

TRES Tidyverse Tutorial

Raphael, Pratik, Theo and Richèl

2020-07-24

Contents

2	Outline	5
3	About	5
4	Schedule	5
5	Possible extras	5
6	Join	6
7	1 Reading files and string manipulation	7
8	1.1 Data import and export with <code>readr</code>	7
9	1.2 String manipulation with <code>stringr</code>	10
10	1.3 String interpolation with <code>glue</code>	18
11	1.4 Strings in <code>ggplot</code>	19
12	2 Reshaping data tables in the tidyverse, and other things	21
13	2.1 The new data frame: <code>tibble</code>	22
14	2.2 The concept of tidy data	24
15	2.3 Reshaping with <code>tidyr</code>	26
16	2.4 Extra: factors and the <code>forcats</code> package	34
17	2.5 External resources	39
18	3 Data manipulation with <code>dplyr</code>	41
19	3.1 Introduction	41
20	3.2 Working with existing variables	42
21	3.3 Working with observations	50
22	3.4 Making new variables	56
23	3.5 Working with multiple tables	65
24	4 Working with lists and iteration	69
25	4.1 List columns with <code>tidyr</code>	69
26	4.2 Iteration with <code>map</code>	72
27	4.3 More <code>map</code> variants	77
28	4.4 Combining <code>map</code> variants and tidyverse functions	78
29	4.5 A return to <code>map</code> variants	80
30	4.6 Other functions for working with lists	82
31	4.7 Lists of <code>ggplots</code> with <code>patchwork</code>	85

32	5 ggplot2 and the grammar of graphics	87
33	5.1 Introduction	88
34	5.2 But first, the data	89
35	5.3 Geom layers	90
36	5.4 Coordinate-system	101
37	5.5 Facetting	103
38	5.6 The right format for the dataset	106
39	5.7 Plotting as part of a pipeline	106
40	5.8 Customization	107
41	5.9 Combining plots	116
42	5.10 Saving a plot	118
43	5.11 High throughput plotting workflow	118
44	5.12 Want more?	121
45	5.13 References	122
46	6 Regular expressions and testthat	123
47	6.1 Introduction	124
48	6.2 Testing	127
49	6.3 Detect a full match	128
50	6.4 Extract a pattern for one submatch	135
51	6.5 Extract a pattern for multiple submatches	139
52	6.6 Mutate	141
53	6.7 Test for match	141
54	6.8 Bigger picture	142
55	6.9 Resources	142
56	7 Programming in the tidyverse	143
57	7.1 An explanation of the problem	144
58	7.2 Flexible selection is easy	146
59	7.3 A first attempt at a flexible function	147
60	7.4 Flexible filtering in a function	148
61	7.5 Flexible grouping in a function	150
62	7.6 Flexible summarising in a function	152
63	7.7 Further resources	158
64	8 Developing R packages	159
65	8.1 Why writing packages?	160
66	8.2 Hands-on workflow	160
67	8.3 Write a vignette	163
68	8.4 Update the description	164
69	8.5 Check the package	165
70	8.6 References	167

Outline

This is the readable version of the TRES tidyverse tutorial. A convenient PDF version can be downloaded by clicking the PDF document icon in the header bar.

About

The TRES tidyverse tutorial is an online workshop on how to use the tidyverse, a set of packages in the R computing language designed at making data handling and plotting easier.

This tutorial will take the form of a one hour per week video stream via Google Meet, every Friday morning at 10.00 (Groningen time) starting from the 29th of May, 2020 and lasting for a couple of weeks (depending on the number of topics we want to cover, but there should be at least 5).

PhD students from outside our department are welcome to attend.

Schedule

Topic	Package	Instructor	Date*
Reading data and string manipulation	readr, stringr, glue	Pratik	29/05/20
Data and reshaping	tibble, tidyr	Raphael	05/06/20
Manipulating data	dplyr	Theo	12/06/20
Working with lists and iteration	purrr	Pratik	19/06/20
Plotting	ggplot2	Raphael	26/06/20
Regular expressions	regex	Richel	17/07/20
Programming with the tidyverse	rlang	Pratik	10/07/20

Possible extras

- Reproducibility and package-making (with e.g. usethis)

- 87 • Embedding C++ code with Rcpp

88 **Join**

89 Join the Slack by clicking this link (Slack account required).

90 *Tentative dates.

91 Chapter 1

92 Reading files and string 93 manipulation



Every use case is ridiculous
until it happens to you.

94
95 Load the packages for the day.

```
library(readr)  
library(stringr)  
library(glue)
```

96 1.1 Data import and export with readr

97 Data in the wild with which ecologists and evolutionary biologists deal is most often in the
98 form of a text file, usually with the extensions `.csv` or `.txt`. Often, such data has to be
99 written to file from within R. `readr` contains a number of functions to help with reading
100 and writing text files.

1.1.1 Reading data

Reading in a csv file with `readr` is done with the `read_csv` function, a faster alternative to the base R `read.csv`. Here, `read_csv` is applied to the `mtcars` example.

```
# get the filepath of the example
some_example = readr_example("mtcars.csv")

# read the file in
some_example = read_csv(some_example)

head(some_example)
#> # A tibble: 6 x 11
#>   mpg   cyl  disp    hp  drat    wt   qsec    vs    am  gear  carb
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <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
```

The `read_csv2` function is useful when dealing with files where the separator between columns is a semicolon `;`, and where the decimal point is represented by a comma `,`.

Other variants include:

- `read_tsv` for tab-separated files, and
- `read_delim`, a general case which allows the separator to be specified manually.

`readr` import function will attempt to guess the column type from the first N lines in the data. This N can be set using the function argument `guess_max`. The `n_max` argument sets the number of rows to read, while the `skip` argument sets the number of rows to be skipped before reading data.

By default, the column names are taken from the first row of the data, but they can be manually specified by passing a character vector to `col_names`.

There are some other arguments to the data import functions, but the defaults usually *just work*.

1.1.2 Writing data

Writing data uses the `write_*` family of functions, with implementations for `csv`, `csv2` etc. (represented by the asterisk), mirroring the import functions discussed above. `write_*` functions offer the `append` argument, which allow a data frame to be added to an existing file.

These functions are not covered here.

1.1.3 Reading and writing lines

Sometimes, there is text output generated in R which needs to be written to file, but is not in the form of a dataframe. A good example is model outputs. It is good practice to save model output as a text file, and add it to version control. Similarly, it may be necessary to import such text, either for display to screen, or to extract data.

This can be done using the `readr` functions `read_lines` and `write_lines`. Consider the model summary from a simple linear model.

```
# get the model
model = lm(mpg ~ wt, data = mtcars)
```

The model summary can be written to file. When writing lines to file, BE AWARE OF THE DIFFERENCES BETWEEN UNIX AND WINDOWS line separators. Usually, this causes no trouble.

```
# capture the model summary output
model_output = capture.output(summary(model))
```

```
# save it to file
write_lines(x = model_output,
           path = "model_output.txt")
```

This model output can be read back in for display, and each line of the model output is an element in a character vector.

```
# read in the model output and display
model_output = read_lines("model_output.txt")

# use cat to show the model output as it would be on screen
cat(model_output, sep = "\n")
#>
#> Call:
#> lm(formula = mpg ~ wt, data = mtcars)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -4.543 -2.365 -0.125  1.410  6.873
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   37.285      1.878   19.86 < 2e-16 ***
#> wt           -5.344      0.559   -9.56 1.3e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.05 on 30 degrees of freedom
#> Multiple R-squared:  0.753, Adjusted R-squared:  0.745
```

```
#> F-statistic: 91.4 on 1 and 30 DF, p-value: 1.29e-10
```

135 These few functions demonstrate the most common uses of `readr`, but most other use
 136 cases for text data can be handled using different function arguments, including reading
 137 data off the web, unzipping compressed files before reading, and specifying the column
 138 types to control for type conversion errors.

139 Excel files

140 Finally, data is often shared or stored by well meaning people in the form of Microsoft
 141 Excel sheets. Indeed, Excel (especially when synced regularly to remote storage) is a good
 142 way of noting down observational data in the field. The `readxl` package allows importing
 143 from Excel files, including reading in specific sheets.

144 1.2 String manipulation with `stringr`

145 `stringr` is the tidyverse package for string manipulation, and exists in an interesting
 146 symbiosis with the `stringi` package. For the most part, `stringr` is a wrapper around
 147 `stringi`, and is almost always more than sufficient for day-to-day needs.

148 `stringr` functions begin with `str_`.

149 1.2.1 Putting strings together

150 Concatenate two strings with `str_c`, and duplicate strings with `str_dup`. Flatten a list or
 151 vector of strings using `str_flatten`.

```
# str_c works like paste(), choose a separator  
str_c("this string", "this other string", sep = "_")  
#> [1] "this string_this other string"
```

```
# str_dup works like rep  
str_dup("this string", times = 3)  
#> [1] "this stringthis stringthis string"
```

```
# str_flatten works on lists and vectors  
str_flatten(string = as.list(letters), collapse = "_")  
#> [1] "a_b_c_d_e_f_g_h_i_j_k_l_m_n_o_p_q_r_s_t_u_v_w_x_y_z"  
str_flatten(string = letters, collapse = "-")  
#> [1] "a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
```

152 `str_flatten` is especially useful when displaying the type of an object that returns a list
 153 when `class` is called on it.

```
# get the class of a tibble and display it as a single string  
class_tibble = class(tibble::tibble(a = 1))  
str_flatten(string = class_tibble, collapse = ", ")  
#> [1] "tbl_df, tbl, data.frame"
```

154 **1.2.2 Detecting strings**

155 Count the frequency of a pattern in a string with `str_count`. Returns an integer. Detect
 156 whether a pattern exists in a string with `str_detect`. Returns a logical and can be used
 157 as a predicate.

158 Both are vectorised, i.e. automatically applied to a vector of arguments.

```
# there should be 5 a-s here
str_count(string = "ababababa", pattern = "a")
#> [1] 5

# vectorise over the input string
# should return a vector of length 2, with integers 5 and 3
str_count(string = c("ababbababa", "banana"), pattern = "a")
#> [1] 5 3

# vectorise over the pattern to count both a-s and b-s
str_count(string = "ababababa", pattern = c("a", "b"))
#> [1] 5 4
```

159 Vectorising over both string and pattern works as expected.

```
# vectorise over both string and pattern
# counts a-s in first input, and b-s in the second
str_count(string = c("ababababa", "banana"),
          pattern = c("a", "b"))
#> [1] 5 1

# provide a longer pattern vector to search for both a-s
# and b-s in both inputs
str_count(string = c("ababababa", "banana"),
          pattern = c("a", "b",
                     "b", "a"))
#> [1] 5 1 4 3
```

160 `str_locate` locates the search pattern in a string, and returns the start and end as a two
 161 column matrix.

```
# the behaviour of both str_locate and str_locate_all is
# to find the first match by default
str_locate(string = "banana", pattern = "ana")
#>      start end
#> [1,]      2  4

# str_detect detects a sequence in a string
str_detect(string = "Bananageddon is coming!",
           pattern = "na")
#> [1] TRUE
```

```

# str_detect is also vectorised and returns a two-element logical vector
str_detect(string = "Bananageddon is coming!",
            pattern = c("na", "don"))
#> [1] TRUE TRUE

# use any or all to convert a multi-element logical to a single logical
# here we ask if either of the patterns is detected
any(str_detect(string = "Bananageddon is coming!",
               pattern = c("na", "don")))
#> [1] TRUE

162 Detect whether a string starts or ends with a pattern. Also vectorised. Both have a negate
163 argument, which returns the negative, i.e., returns FALSE if the search pattern is detected.

# taken straight from the examples, because they suffice
fruit <- c("apple", "banana", "pear", "pineapple")
# str_detect looks at the first character
str_starts(fruit, "p")
#> [1] FALSE FALSE TRUE TRUE

# str_ends looks at the last character
str_ends(fruit, "e")
#> [1] TRUE FALSE FALSE TRUE

# an example of negate = TRUE
str_ends(fruit, "e", negate = TRUE)
#> [1] FALSE TRUE TRUE FALSE

164 str_subset [WHICH IS NOT RELATED TO str_sub] helps with subsetting a character
165 vector based on a str_detect predicate. In the example, all elements containing "ba-
166 nana" are subset.

167 str_which has the same logic except that it returns the vector position and not the ele-
168 ments.

# should return a subset vector containing the first two elements
str_subset(c("banana",
            "bananageddon is coming",
            "appleageddon is not real"),
           pattern = "banana")
#> [1] "banana" "bananageddon is coming"

# returns an integer vector
str_which(c("banana",
            "bananageddon is coming",
            "appleageddon is not real"),
          pattern = "banana")

```

```
#> [1] 1 2
```

1.2.3 Matching strings

`str_match` returns all positive matches of the pattern in the string. The return type is a list, with one element per search pattern.

A simple case is shown below where the search pattern is the phrase “banana”.

```
str_match(string = c("banana",
                     "bananageddon",
                     "bananas are bad"),
          pattern = "banana")
#>      [,1]
#> [1,] "banana"
#> [2,] "banana"
#> [3,] "banana"
```

The search pattern can be extended to look for multiple subsets of the search pattern. Consider searching for dates and times.

Here, the search pattern is a regex pattern that looks for a set of four digits (`\\d{4}`) and a month name (`\\w+`) separated by a hyphen. There’s much more to be explored in dealing with dates and times in `lubridate`, another tidyverse package.

The return type is a list, each element is a character matrix where the first column is the string subset matching the full search pattern, and then as many columns as there are parts to the search pattern. The parts of interest in the search pattern are indicated by wrapping them in parentheses. For example, in the case below, wrapping `[-.]` in parentheses will turn it into a distinct part of the search pattern.

```
# first with [-.] treated simply as a separator
str_match(string = c("1970-somemonth-01",
                     "1990-anothermonth-01",
                     "2010-thismonth-01"),
          pattern = "(\\d{4})[-.](\\w+)")
#>      [,1]      [,2]      [,3]
#> [1,] "1970-somemonth" "1970" "somemonth"
#> [2,] "1990-anothermonth" "1990" "anothermonth"
#> [3,] "2010-thismonth" "2010" "thismonth"

# then with [-.] actively searched for
str_match(string = c("1970-somemonth-01",
                     "1990-anothermonth-01",
                     "2010-thismonth-01"),
          pattern = "(\\d{4})([-.])(\\w+)")
#>      [,1]      [,2]      [,3]      [,4]
#> [1,] "1970-somemonth" "1970" "-" "somemonth"
```

```
#> [2,] "1990-anothermonth" "1990" "-" "anothermonth"
#> [3,] "2010-thismonth"      "2010" "-" "thismonth"
```

183 Multiple possible matches are dealt with using `str_match_all`. An example case is un-
 184 certainty in date-time in raw data, where the date has been entered as 1970-somemonth-
 185 01 or 1970/anothermonth/01.

