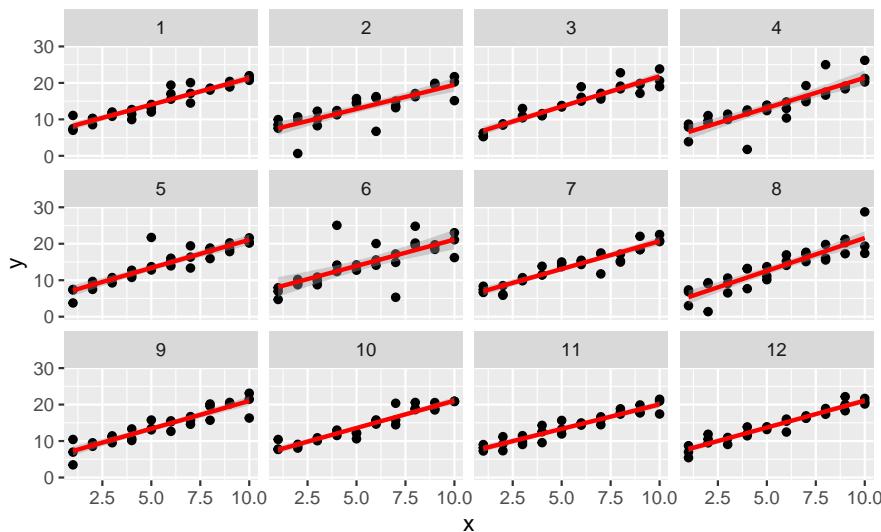


We can also do this more systematically, by generating several simulations and plotting the line.

```
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", colour = "red") +
  facet_wrap(~ .id, ncol = 4)
```



What if we did the same things with normal distributions?

```
sim_norm <- function(i) {
  tibble(
```

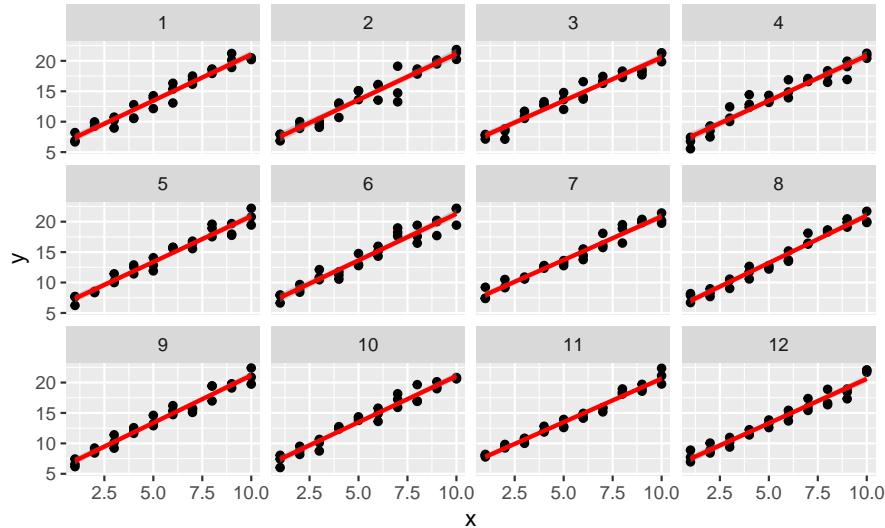
```

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", colour = "red") +
  facet_wrap(~ .id, ncol = 4)

```



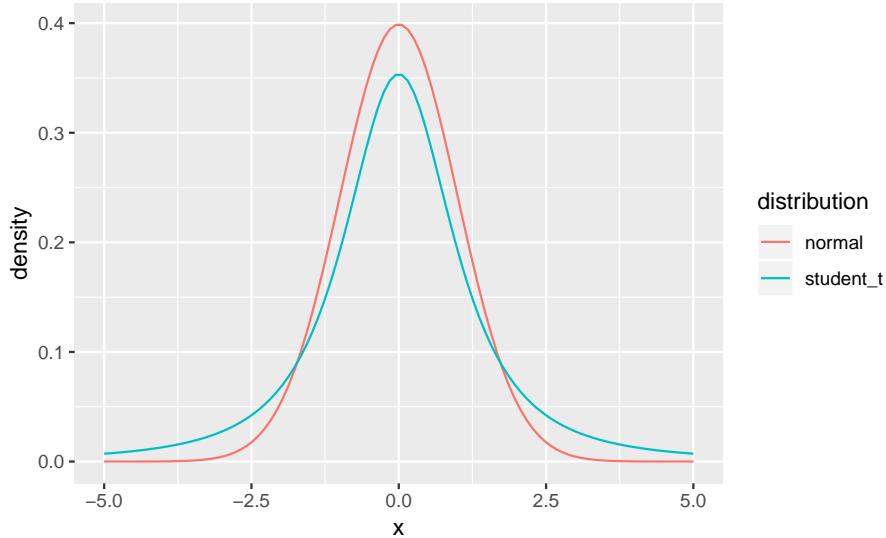
There are not large outliers, and the slopes are more similar.

The reason for this is that the Student's t -distribution, from which we sample with `rt()` has heavier tails than the normal distribution (`rnorm()`). This means that the Student's t -distribution assigns a 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
```

Exercise 23.2.2

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 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 use an existing implementation. See the **MASS** package's `rlm()` and `lqs()` functions for more information and functions to fit robust and resistant linear models.

Exercise 23.2.3

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 $a[1] = a_1$ and $a[3] = a_3$, any other values of $a[1]$ and $a[3]$ where $a[1] + a[3] == (a_1 + a_3)$ 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.

23.3 Visualizing Models

Exercise 23.3.1

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.

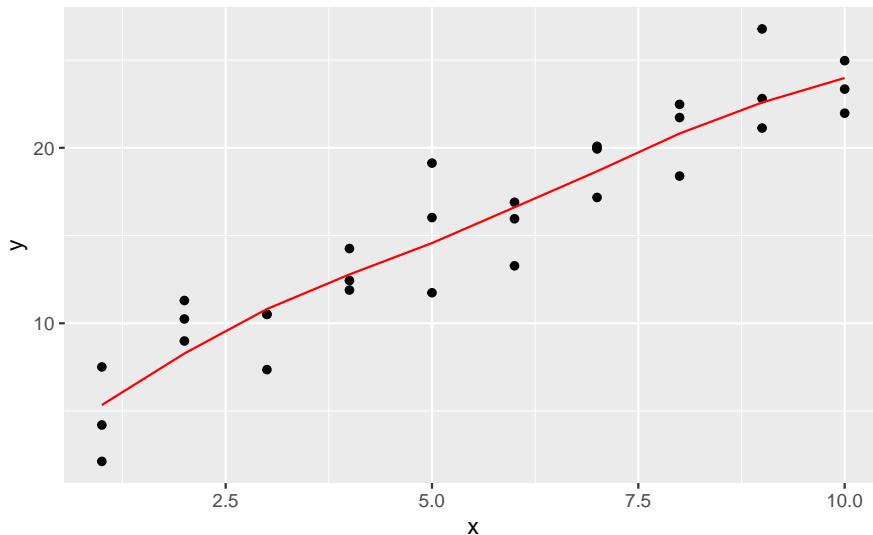
```
sim1_loess <- loess(y ~ x, data = sim1)
sim1_lm <- lm(y ~ x, data = sim1)
```

```
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")
```

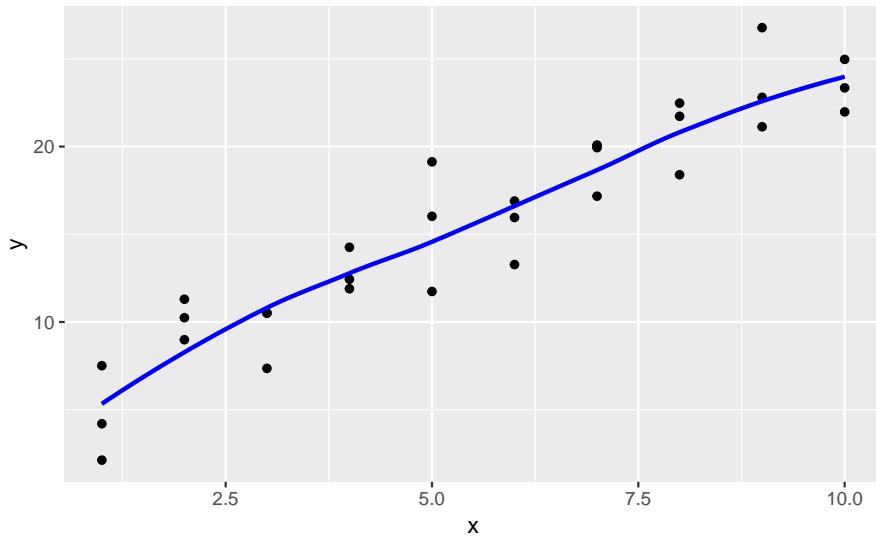
This plots the loess predictions. The loess produces a nonlinear, smooth line through the data.