186 The return type is a list, with one element per input string. Each element is a character
 187 matrix, where each row is one possible match, and each column after the first (the full
 188 match) corresponds to the parts of the search pattern.

```
# first with a single date entry
str_match_all(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01"),
              pattern = "(\\d{4})[\\-\\/](\\[a-z\\]+)")

#> [[1]]
#>      [,1]      [,2]      [,3]
#> [1,] "1970-somemonth" "1970" "somemonth"
#> [2,] "1990/anothermonth" "1990" "anothermonth"

# then with multiple date entries
str_match_all(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01",
                        "1990-somemonth-01 or maybe 2001/anothermonth/01"),
              pattern = "(\\d{4})[\\-\\/](\\[a-z\\]+)")

#> [[1]]
#>      [,1]      [,2]      [,3]
#> [1,] "1970-somemonth" "1970" "somemonth"
#> [2,] "1990/anothermonth" "1990" "anothermonth"
#>
#> [[2]]
#>      [,1]      [,2]      [,3]
#> [1,] "1990-somemonth" "1990" "somemonth"
#> [2,] "2001/anothermonth" "2001" "anothermonth"
```

189 1.2.4 Simpler pattern extraction

190 The full functionality of `str_match_*` can be boiled down to the most common use
 191 case, extracting one or more full matches of the search pattern using `str_extract` and
 192 `str_extract_all` respectively.

193 `str_extract` returns a character vector with the same length as the input string vector,
 194 while `str_extract_all` returns a list, with a character vector whose elements are the
 195 matches.

```
# extracting the first full match using str_extract
str_extract(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01",
                      "1990-somemonth-01 or maybe 2001/anothermonth/01"),
            pattern = "(\\d{4})[\\-\\/](\\[a-z\\]+)")

#> [1] "1970-somemonth" "1990-somemonth"
```

```

# extracting all full matches using str_extract all
str_extract_all(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01",
                           "1990-somemonth-01 or maybe 2001/anothermonth/01"),
                pattern = "(\\d{4})(\\-|\\/)([a-z]+)")

#> [[1]]
#> [1] "1970-somemonth"      "1990/anothermonth"
#>
#> [[2]]
#> [1] "1990-somemonth"      "2001/anothermonth"

```

196 1.2.5 Breaking strings apart

197 `str_split`, `str_sub`, In the above date-time example, when reading filenames from a
 198 path, or when working sequences separated by a known pattern generally, `str_split`
 199 can help separate elements of interest.

200 The return type is a list similar to `str_match`.

```

# split on either a hyphen or a forward slash
str_split(string = c("1970-somemonth-01",
                    "1990/anothermonth/01"),
          pattern = "[\\-|\\/]")

#> [[1]]
#> [1] "1970"      "somemonth" "01"
#>
#> [[2]]
#> [1] "1990"      "anothermonth" "01"

```

201 This can be useful in recovering simulation parameters from a filename, but may require
 202 some knowledge of `regex`.

```

# assume a simulation output file
filename = "sim_param1_0.01_param2_0.05_param3_0.01.ext"

# not quite there
str_split(filename, pattern = "_")
#> [[1]]
#> [1] "sim"      "param1"   "0.01"     "param2"   "0.05"     "param3"   "0.01.ext"

# not really
str_split(filename,
          pattern = "sim_")
#> [[1]]
#> [1] ""
#> [2] "param1_0.01_param2_0.05_param3_0.01.ext"

# getting there but still needs work

```

```

str_split(filename,
           pattern = "(sim_)|_*param\\d{1}_|(.ext)")
#> [[1]]
#> [1] ""      ""      "0.01" "0.05" "0.01" ""

```

203 **str_split_fixed** split the string into as many pieces as specified, and can be especially
 204 useful dealing with filepaths.

```

# split on either a hyphen or a forward slash
str_split_fixed(string = "dir_level_1/dir_level_2/file.ext",
                 pattern = "/",
                 n = 2)
#>      [,1]      [,2]
#> [1,] "dir_level_1" "dir_level_2/file.ext"

```

205 1.2.6 Replacing string elements

206 **str_replace** is intended to replace the search pattern, and can be co-opted into the
 207 task of recovering simulation parameters or other data from regularly named files.
 208 **str_replace_all** works the same way but replaces all matches of the search pattern.

```

# replace all unwanted characters from this hypothetical filename with spaces
filename = "sim_param1_0.01_param2_0.05_param3_0.01.ext"
str_replace_all(filename,
                 pattern = "(sim_)|_*param\\d{1}_|(.ext)",
                 replacement = " ")
#> [1] " 0.01 0.05 0.01 "

```

209 **str_remove** is a wrapper around **str_replace** where the replacement is set to "". This
 210 is not covered here.

211 Having replaced unwanted characters in the filename with spaces, **str_trim** offers a way
 212 to remove leading and trailing whitespaces.

```

# trim whitespaces from this filename after replacing unwanted text
filename = "sim_param1_0.01_param2_0.05_param3_0.01.ext"
filename_with_spaces = str_replace_all(filename,
                                       pattern = "(sim_)|_*param\\d{1}_|(.ext)",
                                       replacement = " ")
filename_without_spaces = str_trim(filename_with_spaces)
filename_without_spaces
#> [1] "0.01 0.05 0.01"

# the result can be split on whitespaces to return useful data
str_split(filename_without_spaces, " ")
#> [[1]]
#> [1] "0.01" "0.05" "0.01"

```


213 1.2.7 Subsetting within strings

214 When strings are highly regular, useful data can be extracted from a string using `str_sub`.

215 In the date-time example, the year is always represented by the first four characters.

```
# get the year as characters 1 - 4
str_sub(string = c("1970-somemonth-01",
                  "1990-anothermonth-01",
                  "2010-thismonth-01"),
        start = 1, end = 4)
#> [1] "1970" "1990" "2010"
```

216 Similarly, it's possible to extract the last few characters using negative indices.

```
# get the day as characters -2 to -1
str_sub(string = c("1970-somemonth-01",
                  "1990-anothermonth-21",
                  "2010-thismonth-31"),
        start = -2, end = -1)
#> [1] "01" "21" "31"
```

217 Finally, it's also possible to replace characters within a string based on the position. This

218 requires using the assignment operator `<-`.

```
# replace all days in these dates to 01
date_times = c("1970-somemonth-25",
               "1990-anothermonth-21",
               "2010-thismonth-31")

# a strictly necessary use of the assignment operator
str_sub(date_times,
        start = -2, end = -1) <- "01"

date_times
#> [1] "1970-somemonth-01" "1990-anothermonth-01" "2010-thismonth-01"
```

219 1.2.8 Padding and truncating strings

220 Strings included in filenames or plots are often of unequal lengths, especially when they

221 represent numbers. `str_pad` can pad strings with suitable characters to maintain equal

222 length filenames, with which it is easier to work.

```
# pad so all values have three digits
str_pad(string = c("1", "10", "100"),
        width = 3,
        side = "left",
        pad = "0")
#> [1] "001" "010" "100"
```

223 Strings can also be truncated if they are too long.

```

str_trunc(string = c("bananas are great and wonderful
                      and more stuff about bananas and
                      it really goes on about bananas"),
          width = 27,
          side = "right", ellipsis = "etc. etc.")
#> [1] "bananas are great etc. etc."

```

224 1.2.9 Stringr aspects not covered here

225 Some stringr functions are not covered here. These include:

- 226 • str_wrap (of dubious use),
 - 227 • str_interp, str_glue* (better to use glue; see below),
 - 228 • str_sort, str_order (used in sorting a character vector),
 - 229 • str_to_case* (case conversion), and
 - 230 • str_view* (a graphical view of search pattern matches).
 - 231 • word, boundary etc. The use of word is covered below.
- 232 stringi, of which stringr is a wrapper, offers a lot more flexibility and control.

233 1.3 String interpolation with glue

234 The idea behind string interpolation is to procedurally generate new complex strings
 235 from pre-existing data.

236 glue is as simple as the example shown.

```

# print that each car name is a car model
cars = rownames(head(mtcars))
glue('The {cars} is a car model')
#> The Mazda RX4 is a car model
#> The Mazda RX4 Wag is a car model
#> The Datsun 710 is a car model
#> The Hornet 4 Drive is a car model
#> The Hornet Sportabout is a car model
#> The Valiant is a car model

```

237 This creates and prints a vector of car names stating each is a car model.

238 The related glue_data is even more useful in printing from a dataframe. In this example,
 239 it can quickly generate command line arguments or filenames.

```

# use dataframes for now
parameter_combinations = data.frame(param1 = letters[1:5],
                                     param2 = 1:5)

```

```

# for command line arguments or to start multiple job scripts on the cluster
glue_data(parameter_combinations,
           'simulation-name {param1} {param2}')
#> simulation-name a 1
#> simulation-name b 2
#> simulation-name c 3
#> simulation-name d 4
#> simulation-name e 5

# for filenames
glue_data(parameter_combinations,
           'sim_data_param1_{param1}_param2_{param2}.ext')
#> sim_data_param1_a_param2_1.ext
#> sim_data_param1_b_param2_2.ext
#> sim_data_param1_c_param2_3.ext
#> sim_data_param1_d_param2_4.ext
#> sim_data_param1_e_param2_5.ext

```

240 Finally, the convenient `glue_sql` and `glue_data_sql` are used to safely write SQL
 241 queries where variables from data are appropriately quoted. This is not covered here,
 242 but it is good to know it exists.

243 `glue` has some more functions — `glue_safe`, `glue_collapse`, and `glue_col`, but these
 244 are infrequently used. Their functionality can be found on the `glue` github page.

245 1.4 Strings in ggplot

246 `ggplot` has two geoms (wait for the `ggplot` tutorial to understand more about geoms)
 247 that work with text: `geom_text` and `geom_label`. These geoms allow text to be pasted
 248 on to the main body of a plot.

249 Often, these may overlap when the data are closely spaced. The package `ggrepel` offers
 250 another geom, `geom_text_repel` (and the related `geom_label_repel`) that help arrange
 251 text on a plot so it doesn't overlap with other features. This is *not perfect*, but it works more
 252 often than not.

253 More examples can be found on the `ggrepel` website.

254 Here, the arguments to `geom_text_repel` are taken both from the `mtcars` data (position),
 255 as well as from the car brands extracted using the `stringr::word` (labels), which tries
 256 to separate strings based on a regular pattern.

257 The details of `ggplot` are covered in a later tutorial.

```

library(ggplot2)
library(ggrepel)

# prepare car labels using word function

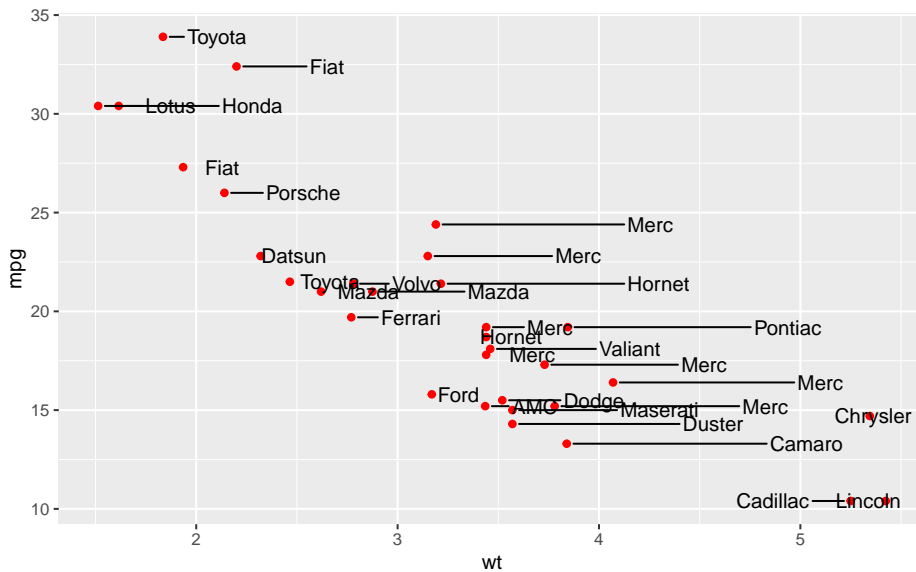
```

```

car_labels = word(rownames(mtcars))

ggplot(mtcars,
  aes(x = wt, y = mpg,
    label = rownames(mtcars)))+
  geom_point(colour = "red")+
  geom_text_repel(aes(label = car_labels),
    direction = "x",
    nudge_x = 0.2,
    box.padding = 0.5,
    point.padding = 0.5)

```



258

259 This is not a good looking plot, because it breaks other rules of plot design, such as
 260 whether this sort of plot should be made at all. Labels and text need to be applied
 261 sparingly, for example drawing attention or adding information to outliers.

Chapter 2

Reshaping data tables in the tidyverse, and other things

Raphael Scherrer

Every use case is ridiculous
until it happens to you.

```
library(tibble)
library(tidyr)
```

In this chapter we will learn what *tidy* means in the context of the tidyverse, and how to reshape our data into a tidy format using the `tidyr` package. But first, let us take a detour and introduce the `tibble`.

2.1 The new data frame: tibble

The `tibble` is the recommended class to use to store tabular data in the tidyverse. Consider it as the operational unit of any data science pipeline. For most practical purposes, a `tibble` is basically a `data.frame`.

```
# Make a data frame
data.frame(who = c("Pratik", "Theo", "Raph"), chapt = c("1, 4", "3", "2, 5"))
#>      who chapt
#> 1 Pratik  1, 4
#> 2  Theo    3
#> 3  Raph   2, 5

# Or an equivalent tibble
tibble(who = c("Pratik", "Theo", "Raph"), chapt = c("1, 4", "3", "2, 5"))
#> # A tibble: 3 x 2
#>   who      chapt
#>   <chr>   <chr>
#> 1 Pratik 1, 4
#> 2 Theo   3
#> 3 Raph   2, 5
```

The difference between `tibble` and `data.frame` is in its display and in the way it is subsetted, among others. Most functions working with `data.frame` will work with `tibble` and vice versa. Use the `as*` family of functions to switch back and forth between the two if needed, using e.g. `as.data.frame` or `as_tibble`.

In terms of display, the `tibble` has the advantage of showing the class of each column: `chr` for character, `fct` for factor, `int` for integer, `dbl` for numeric and `lgl` for logical, just to name the main atomic classes. This may be more important than you think, because many hard-to-find bugs in R are due to wrong variable types and/or cryptic type conversions. This especially happens with `factor` and `character`, which can cause quite some confusion. More about this in the extra section at the end of this chapter!

Note that you can build a `tibble` by rows rather than by columns with `tribble`:

```
tribble(
  ~who, ~chapt,
  "Pratik", "1, 4",
  "Theo", "3",
  "Raph", "2, 5"
)
#> # A tibble: 3 x 2
#>   who      chapt
#>   <chr>   <chr>
#> 1 Pratik 1, 4
#> 2 Theo   3
#> 3 Raph   2, 5
```

285 As a rule of thumb, try to convert your tables to tibbles whenever you can, especially when
 286 the original table is *not* a data frame. For example, the principal component analysis func-
 287 tion `prcomp` outputs a matrix of coordinates in principal component-space.

```
# Perform a PCA on mtcars
pca_scores <- prcomp(mtcars)$x
head(pca_scores) # looks like a data frame or a tibble...
#>      PC1    PC2    PC3    PC4    PC5    PC6    PC7    PC8
#> Mazda RX4      -79.60  2.13 -2.15 -2.707 -0.702 -0.3149 -0.09870 -0.0779
#> Mazda RX4 Wag  -79.60  2.15 -2.22 -2.178 -0.884 -0.4534 -0.00355 -0.0957
#> Datsun 710      -133.89 -5.06 -2.14  0.346  1.106  1.1730  0.00576  0.1362
#> Hornet 4 Drive    8.52 44.99  1.23  0.827  0.424 -0.0579 -0.02431  0.2212
#> Hornet Sportabout 128.69 30.82  3.34 -0.521  0.737 -0.3329  0.10630 -0.0530
#> Valiant         -23.22 35.11 -3.26  1.401  0.803 -0.0884  0.23895  0.4239
#>      PC9    PC10    PC11
#> Mazda RX4      -0.200 -0.2901  0.106
#> Mazda RX4 Wag  -0.353 -0.1928  0.107
#> Datsun 710      -0.198  0.0763  0.267
#> Hornet 4 Drive   0.356 -0.0906  0.209
#> Hornet Sportabout 0.153 -0.1886 -0.109
#> Valiant         0.101 -0.0377  0.276
class(pca_scores) # but is actually a matrix
#> [1] "matrix"

# Convert to tibble
as_tibble(pca_scores)
#> # A tibble: 32 x 11
#>      PC1    PC2    PC3    PC4    PC5    PC6    PC7    PC8    PC9    PC10
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  -79.6    2.13 -2.15 -2.71 -0.702 -0.315 -0.0987 -0.0779 -0.200 -0.290
#> 2  -79.6    2.15 -2.22 -2.18 -0.884 -0.453 -0.00355 -0.0957 -0.353 -0.193
#> 3 -134.    -5.06 -2.14  0.346  1.11  1.17  0.00576  0.136 -0.198  0.0763
#> 4   8.52 45.0  1.23  0.827  0.424 -0.0579 -0.0243  0.221  0.356 -0.0906
#> 5 129.    30.8  3.34 -0.521  0.737 -0.333  0.106 -0.0530  0.153 -0.189
#> 6  -23.2 35.1  -3.26  1.40  0.803 -0.0884  0.239  0.424  0.101 -0.0377
#> # ... with 26 more rows, and 1 more variable: PC11 <dbl>
```

288 This is important because a `matrix` can contain only one type of values (e.g. only numeric
 289 or character), while `tibble` (and `data.frame`) allow you to have columns of different
 290 types.

291 So, in the tidyverse we are going to work with tibbles, got it. But what does “tidy” mean
 292 exactly?

2.2 The concept of tidy data

When it comes to putting data into tables, there are many ways one could organize a dataset. The *tidy* format is one such format. According to the formal definition, a table is tidy if each column is a variable and each row is an observation. In practice, however, I found that this is not a very operational definition, especially in ecology and evolution where we often record multiple variables per individual. So, let's dig in with an example.

Say we have a dataset of several morphometrics measured on Darwin's finches in the Galapagos islands. Let's first get this dataset.

```
# We first simulate random data
beak_lengths <- rnorm(100, mean = 5, sd = 0.1)
beak_widths <- rnorm(100, mean = 2, sd = 0.1)
body_weights <- rgamma(100, shape = 10, rate = 1)
islands <- rep(c("Isabela", "Santa Cruz"), each = 50)

# Assemble into a tibble
data <- tibble(
  id = 1:100,
  body_weight = body_weights,
  beak_length = beak_lengths,
  beak_width = beak_widths,
  island = islands
)

# Snapshot
data
#> # A tibble: 100 x 5
#>   id body_weight beak_length beak_width island
#>   <int>      <dbl>      <dbl>      <dbl> <chr>
#> 1     1      10.8        4.94      1.94 Isabela
#> 2     2      15.4        5.02      2.00 Isabela
#> 3     3      15.0        4.92      1.91 Isabela
#> 4     4       8.51        5.16      2.02 Isabela
#> 5     5      14.9        5.03      1.93 Isabela
#> 6     6       8.41        4.92      2.18 Isabela
#> # ... with 94 more rows
```

Here, we pretend to have measured `beak_length`, `beak_width` and `body_weight` on 100 birds, 50 of them from Isabela and 50 of them from Santa Cruz. In this tibble, each row is an individual bird. This is probably the way most scientists would record their data in the field. However, a single bird is not an “observation” in the sense used in the tidyverse. Our dataset is not tidy but *messy*.

The tidy equivalent of this dataset would be:

```
data <- pivot_longer(
```



```

data,
cols = c("body_weight", "beak_length", "beak_width"),
names_to = "variable"
)
data
#> # A tibble: 300 x 4
#>   id island variable    value
#>   <int> <chr>   <chr>    <dbl>
#> 1     1  Isabel body_weight 10.8
#> 2     1  Isabel beak_length 4.94
#> 3     1  Isabel beak_width  1.94
#> 4     2  Isabel body_weight 15.4
#> 5     2  Isabel beak_length 5.02
#> 6     2  Isabel beak_width  2.00
#> # ... with 294 more rows

```

307 where each *measurement* (and not each *individual*) is now the unit of observation (the rows).
 308 The `pivot_longer` function is the easiest way to get to this format. It belongs to the `tidyr`
 309 package, which we'll cover in a minute.

310 As you can see our tibble now has three times as many rows and fewer columns. This
 311 format is rather unintuitive and not optimal for display. However, it provides a very stan-
 312 dardized and consistent way of organizing data that will be understood (and expected) by
 313 pretty much all functions in the tidyverse. This makes the tidyverse tools work well to-
 314 gether and reduces the time you would otherwise spend reformatting your data from one
 315 tool to the next.

316 That does not mean that the *messy* format is useless though. There may be use-cases
 317 where you need to switch back and forth between formats. For this reason I prefer re-
 318 ferring to these formats using their other names: *long* (tidy) versus *wide* (messy). For ex-
 319 ample, matrix operations work much faster on wide data, and the wide format arguably
 320 looks nicer for display. Luckily the `tidyr` package gives us the tools to reshape our data
 321 as needed, as we shall see shortly.

322 Another common example of wide-or-long dilemma is when dealing with *contingency ta-*
 323 *bles*. This would be our case, for example, if we asked how many observations we have for
 324 each morphometric and each island. We use `table` (from base R) to get the answer:

```

# Make a contingency table
ctg <- with(data, table(island, variable))
ctg
#>           variable
#> island  beak_length beak_width body_weight
#>  Isabel           50           50           50
#> Santa Cruz        50           50           50

```

325 A variety of statistical tests can be used on contingency tables such as Fisher's exact test,
 326 the chi-square test or the binomial test. Contingency tables are in the wide format by con-
 327 struction, but they too can be pivoted to the long format, and the tidyverse manipulation

tools will expect you to do so. Actually, `tibble` knows that very well and does it by default if you convert your `table` into a `tibble`:

```
# Contingency table is pivoted to the long-format automatically
as_tibble(ctg)
#> # A tibble: 6 x 3
#>   island      variable      n
#>   <chr>      <chr>    <int>
#> 1 Isabela    beak_length    50
#> 2 Santa Cruz beak_length    50
#> 3 Isabela    beak_width     50
#> 4 Santa Cruz beak_width     50
#> 5 Isabela    body_weight    50
#> 6 Santa Cruz body_weight    50
```

Summary: Tidy or not tidy

To sum up, the definition of what is tidy and what is not is somewhat subjective. Tables can be in long or wide format, and depending on the complexity of a dataset, there may even be some intermediate states. To be clear, the tidyverse does not only accept long tables, and wide tables may sometimes be the way to go. This is very use-case specific. Have a clear idea of what you want to do with your data (what tidyverse tools you will use), and use that to figure which format makes more sense. And remember, `tidyr` is here to easily do the switching for you.

2.3 Reshaping with `tidyr`

The `tidyr` package implements tools to easily switch between layouts and also perform a few other reshaping operations. Old school R users will be familiar with the `reshape` and `reshape2` packages, of which `tidyr` is the tidyverse equivalent. Beware that `tidyr` is about playing with the general *layout* of the dataset, while *operations* and *transformations* of the data are within the scope of the `dplyr` and `purrr` packages. All these packages work hand-in-hand really well, and analysis pipelines usually involve all of them. But today, we focus on the first member of this holy trinity, which is often the first one you'll need because you will want to reshape your data before doing other things. So, please hold your non-layout-related questions for the next chapters.

2.3.1 Pivoting

Pivoting a dataset between the long and wide layout is the main purpose of `tidyr` (check out the package's logo). We already saw the `pivot_longer` function above. This function converts a table from wide to long format. Similarly, there is a `pivot_wider` function that does exactly the opposite and takes you back to the wide format:

```
pivot_wider(
  data,
```

```

names_from = "variable",
values_from = "value",
id_cols = c("id", "island")
)
#> # A tibble: 100 x 5
#>       id island  body_weight beak_length beak_width
#>   <int> <chr>      <dbl>      <dbl>      <dbl>
#> 1     1  Isabela    10.8        4.94        1.94
#> 2     2  Isabela    15.4        5.02        2.00
#> 3     3  Isabela    15.0        4.92        1.91
#> 4     4  Isabela     8.51        5.16        2.02
#> 5     5  Isabela    14.9        5.03        1.93
#> 6     6  Isabela     8.41        4.92        2.18
#> # ... with 94 more rows

```

345 The order of the columns is not exactly as it was, but this should not matter in a data
 346 analysis pipeline where you should access columns by their names. It is straightforward
 347 to change the order of the columns, but this is more within the scope of the `dplyr` package.

348 If you are familiar with earlier versions of the tidyverse, `pivot_longer` and
 349 `pivot_wider` are the respective equivalents of `gather` and `spread`, which are
 350 now deprecated.

351 There are a few other reshaping operations from `tidyr` that are worth knowing.

352 2.3.2 Handling missing values

353 Say we have some missing measurements in the column “value” of our finch dataset:

```

# We replace 100 random observations by NAs
ii <- sample(nrow(data), 100)
data$value[ii] <- NA
data
#> # A tibble: 300 x 4
#>       id island  variable  value
#>   <int> <chr>    <chr>    <dbl>
#> 1     1  Isabela body_weight 10.8
#> 2     1  Isabela beak_length NA
#> 3     1  Isabela beak_width NA
#> 4     2  Isabela body_weight NA
#> 5     2  Isabela beak_length 5.02
#> 6     2  Isabela beak_width NA
#> # ... with 294 more rows

```

354 We could get rid of the rows that have missing values using `drop_na`:

```

drop_na(data, value)
#> # A tibble: 200 x 4
#>       id island  variable  value

```

```
#>   <int> <chr>   <chr>       <dbl>
#> 1     1  Isabela body_weight 10.8
#> 2     2  Isabela beak_length 5.02
#> 3     3  Isabela body_weight 15.0
#> 4     3  Isabela beak_length 4.92
#> 5     4  Isabela body_weight 8.51
#> 6     4  Isabela beak_width  2.02
#> # ... with 194 more rows
```

355 Else, we could replace the NAs with some user-defined value:

```
replace_na(data, replace = list(value = -999))
#> # A tibble: 300 x 4
#>       id island variable    value
#>   <int> <chr>   <chr>      <dbl>
#> 1     1  Isabela body_weight  10.8
#> 2     1  Isabela beak_length -999
#> 3     1  Isabela beak_width  -999
#> 4     2  Isabela body_weight -999
#> 5     2  Isabela beak_length   5.02
#> 6     2  Isabela beak_width  -999
#> # ... with 294 more rows
```

356 where the `replace` argument takes a named list, and the names should refer to the
357 columns to apply the replacement to.

358 We could also replace NAs with the most recent non-NA values:

```
fill(data, value)
#> # A tibble: 300 x 4
#>       id island variable    value
#>   <int> <chr>   <chr>      <dbl>
#> 1     1  Isabela body_weight 10.8
#> 2     1  Isabela beak_length 10.8
#> 3     1  Isabela beak_width 10.8
#> 4     2  Isabela body_weight 10.8
#> 5     2  Isabela beak_length 5.02
#> 6     2  Isabela beak_width 5.02
#> # ... with 294 more rows
```

359 Note that most functions in the tidyverse take a tibble as their first argument, and
360 columns to which to apply the functions are usually passed as “objects” rather than
361 character strings. In the above example, we passed the `value` column as `value`, not
362 “`value`”. These column-objects are called by the tidyverse functions *in the context* of the
363 data (the tibble) they belong to.

2.3.3 Splitting and combining cells

The `tidyr` package offers tools to split and combine columns. This is a nice extension to the string manipulations we saw last week in the `stringr` tutorial.

Say we want to add the specific dates when we took measurements on our birds (we would normally do this using `dplyr` but for now we will stick to the old way):

```
# Sample random dates for each observation
data$day <- sample(30, nrow(data), replace = TRUE)
data$month <- sample(12, nrow(data), replace = TRUE)
data$year <- sample(2019:2020, nrow(data), replace = TRUE)
data
#> # A tibble: 300 x 7
#>   id island variable    value  day month  year
#>   <int> <chr>   <chr>      <dbl> <int> <int> <int>
#> 1     1  Isabela body_weight 10.8     8     7  2020
#> 2     1  Isabela beak_length NA      19     7  2019
#> 3     1  Isabela beak_width  NA     17    12  2019
#> 4     2  Isabela body_weight  NA     20    12  2020
#> 5     2  Isabela beak_length  5.02    21    10  2020
#> 6     2  Isabela beak_width  NA     23     2  2020
#> # ... with 294 more rows
```

We could combine the day, month and year columns into a single date column, with a dash as a separator, using `unite`:

```
data <- unite(data, day, month, year, col = "date", sep = "-")
data
#> # A tibble: 300 x 5
#>   id island variable    value date
#>   <int> <chr>   <chr>      <dbl> <chr>
#> 1     1  Isabela body_weight 10.8 8-7-2020
#> 2     1  Isabela beak_length NA    19-7-2019
#> 3     1  Isabela beak_width  NA    17-12-2019
#> 4     2  Isabela body_weight  NA    20-12-2020
#> 5     2  Isabela beak_length  5.02 21-10-2020
#> 6     2  Isabela beak_width  NA    23-2-2020
#> # ... with 294 more rows
```

Of course, we can revert back to the previous dataset by splitting the date column with `separate`.

```
separate(data, date, into = c("day", "month", "year"))
#> # A tibble: 300 x 7
#>   id island variable    value day  month year
#>   <int> <chr>   <chr>      <dbl> <chr> <chr> <chr>
#> 1     1  Isabela body_weight 10.8   8     7    2020
#> 2     1  Isabela beak_length NA    19     7    2019
```

```
#> 3      1 Isabela beak_width NA      17      12      2019
#> 4      2 Isabela body_weight NA      20      12      2020
#> 5      2 Isabela beak_length 5.02 21      10      2020
#> 6      2 Isabela beak_width NA      23      2      2020
#> # ... with 294 more rows
```

373 But note that the day, month and year columns are now of class character and not in-
 374 teger anymore. This is because they result from the splitting of date, which itself was a
 375 character column.