```
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
```



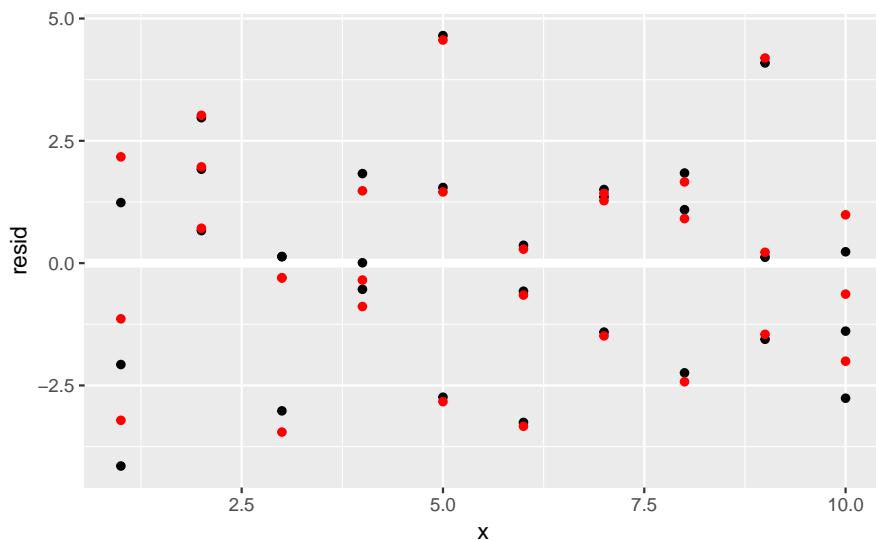
The predictions of loess are the same as the default method for `geom_smooth()` because `geom_smooth()` uses `loess()` by default; the message even tells us that.

```
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")
```



Exercise 23.3.2

`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
#>   <dbl>      <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>   <dbl>   <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
#>   <dbl>      <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
```

Exercise 23.3.3

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.

Exercise 23.3.4

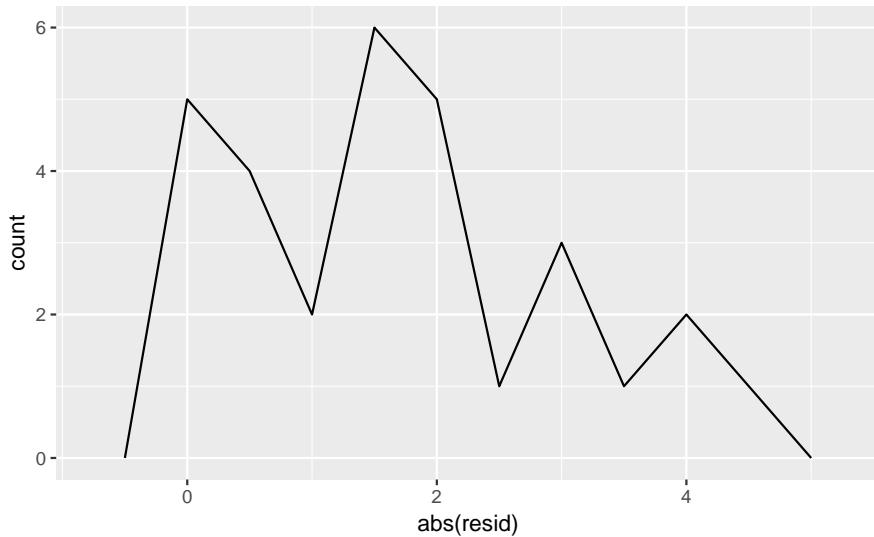
Why might you want to look at a frequency polygon of absolute residuals? What are the pros and cons compared to looking at the raw residuals?

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.

23.4 Formulas and Model Families

Exercise 23.4.1

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 of 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
```

Exercise 23.4.2

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 $x_1 * x_2$ when x_2 is a categorical variable produces indicator variables x_{2b} , x_{2c} , x_{2d} and variables $x_1:x_{2b}$, $x_1:x_{2c}$, and $x_1:x_{2d}$ which are the products of x_1 and x_2 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 $x_1:x_{2b}$ is the product of x_1 and x_{2b} ,

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

and similarly for $x_1:x_{2c}$ and x_{2c} , and $x_1:x_{2d}$ and x_{2d} :

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

For $x_1 * x_2$ where both x_1 and x_2 are continuous variables, `model_matrix()` creates variables x_1 , x_2 , and $x_1:x_2$:

```
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     1
#> 2     1    -1   -1     1
#> 3     1    -1   -1     1
#> 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 $x_1:x_2$ is the product of the x_1 and x_2 ,

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

The asterisk $*$ is good shorthand for an interaction since an interaction between x_1 and x_2 includes terms for x_1 , x_2 , and the product of x_1 and x_2 .

Exercise 23.4.3

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      1
#> 2    -1    -1      1
#> 3    -1    -1      1
#> 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)
  x2lvs <- levels(x2)
  # create an indicator variable for each level
  for (lvl in x2lvs[2:nlevels(x2)]) {
    out[[str_c("x2", lvl)]] <- as.numeric(x2 == lvl)
  }
  # create interactions for each level
  for (lvl in x2lvs[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)
}

```

Exercise 23.4.4

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,

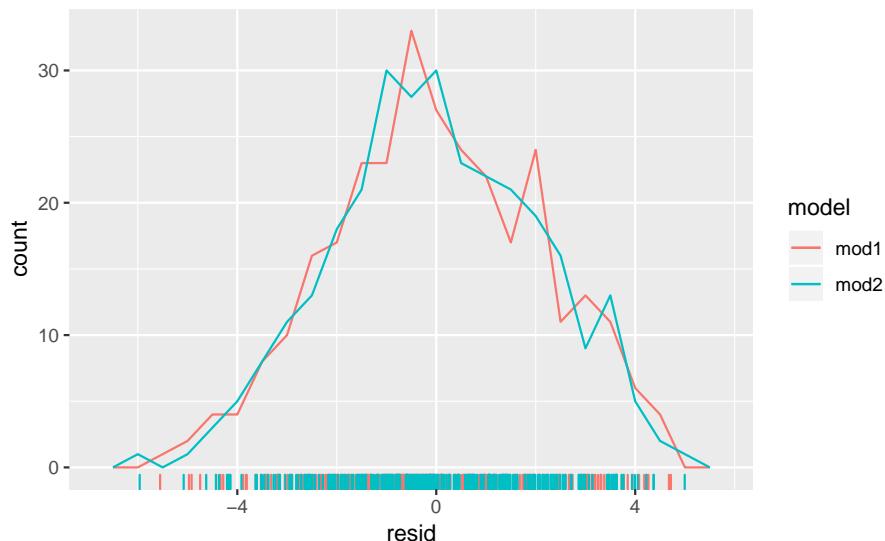
```
sim4_mods <- gather_residuals(sim4, mod1, mod2)
```

Frequency plots of both the residuals,

```

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

```

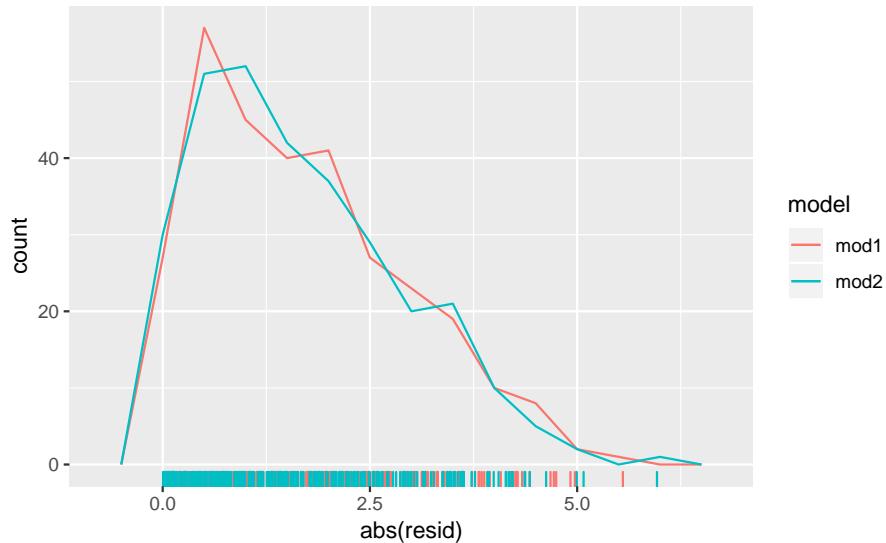


and the absolute values of the residuals,

```

ggplot(sim4_mods, aes(x = abs(resid), colour = 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`.