376 You can also separate a single column into multiple rows using `separate_rows`:

```
separate_rows(data, date)
#> # A tibble: 900 x 5
#>       id island variable    value date
#>   <int> <chr>    <chr>    <dbl> <chr>
#> 1      1 Isabela body_weight  10.8 8
#> 2      1 Isabela body_weight  10.8 7
#> 3      1 Isabela body_weight  10.8 2020
#> 4      1 Isabela beak_length  NA    19
#> 5      1 Isabela beak_length  NA     7
#> 6      1 Isabela beak_length  NA   2019
#> # ... with 894 more rows
```

377 2.3.4 Expanding tables using combinations

378 Instead of getting rid of rows with NAs, we may want to add rows with NAs, for example,
 379 for combinations of parameters that we did not measure.

```
data <- separate(data, date, into = c("day", "month", "year"))
to_rm <- with(data, island == "Santa Cruz" & year == "2020")
data <- data[!to_rm,]
tail(data)
#> # A tibble: 6 x 7
#>       id island variable    value day month year
#>   <int> <chr>    <chr>    <dbl> <chr> <chr> <chr>
#> 1    98 Santa Cruz beak_length  4.94 22    12    2019
#> 2    98 Santa Cruz beak_width   1.90 9      1    2019
#> 3    99 Santa Cruz body_weight  15.0 16     7    2019
#> 4    99 Santa Cruz beak_length  NA    26    10    2019
#> 5    99 Santa Cruz beak_width   2.04 30     7    2019
#> 6   100 Santa Cruz beak_width   NA    23     3    2019
```

380 We could generate a tibble with all combinations of island, morphometric and year using
 381 `expand_grid`:

```
expand_grid(
  island = c("Isabela", "Santa Cruz"),
  year = c("2019", "2020")
)
```

```
)
#> # A tibble: 4 x 2
#>   island   year
#>   <chr>    <chr>
#> 1 Isabela 2019
#> 2 Isabela 2020
#> 3 Santa Cruz 2019
#> 4 Santa Cruz 2020
```

382 If we already have a tibble to work from that contains the variables to combine, we can
 383 use `expand` on that tibble:

```
expand(data, island, year)
#> # A tibble: 4 x 2
#>   island   year
#>   <chr>    <chr>
#> 1 Isabela 2019
#> 2 Isabela 2020
#> 3 Santa Cruz 2019
#> 4 Santa Cruz 2020
```

384 As you can see, we get all the combinations of the variables of interest, even those that are
 385 missing. But sometimes you might be interested in variables that are *nested* within each
 386 other and not *crossed*. For example, say we have measured birds at different locations
 387 within each island:

```
nrow_Isabela <- with(data, length(which(island == "Isabela")))
nrow_SantaCruz <- with(data, length(which(island == "Santa Cruz")))
sites_Isabela <- sample(c("A", "B"), size = nrow_Isabela, replace = TRUE)
sites_SantaCruz <- sample(c("C", "D"), size = nrow_SantaCruz, replace = TRUE)
sites <- c(sites_Isabela, sites_SantaCruz)
data$site <- sites
data
#> # A tibble: 232 x 8
#>       id island variable   value day  month year site
#>   <int> <chr>   <chr>     <dbl> <chr> <chr> <chr> <chr>
#> 1     1 Isabela body_weight 10.8   8     7    2020 A
#> 2     1 Isabela beak_length NA     19    7    2019 B
#> 3     1 Isabela beak_width NA     17   12    2019 B
#> 4     2 Isabela body_weight NA     20   12    2020 A
#> 5     2 Isabela beak_length 5.02  21   10    2020 A
#> 6     2 Isabela beak_width NA     23    2    2020 A
#> # ... with 226 more rows
```

388 Of course, if sites A and B are on Isabela, they cannot be on Santa Cruz, where we have sites
 389 C and D instead. It would not make sense to expand assuming that `island` and `site` are
 390 crossed, instead, they are nested. We can therefore expand using the `nesting` function:

```
expand(data, nesting(island, site, year))
```

```
#> # A tibble: 6 x 3
#>   island   site year
#>   <chr>   <chr> <chr>
#> 1 Isabela A     2019
#> 2 Isabela A     2020
#> 3 Isabela B     2019
#> 4 Isabela B     2020
#> 5 Santa Cruz C     2019
#> 6 Santa Cruz D     2019
```

391 But now the missing data for Santa Cruz in 2020 are not accounted for because `expand`
 392 thinks the `year` is also nested within `island`. To get back the missing combination, we use
 393 `crossing`, the complement of nesting:

```
expand(data, crossing(nesting(island, site), year)) # both can be used together
#> # A tibble: 8 x 3
#>   island   site year
#>   <chr>   <chr> <chr>
#> 1 Isabela A     2019
#> 2 Isabela A     2020
#> 3 Isabela B     2019
#> 4 Isabela B     2020
#> 5 Santa Cruz C     2019
#> 6 Santa Cruz C     2020
#> # ... with 2 more rows
```

394 Here, we specify that `site` is nested within `island` and these two are crossed with `year`.
 395 Easy!

396 But wait a minute. These combinations are all very good, but our measurements have
 397 disappeared! We can get them back by levelling up to the `complete` function instead of
 398 using `expand`:

```
tail(complete(data, crossing(nesting(island, site), year)))
#> # A tibble: 6 x 8
#>   island   site year   id variable  value day  month
#>   <chr>   <chr> <chr> <int> <chr>    <dbl> <chr> <chr>
#> 1 Santa Cruz D    2019    95 beak_width NA    13    10
#> 2 Santa Cruz D    2019    98 beak_length 4.94 22    12
#> 3 Santa Cruz D    2019    99 body_weight 15.0 16     7
#> 4 Santa Cruz D    2019    99 beak_length NA    26    10
#> 5 Santa Cruz D    2019    99 beak_width  2.04 30     7
#> 6 Santa Cruz D    2020    NA <NA>      NA    <NA> <NA>
# the last row has been added, full of NAs
```

399 which nicely keeps the rest of the columns in the tibble and just adds the missing combi-
 400 nations.

2.3.5 Nesting

The `tidyr` package has yet another feature that makes the tidyverse very powerful: the `nest` function. However, it makes little sense without combining it with the functions in the `purrr` package, so we will not cover it in this chapter but rather in the `purrr` chapter.

2.3.6 What else can be tidied up?

2.3.6.1 Model output with broom

Check out the `broom` package and its `tidy` function to tidy up messy linear model output, e.g.

```
library(broom)
fit <- lm(mpg ~ cyl, mtcars)
summary(fit)
#>
#> Call:
#> lm(formula = mpg ~ cyl, data = mtcars)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -4.981 -2.119  0.222  1.072  7.519
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   37.885      2.074   18.27 < 2e-16 ***
#> cyl           -2.876      0.322   -8.92 6.11e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.21 on 30 degrees of freedom
#> Multiple R-squared:  0.726, Adjusted R-squared:  0.717
#> F-statistic: 79.6 on 1 and 30 DF, p-value: 6.11e-10
tidy(fit) # returns a tibble
#> # A tibble: 2 x 5
#>   term          estimate std.error statistic  p.value
#>   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
#> 1 (Intercept)   37.9       2.07      18.3 8.37e-18
#> 2 cyl          -2.88      0.322    -8.92 6.11e-10
```

The `broom` package is just one package among a series of packages together known as `tidymodels` that deal with statistical models according to the tidyverse philosophy, and those include machine learning models.

412 2.3.6.2 Graphs with **tidygraph**

413 For some datasets, sometimes there is no trivial and intuitive way to store them into a
 414 table. This is the case, for example, for data underlying graphs (as in networks), which
 415 contain information about relations between entities. What is the unit of observation in a
 416 network? A node? An edge between two nodes? Nodes and edges in a network may each
 417 have node- or edge-specific variables mapped to them, and both may be equally valid
 418 units of observation. The **tidygraph** package has tools to store graph-data in a tidyverse-
 419 friendly object, consisting of two tibbles: one for node-specific information, the other for
 420 edge-specific information. This package goes hand in hand with the **ggraph**, that makes
 421 plotting networks compatible with the grammar of graphics.

422 2.3.6.3 Trees with **tidytree**

423 Phylogenetic trees are a special type of graphs suffering from the same issue, i.e. of being
 424 non-trivial to store in a table. The **tidytree** package and its companion **treeio** offer
 425 an interface to convert tree-like objects (from most format used by other packages and
 426 software) into a tidyverse-friendly format. Again, the point is that the rest of the tidyverse
 427 can be used to wrangle or plot this type of data in the same way as one would do with
 428 regular tabular data. For plotting a **tidytree** with the grammar of graphics, see **ggtree**.

429 2.4 Extra: factors and the **forcats** package

```
library(forcats)
```

430 Categorical variables can be stored in R as character strings in **character** or **factor**
 431 objects. A **factor** looks like a **character**, but it actually is an **integer** vector, where
 432 each **integer** is mapped to a **character** label. With this respect it is sort of an enhanced
 433 version of **character**. For example,

```
my_char_vec <- c("Pratik", "Theo", "Raph")
my_char_vec
#> [1] "Pratik" "Theo"   "Raph"
```

434 is a **character** vector, recognizable to its double quotes, while

```
my_fact_vec <- factor(my_char_vec) # as.factor would work too
my_fact_vec
#> [1] Pratik Theo   Raph
#> Levels: Pratik Raph Theo
```

435 is a **factor**, of which the *labels* are displayed. The *levels* of the factor are the unique values
 436 that appear in the vector. If I added an extra occurrence of my name:

```
factor(c(my_char_vec, "Raph"))
#> [1] Pratik Theo   Raph   Raph
#> Levels: Pratik Raph Theo
```

we would still have the the same levels. Note that the levels are returned as a character vector in alphabetical order by the `levels` function:

```
levels(my_fact_vec)
#> [1] "Pratik" "Raph"  "Theo"
```

Why does it matter? Well, most operations on categorical variables can be performed on character of factor objects, so it does not matter so much which one you use for your own data. However, some functions in R require you to provide categorical variables in one specific format, and others may even implicitly convert your variables. In `ggplot2` for example, character vectors are converted into factors by default. So, it is always good to remember the differences and what type your variables are.

But this is a tidyverse tutorial, so I would like to introduce here the package `forcats`, which offers tools to manipulate factors. First of all, most tools from `stringr` will work on factors. The `forcats` functions expand the string manipulation toolbox with factor-specific utilities. Similar in philosophy to `stringr` where functions started with `str_`, in `forcats` most functions start with `fct_`.

I see two main ways `forcats` can come handy in the kind of data most people deal with: playing with the order of the levels of a factor and playing with the levels themselves. We will show here a few examples, but the full breadth of factor manipulations can be found online or in the excellent `forcats` cheatsheet.

2.4.1 Change the order of the levels

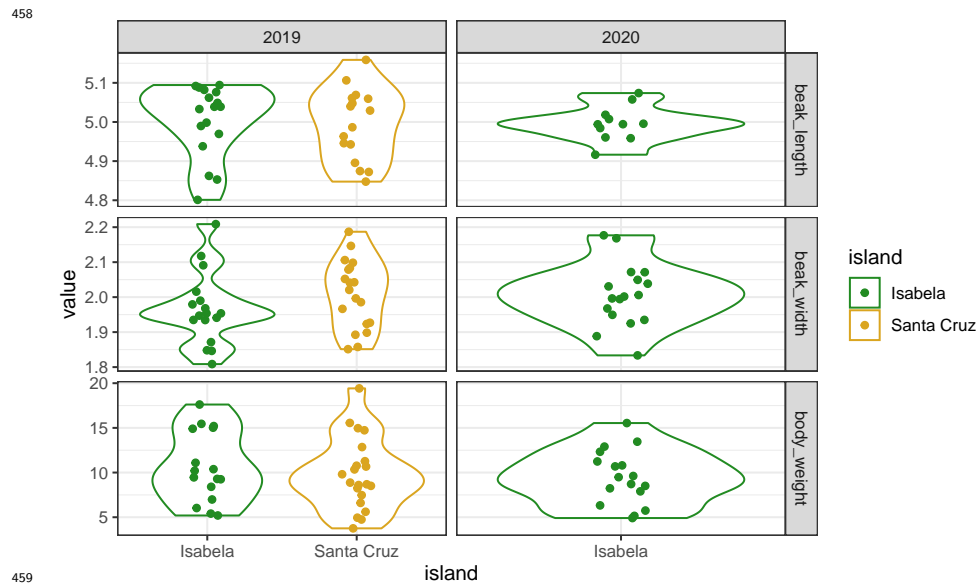
One example use-case where you would want to change the order of the levels of a factor is when plotting. Your categorical variable, for example, may not be plotted in the order you want. If we plot the distribution of each variable across islands, we get

```
# Make the plotting code a function so we can re-use it without copying and pasting
my_plot <- function(data) {

  # We do not cover the ggplot functions in this chapter, this is just to
  # illustrate our use-case, wait until chapter 5!
  library(ggplot2)
  ggplot(data, aes(x = island, y = value, color = island)) +
    geom_violin() +
    geom_jitter(width = 0.1) +
    facet_grid(variable ~ year, scales = "free") +
    theme_bw() +
    scale_color_manual(values = c("forestgreen", "goldenrod"))

}

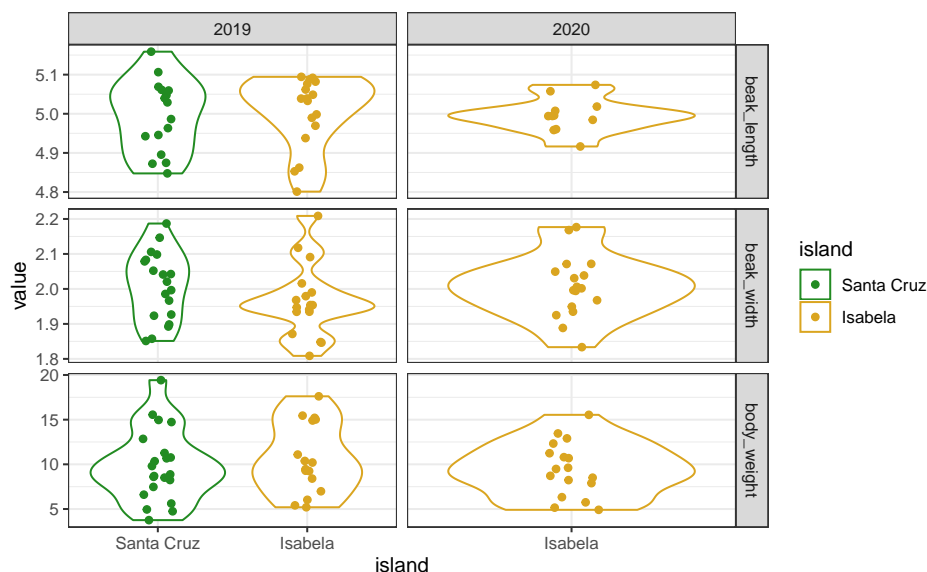
my_plot(data)
# Remember that data are missing from Santa Cruz in 2020
```



460 Here, the islands (horizontal axis) and the variables (the facets) are displayed in alphabet-
 461 ical order. When making a figure you may want to customize these orders in such a way
 462 that your message is optimally conveyed by your figure, and this may involve playing with
 463 the order of levels.

464 Use `fct_relevel` to manually change the order of the levels:

```
data$island <- as.factor(data$island) # turn this column into a factor
data$island <- fct_relevel(data$island, c("Santa Cruz", "Isabela"))
my_plot(data) # order of islands has changed!
```



466

467 Beware that reordering a factor *does not change* the order of the items within the vector,
 468 only the order of the *levels*. So, it does not introduce any mismatch between the `island`
 469 column and the other columns! It only matters when the levels are called, for example, in
 470 a `ggplot`. As you can see:

```
data$island[1:10]
#> [1] Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela
#> [10] Isabela
#> Levels: Santa Cruz Isabela
fct_relevel(data$island, c("Isabela", "Santa Cruz"))[1:10] # same thing, different levels
#> [1] Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela
#> [10] Isabela
#> Levels: Isabela Santa Cruz
```

471 Alternatively, use `fct_inorder` to set the order of the levels to the order in which they
 472 appear:

```
data$variable <- as.factor(data$variable)
levels(data$variable)
#> [1] "beak_length" "beak_width" "body_weight"
levels(fct_inorder(data$variable))
#> [1] "body_weight" "beak_length" "beak_width"
```

473 or `fct_rev` to reverse the order of the levels:

```
levels(fct_rev(data$island)) # back in the alphabetical order
#> [1] "Isabela" "Santa Cruz"
```

474 Other variants exist to do more complex reordering, all present in the `forcats` cheatsheet,
 475 for example: * `fct_infreq` to re-order according to the frequency of each level (how

many observation on each island?) * `fct_shift` to shift the order of all levels by a certain rank (in a circular way so that the last one becomes the first one or vice versa) * `fct_shuffle` if you want your levels in random order * `fct_reorder`, which reorders based on an associated variable (see `fct_reorder2` for even more complex relationship between the factor and the associated variable)

2.4.2 Change the levels themselves

Changing the levels of a factor will change the labels in the actual vector. It is similar to performing a string substitution in `stringr`. One can change the levels of a factor using `fct_recode`:

```
fct_recode(
  my_fact_vec,
  "Pratik Gupte" = "Pratik",
  "Theo Pannetier" = "Theo",
  "Raphael Scherrer" = "Raph"
)
#> [1] Pratik Gupte      Theo Pannetier    Raphael Scherrer
#> Levels: Pratik Gupte Raphael Scherrer Theo Pannetier
```

or collapse factor levels together using `fct_collapse`:

```
fct_collapse(my_fact_vec, EU = c("Theo", "Raph"), NonEU = "Pratik")
#> [1] NonEU EU      EU
#> Levels: NonEU EU
```

Again, we do not provide an exhaustive list of `forcats` functions here but the most usual ones, to give a glimpse of many things that one can do with factors. So, if you are dealing with factors, remember that `forcats` may have handy tools for you. Among others:

- * `fct_anon` to “anonymize”, i.e. replace the levels by random integers
- * `fct_lump` to collapse levels together based on their frequency (e.g. the two most frequent levels together)

2.4.3 Dropping levels

If you use factors in your tibble and get rid of one level, for any reason, the factor will usually remember the old levels, which may cause some problems when applying functions to your data.

```
data <- data[data$island == "Santa Cruz",] # keep only one island
unique(data$island) # Isabela is gone from the labels
#> [1] Santa Cruz
#> Levels: Santa Cruz Isabela
levels(data$island) # but not from the levels
#> [1] "Santa Cruz" "Isabela"
```

Use `droplevels` (from base R) to make sure you get rid of levels that are not in your data anymore:

```
data <- droplevels(data)
levels(data$island)
#> [1] "Santa Cruz"
```

497 Fortunately, most functions within the tidyverse will not complain about missing levels,
498 and will automatically get rid of those inexistant levels for you. But because factors are
499 such common causes of bugs, keep this in mind!

500 Note that this is equivalent to doing:

```
data$island <- fct_drop(data$island)
```

501 2.4.4 Other things

502 Among other things you can use in forcats: * `fct_count` to get the frequency of each
503 level * `fct_c` to combine factors together

504 2.4.5 Take home message for forcats

505 Use this package to manipulate your factors. Do you need factors? Or are character vec-
506 tors enough? That is your call, and may depend on the kind of analyses you want to do
507 and what they require. We saw here that for plotting, having factors can allow you to do
508 quite some tweaking of the display. If you encounter a situation where the order of encod-
509 ing of your character vector starts to matter, then maybe converting into a factor would
510 make your life easier. And if you do so, remember that lots of tools to perform all kinds of
511 manipulation are available to you with both `stringr` and `forcats`.

512 2.5 External resources

513 Find lots of additional info by looking up the following links:

- 514 • The `readr`/`tibble`/`tidyr` and `forcats` cheatsheets.
- 515 • This link on the concept of tidy data
- 516 • The `tibble`, `tidyr` and `forcats` websites
- 517 • The `broom`, `tidymodels`, `tidygraph` and `tidytrees` websites

Chapter 3

Data manipulation with dplyr

```
# load the tidyverse
library(tidyverse)
```

3.1 Introduction

3.1.1 Foreword on dplyr

dplyr is tasked with performing all sorts of transformations on a dataset.

The structure of dplyr revolves around a set of functions, the so-called **verbs**, that share a common syntax and logic, and are meant to work with one another in chained operations. Chained operations are performed with the pipe operator (`%>%`), that will be introduced in section 3.2.2.

The basic syntax is `verb(data, variable)`, where `data` is a data frame and `variable` is the name of one or more columns containing a set of values for each observation.

There are 5 main verbs, which names already hint at what they do: `rename()`, `select()`, `filter()`, `mutate()`, and `summarise()`. I'm going to introduce each of them (and a couple more) through the following sections.

3.1.2 Example data

Through this tutorial, we will be using mammal trait data from the Phylacine database. Let's have a peek at what it contains.

```
phylacine <- read_csv("data/phylacine_traits.csv")
phylacine
#> # A tibble: 5,831 x 24
#>   Binomial.1.2 Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>          <chr>      <chr>      <chr>      <chr>          <dbl>  <dbl>
```

```
#> 1 Abditomys_l~ Rodentia Muridae Abditomys latidens 1 0
#> 2 Abeomelomys~ Rodentia Muridae Abeomelo~ sevia 1 0
#> 3 Abrawayaomy~ Rodentia Cricetidae Abrawaya~ ruschii 1 0
#> 4 Abrocoma_be~ Rodentia Abrocomid~ Abrocoma bennettii 1 0
#> 5 Abrocoma_bo~ Rodentia Abrocomid~ Abrocoma boliviensis 1 0
#> 6 Abrocoma_bu~ Rodentia Abrocomid~ Abrocoma budini 1 0
#> # ... with 5,825 more rows, and 17 more variables: Freshwater <dbl>,
#> # Aerial <dbl>, Life.Habit.Method <chr>, Life.Habit.Source <chr>,
#> # Mass.g <dbl>, Mass.Method <chr>, Mass.Source <chr>, Mass.Comparison <chr>,
#> # Mass.Comparison.Source <chr>, Island.Endemicity <chr>,
#> # IUCN.Status.1.2 <chr>, Added.IUCN.Status.1.2 <chr>, Diet.Plant <dbl>,
#> # Diet.Vertebrate <dbl>, Diet.Invertebrate <dbl>, Diet.Method <chr>,
#> # Diet.Source <chr>
```

535 `readr` automatically loads the data in a `tibble`, as we have seen in chapter 1 and 2. Call-
 536 ing the `tibble` gives a nice preview of what it contains. We have data for 5,831 mammal
 537 species, and the variables contain information on taxonomy, (broad) habitat, mass, IUCN
 538 status, and diet.

539 If you remember Section 1.2 on tidy data, you may see that this data isn't exactly tidy. In
 540 fact, some columns are in wide (and messy) format, like the "habitat" (terrestrial, marine,
 541 etc.) and diet columns.

542 `dplyr` actually does not require your data to be strictly tidy. If you feel that your data
 543 satisfies the definition "one observation per row, one variable per column", that's probably
 544 good enough.

545 I use a `tibble` here, but `dplyr` works equally well on base data frames. In fact, `dplyr` is
 546 built for `data.frame` objects, and `tibbles` are data frames. Therefore, `tibbles` are mortal.

547 3.2 Working with existing variables

548 3.2.1 Renaming variables with `rename()`

549 The variable names in the `phylacine` dataset are descriptive, but quite unpractical. Typing
 550 `Binomial.1.2` is cumbersome and subject to typos (in fact, I just made one). `binomial`
 551 would be much simpler to use.

552 Changing names is straightforward with `rename()`.

```
rename(.data = phylacine, "binomial" = Binomial.1.2)
#> # A tibble: 5,831 x 24
#>   binomial Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>      <chr>      <chr>      <chr>      <chr>      <dbl> <dbl>
#> 1 Abditom~ Rodentia Muridae Abditomys latidens 1 0
#> 2 Abeomel~ Rodentia Muridae Abeomelo~ sevia 1 0
#> 3 Abraway~ Rodentia Cricetidae Abrawaya~ ruschii 1 0
#> 4 Abrocom~ Rodentia Abrocomid~ Abrocoma bennettii 1 0
```

```
#> 5 Abrocom~ Rodentia Abrocomid~ Abrocoma boliviensis      1      0
#> 6 Abrocom~ Rodentia Abrocomid~ Abrocoma budini           1      0
#> # ... with 5,825 more rows, and 17 more variables: Freshwater <dbl>,
#> #   Aerial <dbl>, Life.Habit.Method <chr>, Life.Habit.Source <chr>,
#> #   Mass.g <dbl>, Mass.Method <chr>, Mass.Source <chr>, Mass.Comparison <chr>,
#> #   Mass.Comparison.Source <chr>, Island.Endemicity <chr>,
#> #   IUCN.Status.1.2 <chr>, Added.IUCN.Status.1.2 <chr>, Diet.Plant <dbl>,
#> #   Diet.Vertebrate <dbl>, Diet.Invertebrate <dbl>, Diet.Method <chr>,
#> #   Diet.Source <chr>
```

The first argument is always `.data`, the data table you want to apply change to. Note how columns are referred to. Once the data table has been passed as an argument, there is no need to refer to it directly anymore, `dplyr` understands that you're dealing with variables inside that data frame. So drop that `data$var`, `data[, "var"]`, and forget the very existence of `attach()` / `detach()`.

You can refer to variables names either with strings or directly as objects, whether you're reading or creating them:

```
rename(
  phylacine,
  # this works
  binomial = Binomial.1.2
)
rename(
  phylacine,
  # this works too!
  binomial = "Binomial.1.2"
)
rename(
  phylacine,
  # guess what
  "binomial" = "Binomial.1.2"
)
```

I have applied similar changes to all variables in the dataset. Here is what the new names look like:

```
#> # A tibble: 5,831 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>      <dbl> <dbl>      <dbl> <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~      1      0          0      0
#> 2 Abeomel~ Rode~ Murid~ Abeo~ sevia      1      0          0      0
#> 3 Abraway~ Rode~ Crice~ Abra~ ruschii      1      0          0      0
#> 4 Abrocom~ Rode~ Abroc~ Abro~ bennet~      1      0          0      0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ bolivi~      1      0          0      0
#> 6 Abrocom~ Rode~ Abroc~ Abro~ budini      1      0          0      0
#> # ... with 5,825 more rows, and 15 more variables: life_habit_method <chr>,
```

```

572 #> #   life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
573 #> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
574 #> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
575 #> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
576 #> #   diet_method <chr>, diet_source <chr>

```

577 3.2.2 The pipe operator %>%

578 If you have already come across pieces of code using the tidyverse, chances are that you
 579 have seen this odd symbol. While the pipe is not strictly-speaking a part of the tidyverse
 580 (it comes from its own package, magrittr), it is imported along with each package and
 581 widely used in conjunction with its functions. What does it do? Consider the following
 582 example with `rename()`:

```

phylacine2 <- readr::read_csv("data/phylacine_traits.csv")
# regular syntax
rename(phylacine2, "binomial" = "Binomial.1.2")
#> # A tibble: 5,831 x 24
#>   binomial Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>      <chr>      <chr>      <chr>      <chr>      <dbl> <dbl>
#> 1 Abditom~ Rodentia Muridae Abditomys latidens      1      0
#> 2 Abeomel~ Rodentia Muridae Abeomelo~ sevia      1      0
#> 3 Abraway~ Rodentia Cricetidae Abrawaya~ ruschii      1      0
#> 4 Abrocom~ Rodentia Abrocomid~ Abrocoma bennettii      1      0
#> 5 Abrocom~ Rodentia Abrocomid~ Abrocoma boliviensis      1      0
#> 6 Abrocom~ Rodentia Abrocomid~ Abrocoma budini      1      0
#> # ... with 5,825 more rows, and 17 more variables: Freshwater <dbl>,
#> #   Aerial <dbl>, Life.Habit.Method <chr>, Life.Habit.Source <chr>,
#> #   Mass.g <dbl>, Mass.Method <chr>, Mass.Source <chr>, Mass.Comparison <chr>,
#> #   Mass.Comparison.Source <chr>, Island.Endemicity <chr>,
#> #   IUCN.Status.1.2 <chr>, Added.IUCN.Status.1.2 <chr>, Diet.Plant <dbl>,
#> #   Diet.Vertibrate <dbl>, Diet.Invertebrate <dbl>, Diet.Method <chr>,
#> #   Diet.Source <chr>
# alternative syntax with the pipe operator
phylacine2 %>% rename("binomial" = "Binomial.1.2")
#> # A tibble: 5,831 x 24
#>   binomial Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>      <chr>      <chr>      <chr>      <chr>      <dbl> <dbl>
#> 1 Abditom~ Rodentia Muridae Abditomys latidens      1      0
#> 2 Abeomel~ Rodentia Muridae Abeomelo~ sevia      1      0
#> 3 Abraway~ Rodentia Cricetidae Abrawaya~ ruschii      1      0
#> 4 Abrocom~ Rodentia Abrocomid~ Abrocoma bennettii      1      0
#> 5 Abrocom~ Rodentia Abrocomid~ Abrocoma boliviensis      1      0
#> 6 Abrocom~ Rodentia Abrocomid~ Abrocoma budini      1      0
#> # ... with 5,825 more rows, and 17 more variables: Freshwater <dbl>,
#> #   Aerial <dbl>, Life.Habit.Method <chr>, Life.Habit.Source <chr>,

```

```
#> #   Mass.g <dbl>, Mass.Method <chr>, Mass.Source <chr>, Mass.Comparison <chr>,
#> #   Mass.Comparison.Source <chr>, Island.Endemicity <chr>,
#> #   IUCN.Status.1.2 <chr>, Added.IUCN.Status.1.2 <chr>, Diet.Plant <dbl>,
#> #   Diet.Vertebrate <dbl>, Diet.Invertebrate <dbl>, Diet.Method <chr>,
#> #   Diet.Source <chr>
```

583 Got it? The pipe takes the object on its left-side and silently feeds it to the *first* argument
 584 of the function on its right-side. It could be read as “take x, then do...”. The reason for
 585 using the pipe is because it makes code syntax closer to the syntax of a sentence, and
 586 therefore, easier and faster for your brain to process (and write!) the code. In particular,
 587 the pipe enables easy chains of operations, where you apply something to an object, then
 588 apply something else to the outcome, and so on... Through the later sections, you will see
 589 some examples of chained operations with dplyr functions, but for that I first need to
 590 introduce a couple more verbs.