23.5 Missing values

No exercises

23.6 Other model families

No exercises

Chapter 24

Model building

24.1 Introduction

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

library(nycflights13)
library(lubridate)
```

24.2 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)
```

Exercise 24.2.1

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).

Exercise 24.2.2

If $\log(\text{price}) = a_0 + a_1 * \log(\text{carat})$, 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.

Exercise 24.2.3

Extract the diamonds that have very high and very low residuals. Is there anything unusual about these diamonds? Are they 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.

Exercise 24.2.4

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.

24.3 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)

```

Exercise 24.3.1

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.

Exercise 24.3.2

```

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

```

Exercise 24.3.3

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)

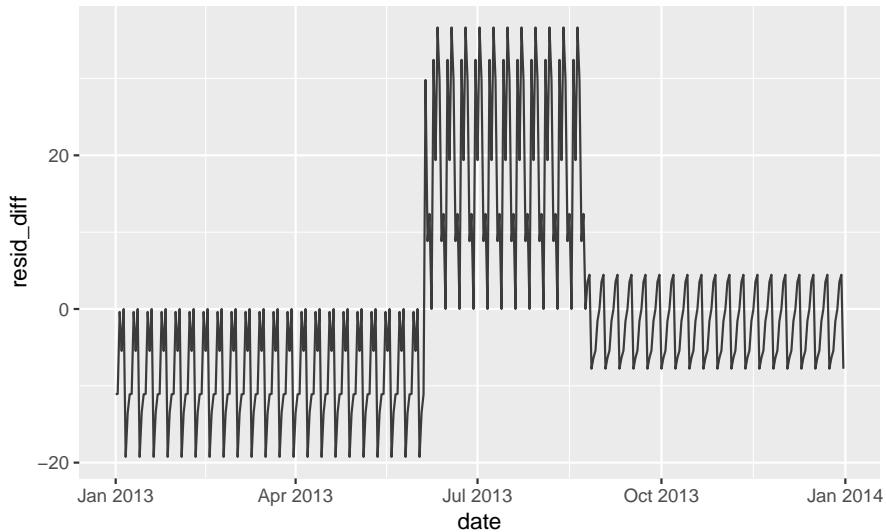
```

```
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)
#> # A tibble: 1 x 4
#>   r.squared sigma    AIC     df
#> *     <dbl> <dbl> <dbl> <int>
#> 1     0.736  47.4 3863.     9

glance(mod2) %>% select(r.squared, sigma, AIC, df)
#> # A tibble: 1 x 4
#>   r.squared sigma    AIC     df
#> *     <dbl> <dbl> <dbl> <int>
#> 1     0.757  46.2 3856.    21
```

Exercise 24.3.4

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
```

Exercise 24.3.5

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.

Exercise 24.3.6

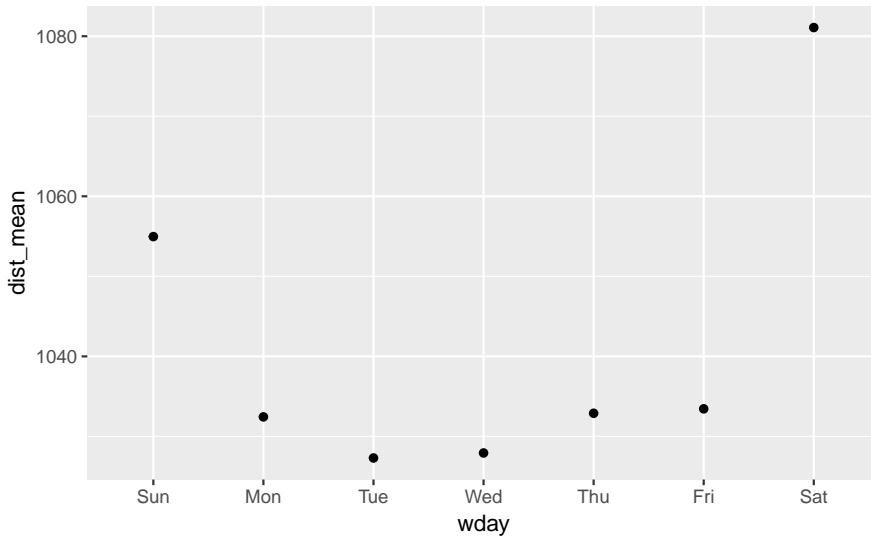
It will estimate a smooth seasonal trend (`ns(date, 5)`) with a day of the week cyclicalty, (`wday`). It probably will not be effective since

Exercise 24.3.7

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.

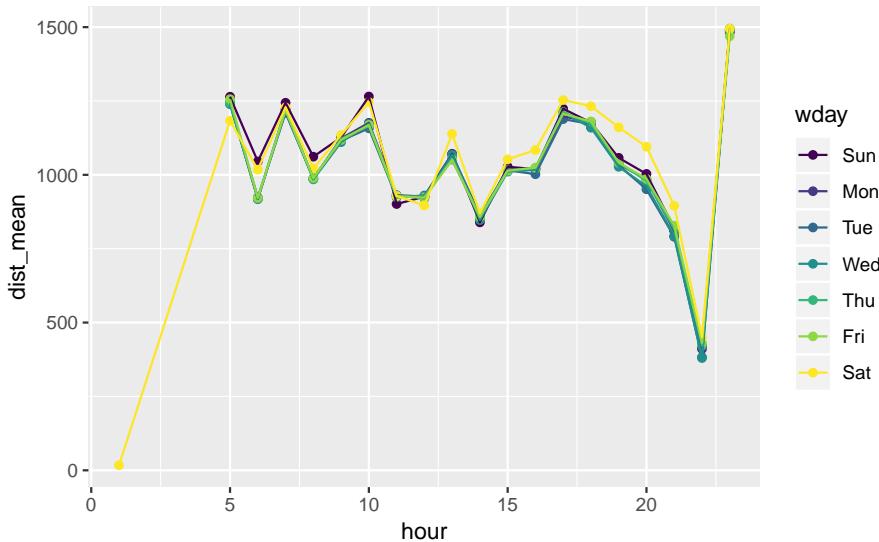
Looking at only day of the week, we see that Sunday flights are on average longer than the rest of the day of the week flights, but not as long as Saturday flights (perhaps vacation flights?).

```
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?

Exercise 24.3.8

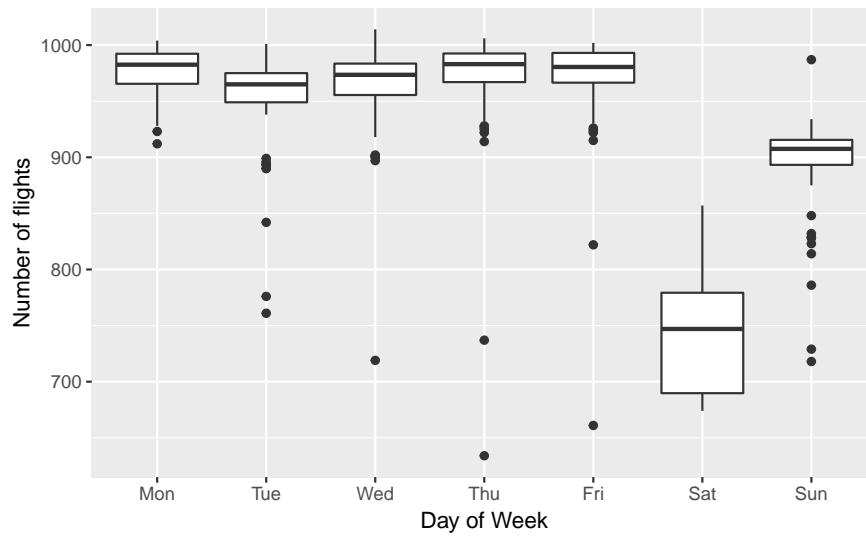
It's a little frustrating that Sunday and Saturday are on separate ends of the plot. Write a small function to set the levels of the factor so that the week starts on Monday.

See the chapter Factors for the function `fct_relevel()`. 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")
```



24.4 Learning more about models

No exercises

Chapter 25

Many models

25.1 Introduction

```
library("modelr")
library("tidyverse")
library("gapminder")
```

25.2 Gapminder

Exercise 25.2.1

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)
}

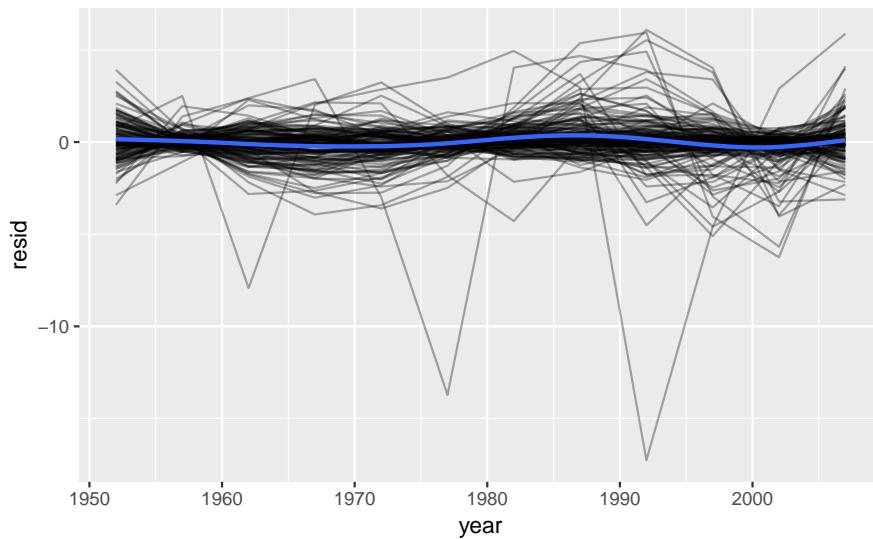
by_country <- gapminder %>%
  group_by(country, continent) %>%
  nest()

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

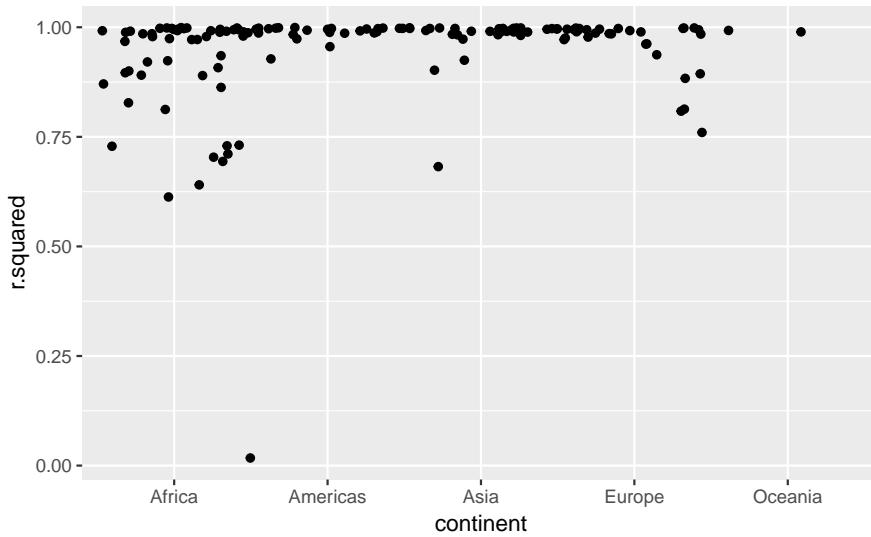
by_country <- by_country %>%
  mutate(
    resids = map2(data, model, add_residuals)
  )
by_country
#> # A tibble: 142 x 5
```

```
#>   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' and formula 'y ~ s(x, bs = "cs")'
```



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

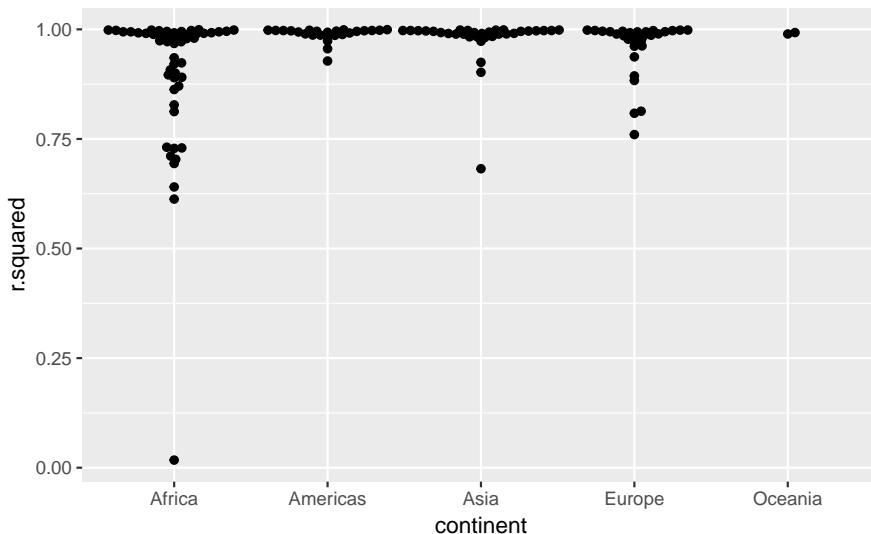


Exercise 25.2.2

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()
```



25.3 List-columns

No exercises

25.4 Creating list-columns

Exercise 25.4.1

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.

Exercise 25.4.2

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

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

```
range(mtcars$mpg)
#> [1] 10.4 33.9
fivenum(mtcars$mpg)
#> [1] 10.4 15.3 19.2 22.8 33.9
```

Exercise 25.4.3

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
#> 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.

Exercise 25.4.4

What does this code do? Why 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 `fun` 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> <list> <list> <list> <list> <list>
#> 1     4 <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~
#> 2     6 <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~ <dbl> ~
#> 3     8 <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.

25.5 Simplifying list-columns

Exercise 25.5.1

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)`.

Exercise 25.5.2

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 26

Introduction

Chapter 27

R Markdown

27.1 Introduction

27.2 R Markdown Basics

Exercise 27.2.1

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.

Exercise 27.2.2

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 + K**.

Exercise 27.2.3

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,

```
---
```

```
output: 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,

```
output: 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 notebook, and vice-versa. More specifically, an `.Rmd` file can be changed to R markdown file or R notebook by changing the value of the `output` key in the header.

The RStudio IDE and the `rmarkdown` package both use the YAML header of an `.Rmd` file to determine the document-type of the file.

Exercise 27.2.4

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.

27.3 Text formatting with R Markdown

Exercise 27.3.1

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.

A minimal example is the following CV.

```
---
title: "Hadley Wickham"
---

## Employment

- Chief Scientist, Rstudio, **2013--present**.
```

- Adjunct Professor, Rice University, Houston, TX, **2013--present**.
 - Assistant Professor, Rice University, Houston, TX, **2008--12**.
- ## Education
- Ph.D. in Statistics, Iowa State University, Ames, IA, **2008**
 - M.Sc. in Statistics, University of Auckland, New Zealand, **2004**
 - B.Sc. in Statistics and Computer Science, First Class Honours, The University of Auckland, New Zealand, **2002**.
 - Bachelor of Human Biology, First Class Honours, The University of Auckland, Auckland, New Zealand, **1999**.

Your own example could be much more detailed.

Exercise 27.3.2

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

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

```
---
title: Horizontal Rules, Block Quotes, and Footnotes
---
```

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.

Exercise 27.3.3

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.

The following R markdown document answers this question as well as exercises [exercise-27.4.1], [exercise-27.4.2], and [exercise-27.4.3].