591 Using the pipe can be quite unsettling at first, because you are not used to think in this
 592 way. But if you push a bit for it, I promise it will make things a lot easier (and it's quite
 593 addictive!). To avoid typing the tedious symbols, magrittr installs a shortcut for you in
 594 RStudio. Use Ctrl + Shift + M on Windows, and Cmd + Shift + M on MacOS.

595 Finally I should emphasize that the use of the pipe isn't limited to the tidyverse, but
 596 extends to almost all R functions. Remember that by default the piped value is always
 597 matched to the first argument of the following function

```
5   %>% rep(3)
#> [1] 5 5 5
"meow" %>% cat()
#> meow
```

598 If you need to pass the left-hand side to an argument other than the first, you can use the
 599 dot place-holder ..

```
"meow" %>% cat("cats", "go")
#> meow cats go
```

600 Because of its syntax, most base R operators are not compatible with the pipe (but this is
 601 very rarely needed). If needed, magrittr introduces alternative functions for operators.

602 Subsetting operators can be piped, with the dot place-holder.

```
# 5 %>% * 3 # no, won't work
# 5 %>% .* 3 # neither
5 %>% magrittr::multiply_by(3) # yes
#> [1] 15

# subsetting
list("monkey see", "monkey do") %>% .[[2]]
#> [1] "monkey do"
phylacine %>% .$binomial %>% head()
```

```
#> [1] "Abditomys_latidens" "Abeomelomys_sevia" "Abrawayaomys_ruschii"
#> [4] "Abrocoma_bennettii" "Abrocoma_boliviensis" "Abrocoma_budini"
```

603 Because subsetting in this way is particularly hideous, dplyr delivers a function to extract
604 values from a single variable. In only works on tables, though.

```
phylacine %>% pull(binomial) %>% head()
#> [1] "Abditomys_latidens" "Abeomelomys_sevia" "Abrawayaomys_ruschii"
#> [4] "Abrocoma_bennettii" "Abrocoma_boliviensis" "Abrocoma_budini"
```

605 3.2.3 Select variables with select()

606 To extract a set of variables (i.e. columns), use the conveniently-named `select()`. The
607 basic syntax is the same as `rename()`: pass your data as the first argument, then call the
608 variables to select, quoted or not.

```
# Single variable
phylacine %>% select(binomial)
#> # A tibble: 5,831 x 1
#>   binomial
#>   <chr>
#> 1 Abditomys_latidens
#> 2 Abeomelomys_sevia
#> 3 Abrawayaomys_ruschii
#> 4 Abrocoma_bennettii
#> 5 Abrocoma_boliviensis
#> 6 Abrocoma_budini
#> # ... with 5,825 more rows
# A set of variables
phylacine %>% select(genus, "species", mass_g)
#> # A tibble: 5,831 x 3
#>   genus      species      mass_g
#>   <chr>      <chr>      <dbl>
#> 1 Abditomys latidens      269
#> 2 Abeomelomys sevia       52
#> 3 Abrawayaomys ruschii     63
#> 4 Abrocoma   bennettii    250
#> 5 Abrocoma   boliviensis  158
#> 6 Abrocoma   budini      361.
#> # ... with 5,825 more rows
# A range of contiguous variables
phylacine %>% select(family:terrestrial)
#> # A tibble: 5,831 x 4
#>   family      genus      species      terrestrial
#>   <chr>      <chr>      <chr>      <dbl>
#> 1 Muridae   Abditomys latidens      1
#> 2 Muridae   Abeomelomys sevia      1
```

```
#> 3 Cricetidae Abrawayaomys ruschii 1
#> 4 Abrocomidae Abrocoma bennettii 1
#> 5 Abrocomidae Abrocoma boliviensis 1
#> 6 Abrocomidae Abrocoma budini 1
#> # ... with 5,825 more rows
```

609 You can select by variable numbers. This is not recommended, as prone to errors, espe-
610 cially if you change the variable order.

```
phylacine %>% select(2)
#> # A tibble: 5,831 x 1
#>   order
#>   <chr>
#> 1 Rodentia
#> 2 Rodentia
#> 3 Rodentia
#> 4 Rodentia
#> 5 Rodentia
#> 6 Rodentia
#> # ... with 5,825 more rows
```

611 `select()` can also be used to *exclude* variables:

```
phylacine %>% select(-binomial)
#> # A tibble: 5,831 x 23
#>   order family genus species terrestrial marine freshwater aerial
#>   <chr> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl>
#> 1 Rode~ Murid~ Abdi~ latide~ 1 0 0 0
#> 2 Rode~ Murid~ Abeo~ sevia 1 0 0 0
#> 3 Rode~ Crice~ Abra~ ruschii 1 0 0 0
#> 4 Rode~ Abroc~ Abro~ bennet~ 1 0 0 0
#> 5 Rode~ Abroc~ Abro~ bolivi~ 1 0 0 0
#> 6 Rode~ Abroc~ Abro~ budini 1 0 0 0
#> # ... with 5,825 more rows, and 15 more variables: life_habit_method <chr>,
#> #   life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
phylacine %>% select(-(binomial:species))
#> # A tibble: 5,831 x 19
#>   terrestrial marine freshwater aerial life_habit_meth~ life_habit_sour~ mass_g
#>   <dbl> <dbl> <dbl> <dbl> <chr> <chr> <dbl>
#> 1 1 0 0 0 Reported IUCN. 2016. IUC~ 269
#> 2 1 0 0 0 Reported IUCN. 2016. IUC~ 52
#> 3 1 0 0 0 Reported IUCN. 2016. IUC~ 63
#> 4 1 0 0 0 Reported IUCN. 2016. IUC~ 250
#> 5 1 0 0 0 Reported IUCN. 2016. IUC~ 158
```

```
#> 6      1      0      0      0 Reported      IUCN. 2016. IUC~ 361.
#> # ... with 5,825 more rows, and 12 more variables: mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

612 `select()` and `rename()` are pretty similar, and in fact, `select()` can also rename vari-
613 ables along the way:

```
phylacine %>% select("linnaeus" = binomial)
#> # A tibble: 5,831 x 1
#>   linnaeus
#>   <chr>
#> 1 Abditomys_latidens
#> 2 Abeomelomys_sevia
#> 3 Abrawayaomys_ruschii
#> 4 Abrocoma_bennettii
#> 5 Abrocoma_boliviensis
#> 6 Abrocoma_budini
#> # ... with 5,825 more rows
```

614 And you can mix all of that at once:

```
phylacine %>% select(
  "fam" = family,
  genus:freshwater,
  -terrestrial
)
#> # A tibble: 5,831 x 5
#>   fam      genus      species      marine freshwater
#>   <chr>    <chr>    <chr>    <dbl>    <dbl>
#> 1 Muridae  Abditomys  latidens      0      0
#> 2 Muridae  Abeomelomys sevia      0      0
#> 3 Cricetidae Abrawayaomys ruschii      0      0
#> 4 Abrocomidae Abrocoma  bennettii      0      0
#> 5 Abrocomidae Abrocoma  boliviensis      0      0
#> 6 Abrocomidae Abrocoma  budini      0      0
#> # ... with 5,825 more rows
```

615 3.2.4 Select variables with helpers

616 The Rstudio team just released `dplyr 1.0.0`, and along with it, some nice helper func-
617 tions to ease the selection of a set of variables. I give three examples here, and encourage
618 you to look at the documentation (`?select()`) to find out more.

```
phylacine %>% select(where(is.numeric))
#> # A tibble: 5,831 x 8
```



```
#>   terrestrial marine freshwater aerial mass_g diet_plant diet_vertbrate
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1      1      0      0      0  269     100      0
#> 2      1      0      0      0   52      78      3
#> 3      1      0      0      0   63      88      1
#> 4      1      0      0      0  250     100      0
#> 5      1      0      0      0  158     100      0
#> 6      1      0      0      0  361.     100      0
#> # ... with 5,825 more rows, and 1 more variable: diet_invertebrate <dbl>
phylacine %>% select(contains("mass") | contains("diet"))
#> # A tibble: 5,831 x 10
#>   mass_g mass_method mass_source mass_comparison mass_comparison~ diet_plant
#>   <dbl> <chr> <chr> <chr> <chr> <dbl>
#> 1  269 Reported Smith, F. ~ <NA> <NA> 100
#> 2   52 Reported Smith, F. ~ <NA> <NA> 78
#> 3   63 Reported Smith, F. ~ <NA> <NA> 88
#> 4  250 Reported Smith, F. ~ <NA> <NA> 100
#> 5  158 Reported Smith, F. ~ <NA> <NA> 100
#> 6  361. Assumed is~ Journal of~ Abrocoma_ciner~ Journal of Mamm~ 100
#> # ... with 5,825 more rows, and 4 more variables: diet_vertbrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>

habitats <- c("terrestrial", "marine", "arboreal", "fossorial", "freshwater")
phylacine %>% select(any_of(habitats))
#> # A tibble: 5,831 x 3
#>   terrestrial marine freshwater
#>   <dbl> <dbl> <dbl>
#> 1      1      0      0
#> 2      1      0      0
#> 3      1      0      0
#> 4      1      0      0
#> 5      1      0      0
#> 6      1      0      0
#> # ... with 5,825 more rows
```

619 3.2.5 Rearranging variable order with relocate()

620 The order of variables seldom matters in dplyr, but due to popular demand, dplyr now
 621 has a dedicated verb to rearrange the order of variables. The syntax is identical to re-
 622 name(), select().

```
phylacine %>% relocate(mass_g, .before = binomial)
#> # A tibble: 5,831 x 24
#>   mass_g binomial order family genus species terrestrial marine freshwater
#>   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1  269 Abditom~ Rode~ Murid~ Abdi~ latide~ 1 0 0
```

```

#> 2 52 Abeomel~ Rode~ Murid~ Abeo~ sevia 1 0 0
#> 3 63 Abraway~ Rode~ Crice~ Abra~ ruschii 1 0 0
#> 4 250 Abrocom~ Rode~ Abroc~ Abro~ bennet~ 1 0 0
#> 5 158 Abrocom~ Rode~ Abroc~ Abro~ bolivi~ 1 0 0
#> 6 361. Abrocom~ Rode~ Abroc~ Abro~ budini 1 0 0
#> # ... with 5,825 more rows, and 15 more variables: aerial <dbl>,
#> # life_habit_method <chr>, life_habit_source <chr>, mass_method <chr>,
#> # mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> # island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> # diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> # diet_method <chr>, diet_source <chr>
phylacine %>% relocate(starts_with("diet"), .after = species)
#> # A tibble: 5,831 x 24
#> binomial order family genus species diet_plant diet_vertebrate
#> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~ 100 0
#> 2 Abeomel~ Rode~ Murid~ Abeo~ sevia 78 3
#> 3 Abraway~ Rode~ Crice~ Abra~ ruschii 88 1
#> 4 Abrocom~ Rode~ Abroc~ Abro~ bennet~ 100 0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ bolivi~ 100 0
#> 6 Abrocom~ Rode~ Abroc~ Abro~ budini 100 0
#> # ... with 5,825 more rows, and 17 more variables: diet_invertebrate <dbl>,
#> # diet_method <chr>, diet_source <chr>, terrestrial <dbl>, marine <dbl>,
#> # freshwater <dbl>, aerial <dbl>, life_habit_method <chr>,
#> # life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
#> # mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> # island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>

```

623 3.3 Working with observations

624 3.3.1 Ordering rows by value - arrange()

625 `arrange()` sorts rows in the data by **ascending** value for a given variable. Use the wrapper
 626 `desc()` to sort by descending values instead.

```

# Smallest mammals
phylacine %>%
  arrange(mass_g) %>%
  select(binomial, mass_g)
#> # A tibble: 5,831 x 2
#> binomial mass_g
#> <chr> <dbl>
#> 1 Sorex_yukonicus 1.6
#> 2 Crocidura_levicula 1.8
#> 3 Suncus_remyi 1.8
#> 4 Crocidura_lusitania 2

```

```

#> 5 Kerivoula_minuta      2.1
#> 6 Suncus_etruscus      2.1
#> # ... with 5,825 more rows

# Largest mammals
phylacine %>%
  arrange(desc(mass_g)) %>%
  select(binomial, mass_g)
#> # A tibble: 5,831 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Balaenoptera_musculus 1900000000
#> 2 Balaena_mysticetus 1000000000
#> 3 Balaenoptera_physalus 700000000
#> 4 Caperea_marginata 320000000
#> 5 Megaptera_novaeangliae 300000000
#> 6 Eschrichtius_robustus 285000000
#> # ... with 5,825 more rows

# Extra variables are used to sort ties in the first variable
phylacine %>%
  arrange(mass_g, desc(binomial)) %>%
  select(binomial, mass_g)
#> # A tibble: 5,831 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Sorex_yukonicus      1.6
#> 2 Suncus_remyi          1.8
#> 3 Crocidura_levicula    1.8
#> 4 Crocidura_lusitania    2
#> 5 Suncus_etruscus      2.1
#> 6 Kerivoula_minuta      2.1
#> # ... with 5,825 more rows

```

627 **Important:** NA values, if present, are always ordered at the end!

628 3.3.2 Subset rows by position - slice()

629 Use slice() and its variants to extract particular rows.