```
---
title: "Diamond sizes"
date: 2018-07-15
output: html_document
---

```{r knitr_opts, include = FALSE}
knitr::opts_chunk$set(echo = FALSE)
```

```{r setup, message = FALSE}
library("ggplot2")
library("dplyr")

smaller <- diamonds %>%
 filter(carat <= 2.5)
```

```{r include = FALSE}
Hide objects and functions ONLY used inline
n_larger <- nrow(diamonds) - nrow(smaller)
pct_larger <- n_larger / nrow(diamonds) * 100

comma <- function(x) {
 format(x, digits = 2, big.mark = ",")
}
```

## Size and Cut, Color, and Clarity

Diamonds with lower quality cuts (cuts are ranked from "Ideal" to "Fair") tend to be larger.

```{r}
ggplot(diamonds, aes(y = carat, x = cut)) +
 geom_boxplot()
```

Likewise, diamonds with worse color (diamond colors are ranked from J (worst) to D (best)) tend to be larger:

```{r}
ggplot(diamonds, aes(y = carat, x = color)) +
 geom_boxplot()
```

The pattern present in cut and color is also present in clarity. Diamonds with worse clarity (I1 (worst), SI1, SI2, VS1, VS2, VVS1, VVS2, IF (best)) tend to be larger.

```{r}
ggplot(diamonds, aes(y = carat, x = clarity)) +
 geom_boxplot()
```

These patterns are consistent with there being a profitability threshold for
```

retail diamonds that is a function of carat, clarity, color, cut and other characteristics. A diamond may be profitable to sell if a poor value of one feature, for example, poor clarity, color, or cut, is be offset by a good value of another feature, such as a large size. This can be considered an example of [Berkson's paradox] (https://en.wikipedia.org/wiki/Berkson%27s_paradox).

```
## Largest Diamonds
```

We have data about `r comma(nrow(diamonds))` diamonds. Only `r n_larger` (`r round(pct_larger, 1)`%) are larger than 2.5 carats. The distribution of the remainder is shown below:

```
```{r}
smaller %>%
 ggplot(aes(carat)) +
 geom_freqpoly(binwidth = 0.01)
```
```

The frequency distribution of diamond sizes is marked by spikes at whole-number and half-carat values, as well as several other carat values corresponding to fractions.

The largest twenty diamonds (by carat) in the datasets are,

```
```{r results = "asis"}
diamonds %>%
 arrange(desc(carat)) %>%
 slice(1:20) %>%
 select(carat, cut, color, clarity) %>%
 knitr::kable(
 caption = "The largest 20 diamonds in the `diamonds` dataset."
)
```
```

Most of the twenty largest datasets are in the lowest clarity category ("I1"), with one being in the second best category ("VVS2") The top twenty diamonds have colors ranging from the worst, "J", to best, "D", categories, though most are in the lower categories "J" and "I". The top twenty diamonds are more evenly distributed among the cut categories, from "Fair" to "Ideal", although the worst category (Fair) is the most common.

27.4 Code Chunks

Exercise 27.4.1

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.

See the answer to [exercise-27.3.3].

Exercise 27.4.2

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.

See the answer to [exercise-27.3.3]. I use `arrange()` and `slice()` to select the largest twenty diamonds, and `knitr::kable()` to produce a formatted table.

Exercise 27.4.3

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.

See the answer to Exercise [exercise-27.3.3].

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.

Exercise 27.4.4

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.

```
---
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 chunks, and the same as the first time the document was run with caching.

## 27.5 YAML header

No exercises

## 27.6 Learning more

No exercises



# Chapter 28

## Graphics for communication

### 28.1 Introduction

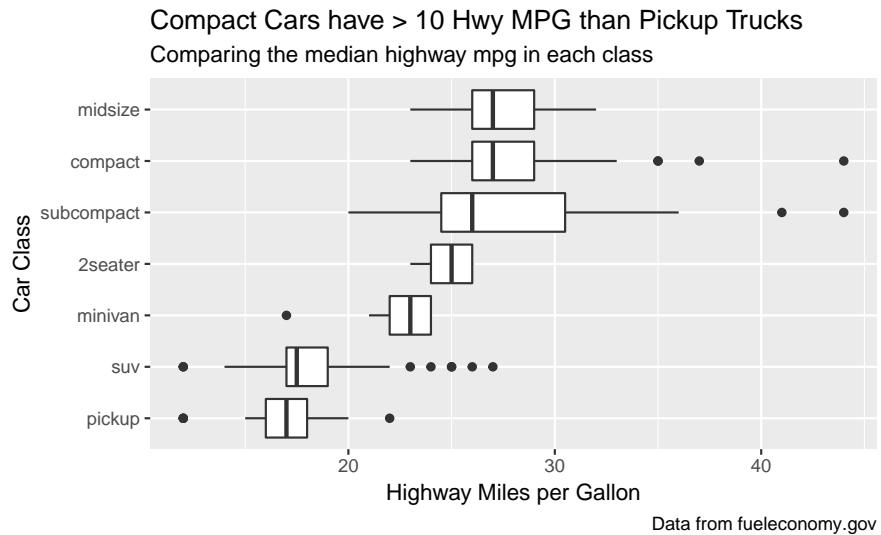
```
library("tidyverse")
library("modelr")
library("lubridate")
```

### 28.2 Label

#### Exercise 28.2.1

Create one plot on the fuel economy data with customized `title`, `subtitle`, `caption`, `x`, `y`, and `colour` labels.

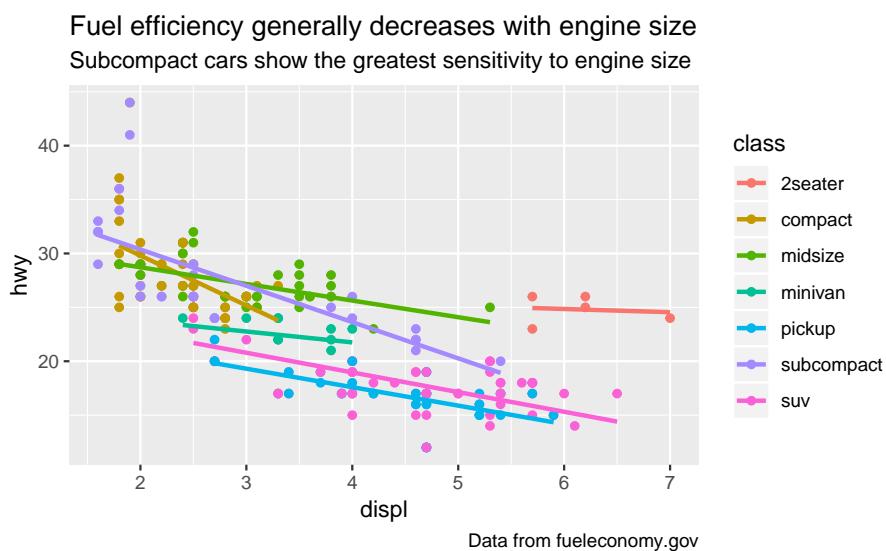
```
ggplot(data = mpg,
 mapping = aes(x = reorder(class, hwy, median), y = hwy)) +
 geom_boxplot() +
 coord_flip() +
 labs(
 title = "Compact Cars have > 10 Hwy MPG than Pickup Trucks",
 subtitle = "Comparing the median highway mpg in each class",
 caption = "Data from fueleconomy.gov",
 x = "Car Class",
 y = "Highway Miles per Gallon"
)
```



### Exercise 28.2.2

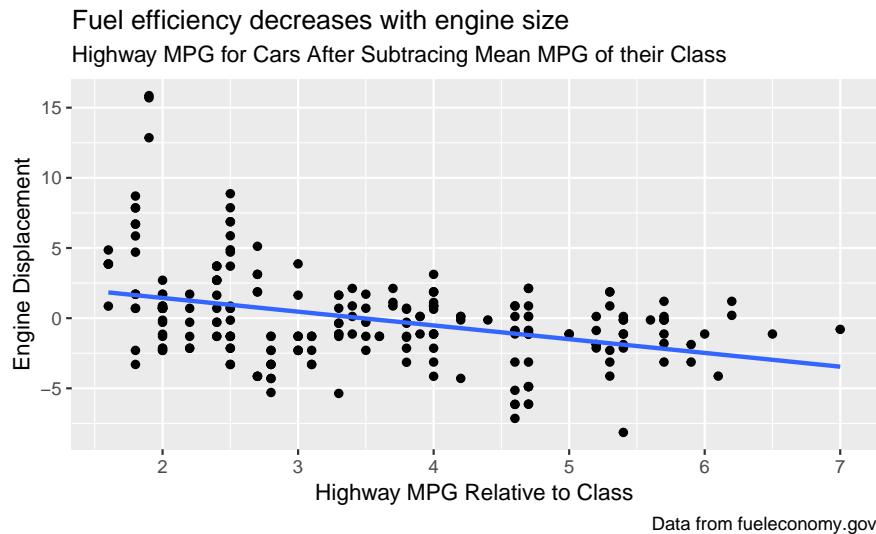
The `geom_smooth()` is somewhat misleading because the `hwy` for large engines is skewed upwards due to the inclusion of lightweight sports cars with big engines. Use your modeling tools to fit and display a better model.

```
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 Subtracting Mean MPG of their Class",
 caption = "Data from fueleconomy.gov",
 x = "Highway MPG Relative to Class",
 y = "Engine Displacement"
)
```



### Exercise 28.2.3

Take an exploratory graphic that you've created in the last month, and add informative titles to make it easier for others to understand.

By its very nature, this exercise is left to readers.

## 28.3 Annotations

### Exercise 28.3.1

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.

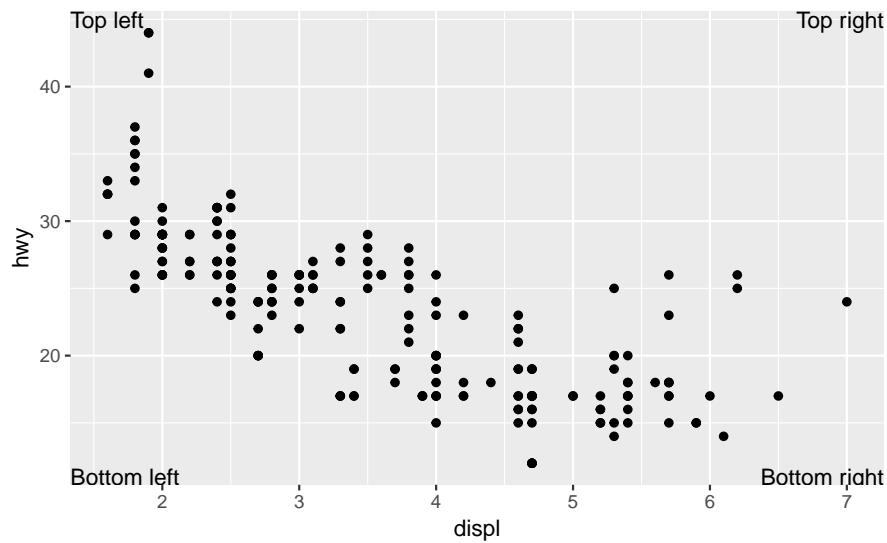
```
label <- tribble(
 ~displ, ~hwy, ~label, ~vjust, ~hjust,
 Inf, Inf, "Top right", "top", "right",
 Inf, -Inf, "Bottom right", "bottom", "right",
 -Inf, Inf, "Top left", "top", "left",
```

```

-Inf, -Inf, "Bottom left", "bottom", "left"
)

ggplot(mpg, aes(displ, hwy)) +
 geom_point() +
 geom_text(aes(label = label, vjust = vjust, hjust = hjust), data = label)

```



### Exercise 28.3.2

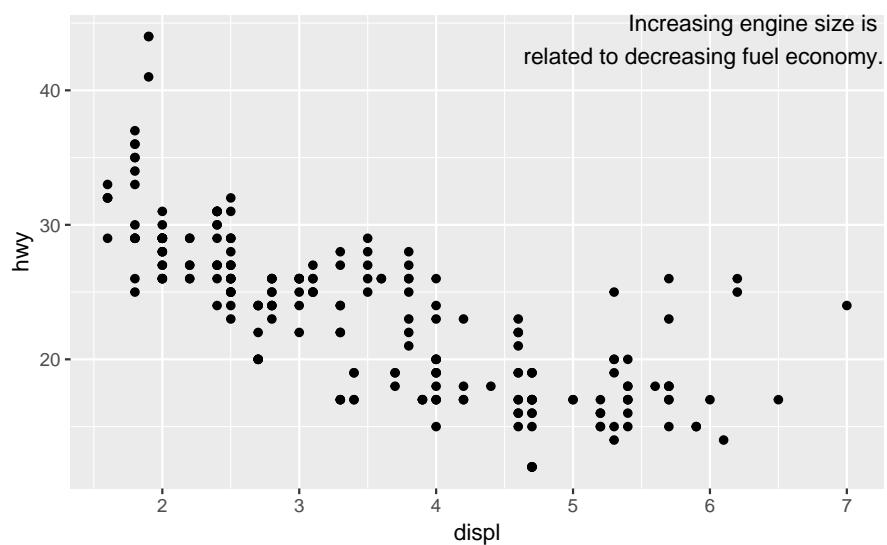
Read the documentation for `annotate()`. How can you use it to add a text label to a plot without having to create a tibble?

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", hjust = "right")

```



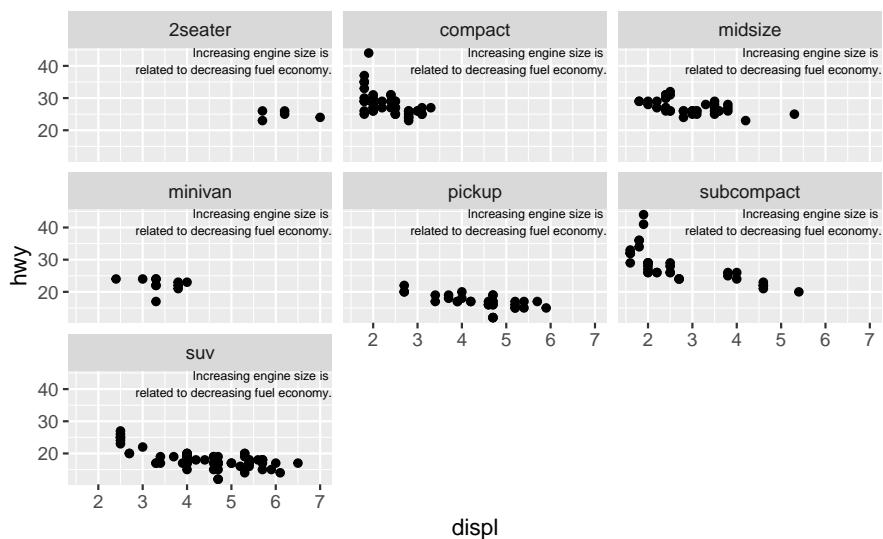
### Exercise 28.3.3

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."
)

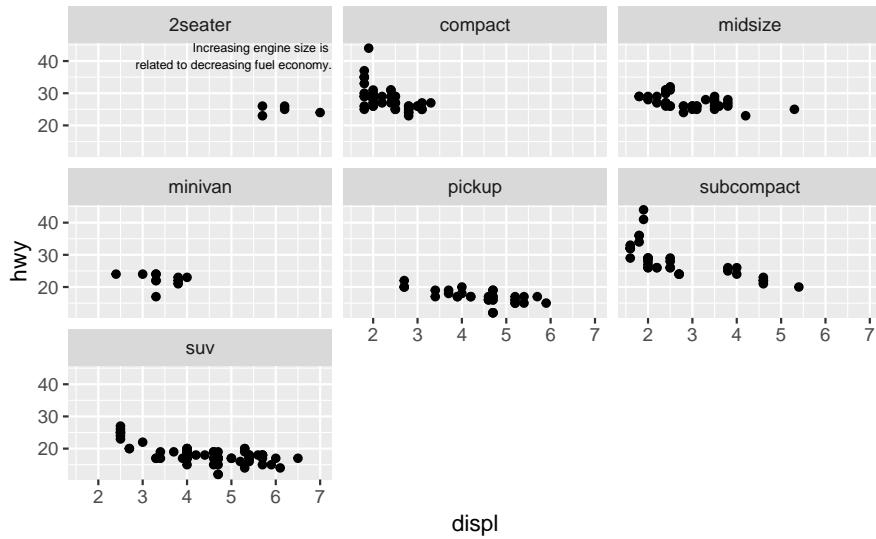
ggplot(mpg, aes(displ, hwy)) +
 geom_point() +
 geom_text(aes(label = label), data = label, vjust = "top", hjust = "right",
 size = 2) +
 facet_wrap(~ class)
```



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."
)