```

phylacine %>% slice(3) # third row
#> # A tibble: 1 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>      <dbl> <dbl>      <dbl> <dbl>
#> 1 Abraway~ Rode~ Crice~ Abra~ ruschii      1      0          0      0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> #   mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,

```

```

#> # mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> # added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> # diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
phylacine %>% slice(5, 1, 2) # fifth, first and second row
#> # A tibble: 3 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>          <dbl>  <dbl>          <dbl>  <dbl>
#> 1 Abrocom~ Rode~ Abroc~ Abro~ bolivi~      1    0              0    0
#> 2 Abditom~ Rode~ Murid~ Abdi~ latide~      1    0              0    0
#> 3 Abeomel~ Rode~ Murid~ Abeo~ sevia  1    0              0    0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> # mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> # mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> # added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> # diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
phylacine %>% slice(rep(3, 2)) # duplicate the third row
#> # A tibble: 2 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>          <dbl>  <dbl>          <dbl>  <dbl>
#> 1 Abaway~ Rode~ Crice~ Abra~ ruschii      1    0              0    0
#> 2 Abaway~ Rode~ Crice~ Abra~ ruschii      1    0              0    0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> # mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> # mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> # added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> # diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
phylacine %>% slice(-c(2:5830)) # exclude all but first and last row
#> # A tibble: 2 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>          <dbl>  <dbl>          <dbl>  <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~      1    0              0    0
#> 2 Zyzomys~ Rode~ Murid~ Zyzo~ woodwa~      1    0              0    0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> # mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> # mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> # added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> # diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>

phylacine %>% slice_tail(n = 3) # last three rows
#> # A tibble: 3 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>          <dbl>  <dbl>          <dbl>  <dbl>
#> 1 Zyzomys~ Rode~ Murid~ Zyzo~ palata~      1    0              0    0
#> 2 Zyzomys~ Rode~ Murid~ Zyzo~ pedunc~      1    0              0    0
#> 3 Zyzomys~ Rode~ Murid~ Zyzo~ woodwa~      1    0              0    0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,

```

```
#> # mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> # mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> # added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> # diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
phylacine %>% slice_max(mass_g) # largest mammal
#> # A tibble: 1 x 24
#> binomial order family genus species terrestrial marine freshwater aerial
#> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl>
#> 1 Balaeno~ Ceta~ Bala~ Bala~ muscul~ 0 1 0 0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> # mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> # mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> # added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> # diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
```

630 You can also sample random rows in the data:

```
phylacine %>% slice_sample() # a random row
#> # A tibble: 1 x 24
#> binomial order family genus species terrestrial marine freshwater aerial
#> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl>
#> 1 Crocidu~ Euli~ Soric~ Croc~ levicu~ 1 0 0 0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> # mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> # mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> # added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> # diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>

# bootstrap
phylacine %>% slice_sample(n = 5831, replace = TRUE)
#> # A tibble: 5,831 x 24
#> binomial order family genus species terrestrial marine freshwater aerial
#> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl>
#> 1 Rhinolo~ Chir~ Rhino~ Rhin~ adami 0 0 0 1
#> 2 Hylomys~ Euli~ Erina~ Hylo~ megal~ 1 0 0 0
#> 3 Sciurus~ Rode~ Sciur~ Sciu~ yucata~ 1 0 0 0
#> 4 Emballo~ Chir~ Embal~ Emba~ alecto 0 0 0 1
#> 5 Pteralo~ Chir~ Ptero~ Pter~ taki 0 0 0 1
#> 6 Lasiorh~ Dipr~ Vomba~ Lasi~ latifr~ 1 0 0 0
#> # ... with 5,825 more rows, and 15 more variables: life_habit_method <chr>,
#> # life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
#> # mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> # island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> # diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
#> # diet_method <chr>, diet_source <chr>
```

631 3.3.3 Subsetting rows by value with `filter()`

632 `filter()` does a similar job as `slice()`, but extract rows that satisfy a set of conditions.
 633 The conditions are supplied much the same way as you would do for an `if` statement.

634 Along with `mutate()` (next section), this is probably the function you are going to use the
 635 most.

636 For example, I might want to extract mammals above a given mass:

```
# mega fauna
phylacine %>%
  filter(mass_g > 1e5) %>% # 100 kg
  select(binomial, mass_g)
#> # A tibble: 302 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Ailuropoda_melanoleuca 108400
#> 2 Alcelaphus_buselaphus 171002.
#> 3 Alces_alces          356998
#> 4 Archaeoindris_fontoynonti 160000
#> 5 Arctocephalus_forsteri 101250
#> 6 Arctocephalus_pusillus 178500
#> # ... with 296 more rows

# non-extinct mega fauna
phylacine %>%
  filter(mass_g > 1e5, iucn_status != "EP") %>%
  select(binomial, mass_g, iucn_status)
#> # A tibble: 178 x 3
#>   binomial      mass_g iucn_status
#>   <chr>      <dbl> <chr>
#> 1 Ailuropoda_melanoleuca 108400 VU
#> 2 Alcelaphus_buselaphus 171002. LC
#> 3 Alces_alces          356998 LC
#> 4 Arctocephalus_forsteri 101250 LC
#> 5 Arctocephalus_pusillus 178500 LC
#> 6 Arctocephalus_townsendi 105000 LC
#> # ... with 172 more rows
```

637 Are there any flying mammals that aren't bats?

```
phylacine %>%
  filter(aerial == 1, order != "Chiroptera")
#> # A tibble: 0 x 24
#> # ... with 24 variables: binomial <chr>, order <chr>, family <chr>,
#> #   genus <chr>, species <chr>, terrestrial <dbl>, marine <dbl>,
#> #   freshwater <dbl>, aerial <dbl>, life_habit_method <chr>,
```

```
#> #   life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
# no :(
```

638 Are humans included in the table?

```
phylacine %>% filter(binomial == "Homo_sapiens")
#> # A tibble: 1 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>          <dbl>  <dbl>          <dbl>  <dbl>
#> 1 Homo_sa~ Prim~ Homin~ Homo sapiens          1      0              0      0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> #   mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> #   mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
```

639 filter() can be used to deal with NAs:

```
phylacine %>%
  filter(!is.na(mass_comparison))
#> # A tibble: 754 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr> <chr>          <dbl>  <dbl>          <dbl>  <dbl>
#> 1 Abrocom~ Rode~ Abroc~ Abro~ budini          1      0              0      0
#> 2 Abrocom~ Rode~ Abroc~ Abro~ famati~          1      0              0      0
#> 3 Abrocom~ Rode~ Abroc~ Abro~ shista~          1      0              0      0
#> 4 Abrocom~ Rode~ Abroc~ Abro~ uspaill~          1      0              0      0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ vaccar~          1      0              0      0
#> 6 Acerodo~ Chir~ Ptero~ Acer~ humilis          0      0              0      1
#> # ... with 748 more rows, and 15 more variables: life_habit_method <chr>,
#> #   life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

640 Tip: dplyr introduces the useful function between() that does exactly what the name
641 implies

```
between(1:5, 2, 4)
#> [1] FALSE TRUE TRUE TRUE FALSE

# Mesofauna
phylacine %>%
  filter(mass_g > 1e3, mass_g < 1e5) %>%
```

```

select(binomial, mass_g)
#> # A tibble: 1,126 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Acerodon_jubatus    1075
#> 2 Acinonyx_jubatus   46700
#> 3 Acratocnus_odontrigonus 22990
#> 4 Acratocnus_ye      21310
#> 5 Addax_nasomaculatus   70000.
#> 6 Aepyceros_melampus   52500.
#> # ... with 1,120 more rows

# same thing
phylacine %>%
  filter(mass_g %>% between(1e3, 1e5)) %>%
  select(binomial, mass_g)
#> # A tibble: 1,148 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Acerodon_jubatus    1075
#> 2 Acinonyx_jubatus   46700
#> 3 Acratocnus_odontrigonus 22990
#> 4 Acratocnus_ye      21310
#> 5 Addax_nasomaculatus   70000.
#> 6 Aepyceros_melampus   52500.
#> # ... with 1,142 more rows

```

642 Note that you can pipe operations inside function arguments as in the last line above (ar-
 643 guments are expressions, after all!).

644 3.4 Making new variables

645 3.4.1 Create new variables with mutate()

646 Very often in data analysis, you will want to create new variables, or edit existing ones.
 647 This is done easily through `mutate()`. For example, consider the diet data:

```

diet <- phylacine %>%
  select(
    binomial,
    contains("diet") & !contains(c("method", "source"))
  )
diet
#> # A tibble: 5,831 x 4
#>   binomial      diet_plant diet_vertibrate diet_invertebrate
#>   <chr>      <dbl>      <dbl>      <dbl>

```



```
#> 1 Abditomys_latidens      100      0      0
#> 2 Abeomelomys_sevia       78      3     19
#> 3 Abrawayaomys_ruschi     88      1     11
#> 4 Abrocoma_bennettii     100      0      0
#> 5 Abrocoma_boliviensis   100      0      0
#> 6 Abrocoma_budini       100      0      0
#> # ... with 5,825 more rows
```

648 These three variables show the percentage of each category of food that make the diet of
 649 that species. They should sum to 100, unless the authors made a typo or other entry error.
 650 To assert this, I'm going to create a new variable, `total_diet`.

```
diet <- diet %>% mutate(
  "total_diet" = diet_vertibrate + diet_invertebrate + diet_plant
)
diet
#> # A tibble: 5,831 x 5
#>   binomial      diet_plant diet_vertibrate diet_invertebrate total_diet
#>   <chr>          <dbl>          <dbl>          <dbl>          <dbl>
#> 1 Abditomys_latidens      100            0            0            100
#> 2 Abeomelomys_sevia       78            3           19            100
#> 3 Abrawayaomys_ruschi     88            1           11            100
#> 4 Abrocoma_bennettii     100            0            0            100
#> 5 Abrocoma_boliviensis   100            0            0            100
#> 6 Abrocoma_budini       100            0            0            100
#> # ... with 5,825 more rows

all(diet$total_diet == 100)
#> [1] TRUE
# cool and good
```

651 `mutate()` adds a variable to the table, and keeps all other variables. Sometimes you may
 652 want to just keep the new variable, and drop the other ones. That's the job of `mutate()`'s
 653 twin sibling, `transmute()`. For example, I want to combine `diet_invertebrate` and
 654 `diet_vertibrate` together:

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertibrate
  )
#> # A tibble: 5,831 x 1
#>   diet_animal
#>   <dbl>
#> 1      0
#> 2     22
#> 3     12
#> 4      0
#> 5      0
```

```
#> 6          0
#> # ... with 5,825 more rows
```

655 You may want to keep some variables and drop others. You could pipe `mutate()` and
656 `select()` to do so, or you could just pass the variables to keep to `transmute()`.

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertebrate,
    diet_plant
  )
#> # A tibble: 5,831 x 2
#>   diet_animal diet_plant
#>   <dbl>      <dbl>
#> 1         0        100
#> 2        22         78
#> 3        12         88
#> 4         0        100
#> 5         0        100
#> 6         0        100
#> # ... with 5,825 more rows
```

657 You can also refer to variables you're creating to derive new variables from them as part
658 of the same operation, this is not an issue.

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertebrate,
    diet_plant,
    "total_diet" = diet_animal + diet_plant
  )
#> # A tibble: 5,831 x 3
#>   diet_animal diet_plant total_diet
#>   <dbl>      <dbl>      <dbl>
#> 1         0        100        100
#> 2        22         78        100
#> 3        12         88        100
#> 4         0        100        100
#> 5         0        100        100
#> 6         0        100        100
#> # ... with 5,825 more rows
```

659 Sometimes, you may need to perform an operation based on the row number (I don't have
660 a good example in mind). `tibble` has a built-in function to do just that:

```
phylacine %>%
  select(binomial) %>%
  tibble::rownames_to_column(var = "row_nb")
#> # A tibble: 5,831 x 2
```

```
#> row_nb binomial
#> <chr> <chr>
#> 1 1 Abditomys_latidens
#> 2 2 Abeomelomys_sevia
#> 3 3 Abrawayaomys_ruschi
#> 4 4 Abrocoma_bennettii
#> 5 5 Abrocoma_boliviensis
#> 6 6 Abrocoma_budini
#> # ... with 5,825 more rows
```

661 3.4.2 Summarise observations with summarise()

662 `mutate()` applies operations to all observations in a table. By contrast, `summarise()` ap-
 663 plies operations to *groups* of observations, and returns, er, summaries. The default group-
 664 ing unit is the entire table:

```
phylacine %>%
  summarise(
    "nb_species" = n(), # counts observations
    "nb_terrestrial" = sum(terrestrial),
    "nb_marine" = sum(marine),
    "nb_freshwater" = sum(freshwater),
    "nb_aerial" = sum(aerial),
    "mean_mass_g" = mean(mass_g)
  )
#> # A tibble: 1 x 6
#>   nb_species nb_terrestrial nb_marine nb_freshwater nb_aerial mean_mass_g
#>   <int>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
#> 1     5831      4575      135      156      1162     156882.
```

665 Above you can see that bats account for a large portion of mammal species diversity
 666 (`nb_aerial`). How much exactly? Just as with `mutate()`, you can perform operations
 667 on the variables you just created, in the same statement:

```
phylacine %>%
  summarise(
    "nb_species" = n(),
    "nb_aerial" = sum(aerial), # bats
    "prop_aerial" = nb_aerial / nb_species
  )
#> # A tibble: 1 x 3
#>   nb_species nb_aerial prop_aerial
#>   <int>      <dbl>      <dbl>
#> 1     5831     1162      0.199
```

668 One fifth!

669 If the british spelling bothers you, `summarize()` exists and is strictly equivalent.

Here's a simple trick with logical (TRUE / FALSE) variables. Their sum is the count of observations that evaluate to TRUE (because TRUE is taken as 1 and FALSE as 0) and their mean is the proportion of TRUE observations. This can be exploited to count the number of observations that satisfy a condition:

```
phylacine %>%
  summarise(
    "nb_species" = n(),
    "nb_megafauna" = sum(mass_g > 100000),
    "p_megafauna" = mean(mass_g > 100000)
  )
#> # A tibble: 1 x 3
#>   nb_species nb_megafauna p_megafauna
#>   <int>      <int>      <dbl>
#> 1      5831         302      0.0518
```

There are more summaries that just means and counts (see `?summarise()` for some helpful functions). In fact, `summarise` can use any function or expression that evaluates to a single value or a *vector* of values. This includes base R `max()`, `quantiles`, etc.

`mutate()` and `transmute()` can compute summaries as well, but they will return the summary once for each observation, in a new column.