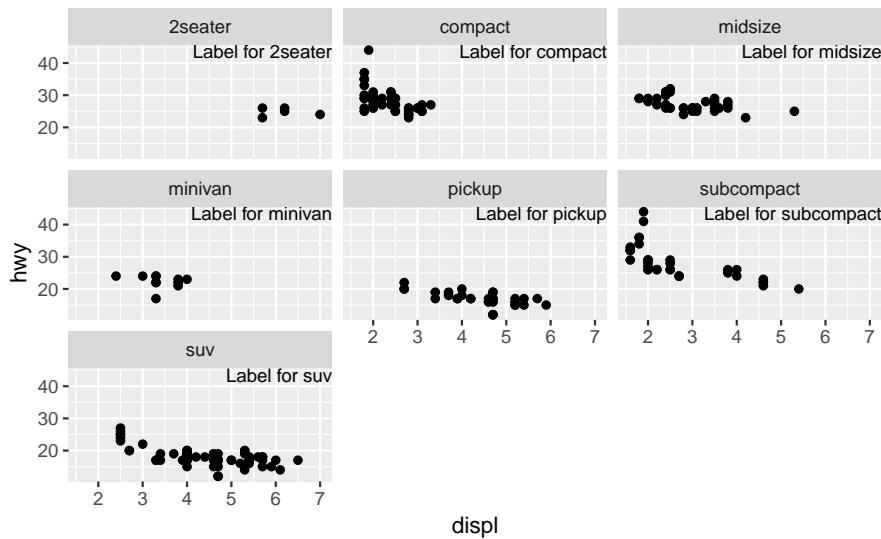
ggplot(mpg, aes(displ, hwy)) +
 geom_point() +
 geom_text(aes(label = label), data = label, vjust = "top", hjust = "right",
 size = 2) +
 facet_wrap(~ class)
```



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)
```



### Exercise 28.3.4

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

### Exercise 28.3.5

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)

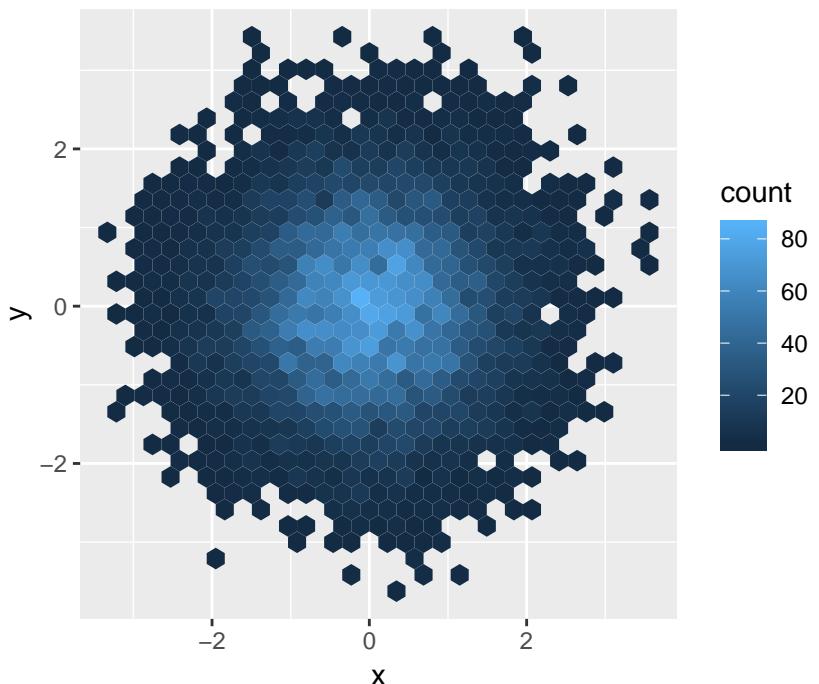
- `angle` : angle of arrow head
- `length` : length of the arrow head
- `ends`: ends of the line to draw arrow head
- `type`: "open" or "close": whether the arrow head is a closed or open triangle

## 28.4 Scales

### Exercise 28.4.1

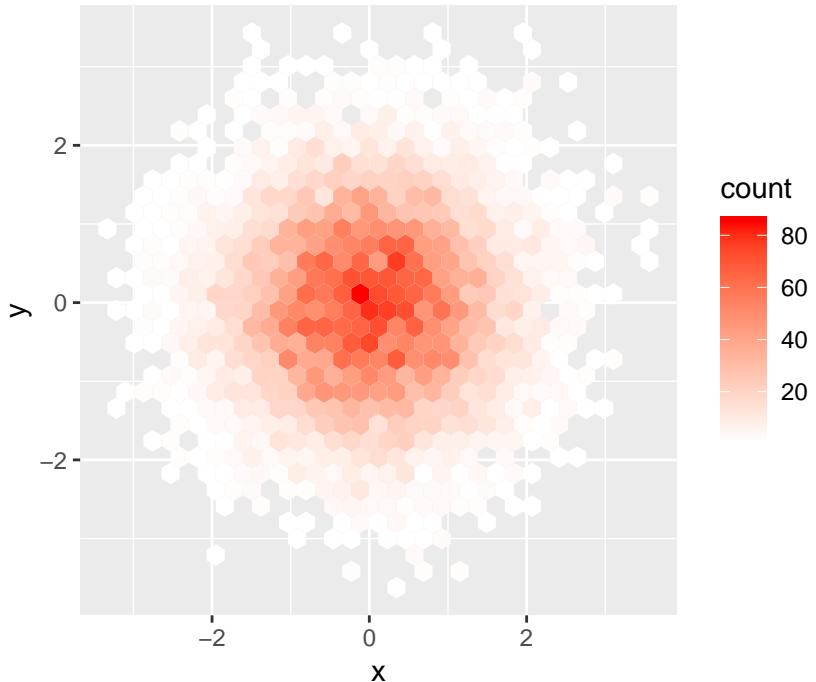
Why doesn't the following code override the default scale?

```
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.

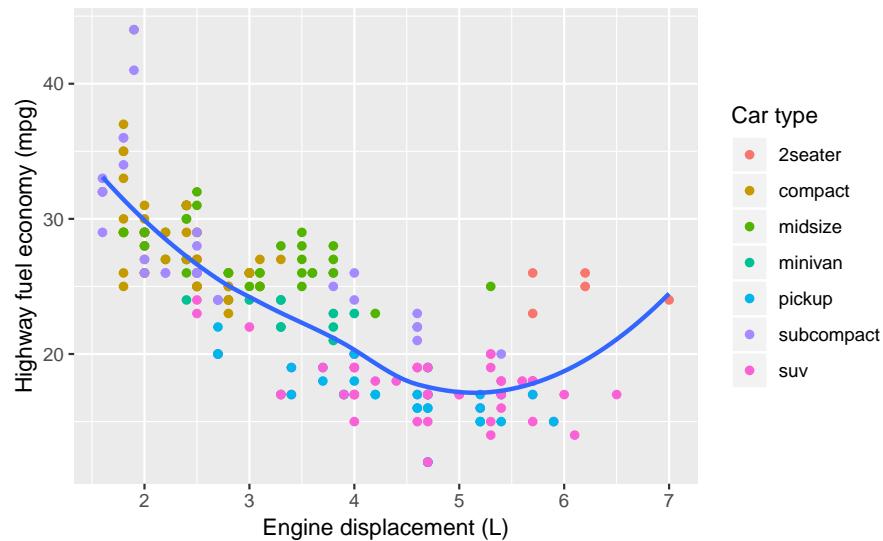
```
ggplot(df, aes(x, y)) +
 geom_hex() +
 scale_fill_gradient(low = "white", high = "red") +
 coord_fixed()
```



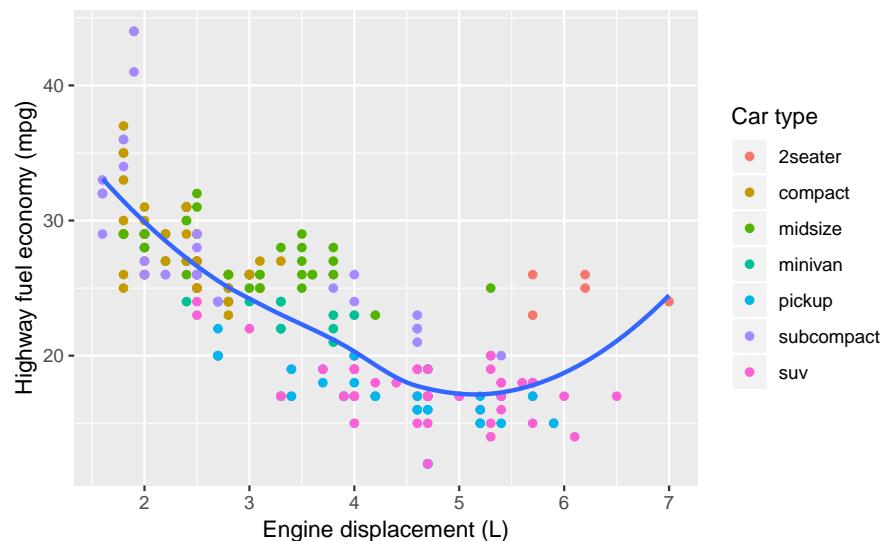
### Exercise 28.4.2

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' and formula 'y ~ x'
```



```
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' and formula 'y ~ x'
```



### Exercise 28.4.3

Change the display of the presidential terms by:

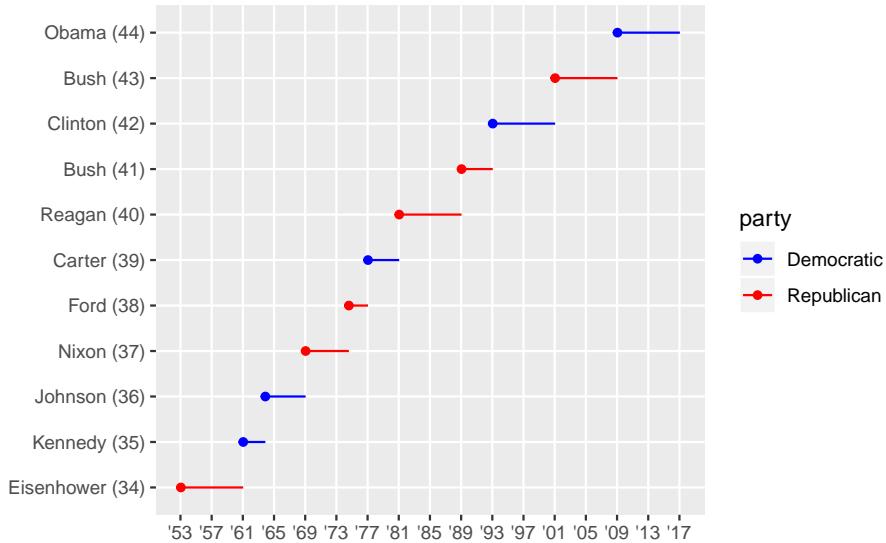
1. Combining the two variants shown above.
2. Improving the display of the y axis.
3. Labeling each term with the name of the president.
4. Adding informative plot labels.
5. Placing breaks every 4 years (this is trickier than it seems!).

```

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())

```



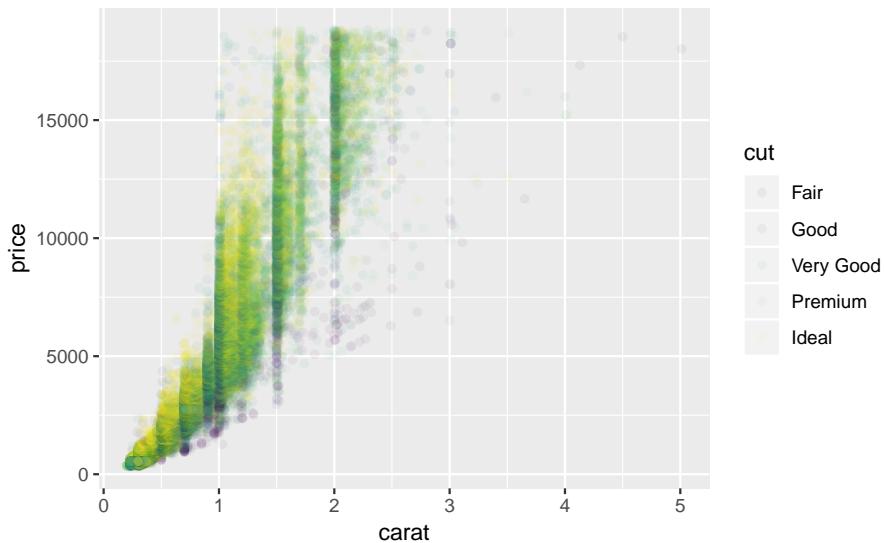
#### Exercise 28.4.4

Use `override.aes` to make the legend on the following plot easier to see.

```

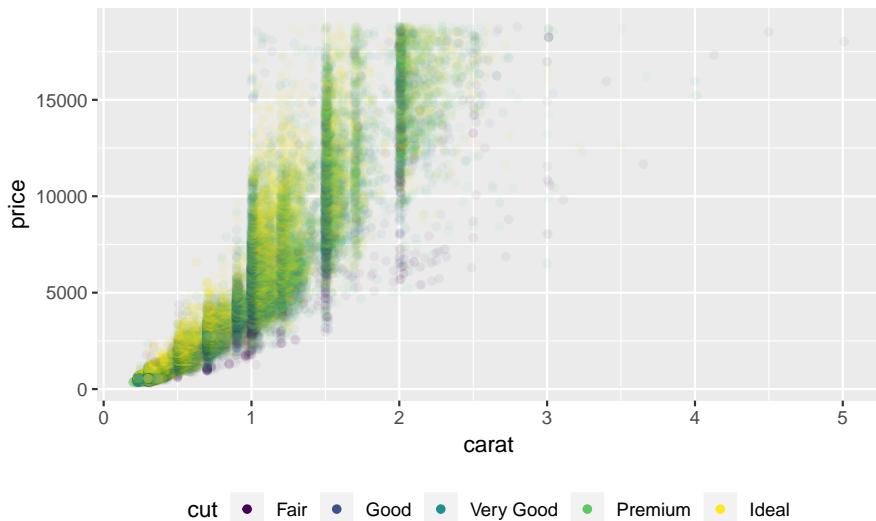
ggplot(diamonds, aes(carat, price)) +
 geom_point(aes(colour = cut), alpha = 1/20)

```



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)))
```



## 28.5 Zooming

No exercises.

## 28.6 Themes

No exercises.

## 28.7 Saving your plots

No exercises.

## 28.8 Learning more

No exercises.

# Chapter 29

## R Markdown formats

No exercises.

This document was built with **bookdown**. You can see the source code and text at <https://github.com/jrnold/r4ds-exercise-solutions>.



# Chapter 30

## R Markdown workflow

No exercises

- Cleveland, William S. 1993a. “A Model for Studying Display Methods of Statistical Graphics.” *Journal of Computational and Graphical Statistics* 2 (4). Taylor & Francis:323–43. <https://doi.org/10.1080/10618600.1993.10474616>.
- . 1993b. *Visualizing Information*. Hobart Press.
- . 1994. *The Elements of Graphing Data*. Hobart Press.
- Cleveland, William S., Marylyn E. McGill, and Robert McGill. 1988. “The Shape Parameter of a Two-Variable Graph.” *Journal of the American Statistical Association* 83 (402). [American Statistical Association, Taylor & Francis, Ltd.]:289–300. <https://www.jstor.org/stable/2288843>.
- Doane, David P., and Lori E. Seward. 2011. “Measuring Skewness: A Forgotten Statistic?” *Journal of Statistics Education* 19 (2). Taylor & Francis:null. <https://doi.org/10.1080/10691898.2011.11889611>.
- Heer, Jeffrey, and Maneesh Agrawala. 2006. “Multi-Scale Banking to 45°.” *Ieee Transactions on Visualization and Computer Graphics* 12 (5, September/October). <https://doi.org/10.1109/TVCG.2006.163>.
- Hintze, Jerry L., and Ray D. Nelson. 1998. “Violin Plots: A Box Plot-Density Trace Synergism.” *The American Statistician* 52 (2). Taylor & Francis:181–84. <https://doi.org/10.1080/00031305.1998.10480559>.
- Hofmann, Heike, Hadley Wickham, and Karen Kafadar. 2017. “Letter-Value Plots: Boxplots for Large Data.” *Journal of Computational and Graphical Statistics* 26 (3). Taylor & Francis:469–77. <https://doi.org/10.1080/10618600.2017.1305277>.
- Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 1st ed. O’Reilly Media.