```
phylacine %>%
  mutate("nb_species" = n()) %>%
  select(binomial, nb_species)
#> # A tibble: 5,831 x 2
#>   binomial          nb_species
#>   <chr>          <int>
#> 1 Abditomys_latidens      5831
#> 2 Abeomelomys_sevia      5831
#> 3 Abrawayaomys_ruschii   5831
#> 4 Abrocoma_bennettii     5831
#> 5 Abrocoma_boliviensis   5831
#> 6 Abrocoma_budini       5831
#> # ... with 5,825 more rows
```

3.4.3 Grouping observations by variables

In most cases you don't want to run summary operations on the entire set of observations, but instead on observations that share a common value, i.e. groups. For example, I want to run the summary displayed above, but for each Order of mammals.

`distinct()` extracts all the unique values of a variable

```
phylacine %>% distinct(order)
#> # A tibble: 29 x 1
#>   order
#>   <chr>
```

```
#> 1 Rodentia
#> 2 Chiroptera
#> 3 Carnivora
#> 4 Pilosa
#> 5 Diprotodontia
#> 6 Cetartiodactyla
#> # ... with 23 more rows
```

684 I could work my way with what we have already seen, filtering observations
 685 (`filter(order == "Rodentia")`) and then piping the output to `summarise()`,
 686 and do it again for each Order. But that would be tedious.

687 Instead, I can use `group_by()` to pool observations by order.

```
phylacine %>%
  group_by(order)
#> # A tibble: 5,831 x 24
#> # Groups:   order [29]
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr> <chr> <chr>         <dbl>  <dbl>         <dbl>  <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~         1      0             0      0
#> 2 Abeomel~ Rode~ Murid~ Abeo~ sevia         1      0             0      0
#> 3 Abraway~ Rode~ Crice~ Abra~ ruschii        1      0             0      0
#> 4 Abrocom~ Rode~ Abroc~ Abro~ bennet~        1      0             0      0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ bolivi~        1      0             0      0
#> 6 Abrocom~ Rode~ Abroc~ Abro~ budini         1      0             0      0
#> # ... with 5,825 more rows, and 15 more variables: life_habit_method <chr>,
#> #   life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

688 At first glance, nothing has changed, apart from an extra line of information in the output
 689 that tells me the observations have been grouped. But now here's what happen if I run
 690 the same `summarise()` statement on an ungrouped and a grouped table

```
phylacine %>%
  summarise(
    "n_species" = n(),
    "mean_mass_g" = mean(mass_g)
  )
#> # A tibble: 1 x 2
#>   n_species mean_mass_g
#>   <int>      <dbl>
#> 1     5831    156882.
```

```
phylacine %>%
```

```

group_by(order) %>%
  summarise(
    "n_species" = n(),
    "mean_mass_g" = mean(mass_g)
  )
#> # A tibble: 29 x 3
#>   order          n_species mean_mass_g
#>   <chr>          <int>      <dbl>
#> 1 Afrosoricida      57        306.
#> 2 Carnivora        313       47905.
#> 3 Cetartiodactyla   392      1854811.
#> 4 Chiroptera       1162        49.1
#> 5 Cingulata         39      235529.
#> 6 Dasyuromorphia    74        748.
#> # ... with 23 more rows

```

691 I get one value for each group.

692 Observations can be grouped by multiple variables, which will output a summary for every
 693 unique combination of groups.

```

phylacine %>%
  group_by(order, iucn_status) %>%
  summarise(
    "n_species" = n()
  )
#> # A tibble: 138 x 3
#> # Groups:   order [29]
#>   order          iucn_status n_species
#>   <chr>          <chr>          <int>
#> 1 Afrosoricida CR              1
#> 2 Afrosoricida DD              4
#> 3 Afrosoricida EN              7
#> 4 Afrosoricida EP              2
#> 5 Afrosoricida LC             32
#> 6 Afrosoricida NT              3
#> # ... with 132 more rows

```

694 Whenever you call summarise(), the last level of grouping is dropped. Note how in the
 695 output table above, observations are still grouped by order, and no longer by IUCN status.
 696 If I summarise observations again:

```

phylacine %>%
  group_by(order, iucn_status) %>%
  summarise(
    "n_species" = n()
  ) %>%
  summarise(

```

```

    "n_species_2" = n()
  )
#> # A tibble: 29 x 2
#>   order      n_species_2
#>   <chr>          <int>
#> 1 Afrosoricida         7
#> 2 Carnivora            8
#> 3 Cetartiodactyla      9
#> 4 Chiroptera           8
#> 5 Cingulata            5
#> 6 Dasyuromorphia       7
#> # ... with 23 more rows

```

697 I get the summary across orders, and the table is no longer grouped at all. This is useful
 698 to consider if you need to work on summaries across different levels of the data.

699 For example, I would like to know how the species in each order are distributed between
 700 the different levels of threat in the IUCN classification. To get these proportions, I need to
 701 first get the count of each number of species in a level of threat inside an order, and divide
 702 that by the number of species in that order.

```

phylacine %>%
  group_by(order, iucn_status) %>%
  summarise("n_order_iucn" = n()) %>%
  # grouping by iucn_status silently dropped
  mutate(
    "n_order" = sum(n_order_iucn),
    "p_iucn" = n_order_iucn / n_order
  )
#> # A tibble: 138 x 5
#> # Groups:   order [29]
#>   order      iucn_status n_order_iucn n_order p_iucn
#>   <chr>          <chr>          <int>   <int>   <dbl>
#> 1 Afrosoricida CR              1       57 0.0175
#> 2 Afrosoricida DD              4       57 0.0702
#> 3 Afrosoricida EN              7       57 0.123
#> 4 Afrosoricida EP              2       57 0.0351
#> 5 Afrosoricida LC             32       57 0.561
#> 6 Afrosoricida NT              3       57 0.0526
#> # ... with 132 more rows

```

703 10.2% of Carnivores are Endangered ("EN").

704 3.4.4 Grouped data and other dplyr verbs

705 Grouping does not only affect the behaviour of summarise, but under circumstances,
 706 other verbs can (and will!) perform operations by groups.

```

# Species with a higher mass than the mammal mean
phylacine %>%
  select("binomial", "mass_g") %>%
  filter(mass_g > mean(mass_g, na.rm = TRUE))
#> # A tibble: 234 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Alcelaphus_buselaphus 171002.
#> 2 Alces_alces          356998
#> 3 Archaeoindris_fontoynonti 160000
#> 4 Arctocephalus_pusillus 178500
#> 5 Arctodus_simus        709500
#> 6 Balaena_mysticetus    100000000
#> # ... with 228 more rows

# Species with a higher mass than the mean in their order
phylacine %>%
  group_by(order) %>%
  select("binomial", "mass_g") %>%
  filter(mass_g > mean(mass_g, na.rm = TRUE))
#> # A tibble: 890 x 3
#> # Groups:   order [27]
#>   order      binomial      mass_g
#>   <chr>      <chr>      <dbl>
#> 1 Chiroptera Acerodon_celebensis 390
#> 2 Chiroptera Acerodon_humilis    600.
#> 3 Chiroptera Acerodon_jubatus   1075
#> 4 Chiroptera Acerodon_leucotis   513.
#> 5 Chiroptera Acerodon_mackloti   470.
#> 6 Rodentia  Aeretes_melanopterus  732.
#> # ... with 884 more rows

# Largest mammal
phylacine %>%
  select(binomial, mass_g) %>%
  slice_max(mass_g)
#> # A tibble: 1 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Balaenoptera_musculus 1900000000
# Largest species in each order
phylacine %>%
  group_by(order) %>%
  select(binomial, mass_g) %>%
  slice_max(mass_g)
#> # A tibble: 30 x 3

```



```

#> # Groups:   order [29]
#>   order      binomial      mass_g
#>   <chr>      <chr>      <dbl>
#> 1 Afrosoricida Plesiorycteropus_madagascariensis 13220
#> 2 Carnivora    Mirounga_leonina      1600000
#> 3 Cetartiodactyla Balaenoptera_musculus 190000000
#> 4 Chiroptera    Acerodon_jubatus      1075
#> 5 Cingulata     Glyptodon_clavipes     2000000
#> 6 Dasyuromorphia Thylacinus_cynocephalus 30000
#> # ... with 24 more rows

```

707 To avoid grouped operations, you can simply drop grouping with `ungroup()`.

708 3.5 Working with multiple tables

709 3.5.1 Binding tables

710 `dplyr` introduces `bind_rows()` and `bind_cols()`, which are equivalent to base R
 711 `rbind()` and `cbind()`, with a few extra feature. They are faster, and can bind many
 712 tables at once, and bind data frames with vectors or lists.

713 `bind_rows()` has an option to pass a variable specifying which dataset each observation
 714 originates from.

```

porpoises <- phylacine %>%
  filter(family == "Phocoenidae") %>%
  select(binomial, iucn_status)
echidnas <- phylacine %>%
  filter(family == "Tachyglossidae") %>%
  select(binomial, iucn_status)

bind_rows(
  "porpoise" = porpoises,
  "echidna" = echidnas,
  .id = "kind"
)
#> # A tibble: 13 x 3
#>   kind      binomial      iucn_status
#>   <chr>      <chr>      <chr>
#> 1 porpoise Neophocaena_asiaeorientalis VU
#> 2 porpoise Neophocaena_phocaenoides VU
#> 3 porpoise Phocoena_dioptrica DD
#> 4 porpoise Phocoena_phocoena LC
#> 5 porpoise Phocoena_sinus CR
#> 6 porpoise Phocoena_spinipinnis DD
#> # ... with 7 more rows

```

715 3.5.2 Combining variables of two tables with mutating joins

716 Mutating joins are tailored to combine tables that share a set of observations but have
717 different variables.

718 As an example, let's split the phylacine dataset in two smaller datasets, one containing
719 information on diet and one on the dominant habitat.

```
diet <- phylacine %>%
  select(binomial, diet_plant:diet_invertebrate) %>%
  slice(1:5)
diet
#> # A tibble: 5 x 4
#>   binomial      diet_plant diet_vertbrate diet_invertebrate
#>   <chr>          <dbl>         <dbl>         <dbl>
#> 1 Abditomys_latidens      100             0             0
#> 2 Abeomelomys_sevia       78             3            19
#> 3 Abrawayaomys_ruschii     88             1            11
#> 4 Abrocoma_bennettii     100             0             0
#> 5 Abrocoma_boliviensis   100             0             0
```

```
life_habit <- phylacine %>% select(binomial, terrestrial:aerial) %>%
  slice(1:3, 6:7)
life_habit
#> # A tibble: 5 x 5
#>   binomial      terrestrial marine freshwater aerial
#>   <chr>          <dbl>    <dbl>         <dbl>    <dbl>
#> 1 Abditomys_latidens      1      0             0      0
#> 2 Abeomelomys_sevia      1      0             0      0
#> 3 Abrawayaomys_ruschii    1      0             0      0
#> 4 Abrocoma_budini        1      0             0      0
#> 5 Abrocoma_cinerea        1      0             0      0
```

720 The two datasets each contain 5 species, the first three are shared, and the two last differ
721 between the two.

```
intersect(diet$binomial, life_habit$binomial)
#> [1] "Abitomys_latidens" "Abeomelomys_sevia" "Abrawayaomys_ruschii"
setdiff(diet$binomial, life_habit$binomial)
#> [1] "Abrocoma_bennettii" "Abrocoma_boliviensis"
```

722 To use mutate-joins, both tables need to have a **key**, a variable that identifies each obser-
723 vation. Here, that would be binomial, the species names. If your table doesn't have such
724 a key and the rows between the tables match one another, remember you can create a row
725 number variable easily with `tibble::column_to_rownames()`.

```
inner_join(diet, life_habit, by = "binomial")
#> # A tibble: 3 x 8
#>   binomial diet_plant diet_vertbrate diet_invertebra~ terrestrial marine
```

```
#>   <chr>           <dbl>           <dbl>           <dbl>           <dbl> <dbl>
#> 1 Abditom~      100             0             0             1      0
#> 2 Abeomel~      78             3             19            1      0
#> 3 Abraway~      88             1             11            1      0
#> # ... with 2 more variables: freshwater <dbl>, aerial <dbl>
```

726 `inner_join` combined the variables, and dropped the observations that weren't matched
 727 between the two tables. There are three other variations of mutating joins, differing in
 728 what they do with unmatching variables.

```
left_join(diet, life_habit, by = "binomial")
```

```
#> # A tibble: 5 x 8
#>   binomial diet_plant diet_vertibrate diet_invertebra~ terrestrial marine
#>   <chr>           <dbl>           <dbl>           <dbl>           <dbl> <dbl>
#> 1 Abditom~      100             0             0             1      0
#> 2 Abeomel~      78             3             19            1      0
#> 3 Abraway~      88             1             11            1      0
#> 4 Abrocom~     100             0             0             NA     NA
#> 5 Abrocom~     100             0             0             NA     NA
#> # ... with 2 more variables: freshwater <dbl>, aerial <dbl>
```

```
right_join(diet, life_habit, by = "binomial")
```

```
#> # A tibble: 5 x 8
#>   binomial diet_plant diet_vertibrate diet_invertebra~ terrestrial marine
#>   <chr>           <dbl>           <dbl>           <dbl>           <dbl> <dbl>
#> 1 Abditom~      100             0             0             1      0
#> 2 Abeomel~      78             3             19            1      0
#> 3 Abraway~      88             1             11            1      0
#> 4 Abrocom~      NA             NA            NA            1      0
#> 5 Abrocom~      NA             NA            NA            1      0
#> # ... with 2 more variables: freshwater <dbl>, aerial <dbl>
```

```
full_join(diet, life_habit, by = "binomial")
```

```
#> # A tibble: 7 x 8
#>   binomial diet_plant diet_vertibrate diet_invertebra~ terrestrial marine
#>   <chr>           <dbl>           <dbl>           <dbl>           <dbl> <dbl>
#> 1 Abditom~      100             0             0             1      0
#> 2 Abeomel~      78             3             19            1      0
#> 3 Abraway~      88             1             11            1      0
#> 4 Abrocom~     100             0             0             NA     NA
#> 5 Abrocom~     100             0             0             NA     NA
#> 6 Abrocom~      NA             NA            NA            1      0
#> # ... with 1 more row, and 2 more variables: freshwater <dbl>, aerial <dbl>
```

```
semi_join(diet, life_habit, by = "binomial")
```

```
#> # A tibble: 3 x 4
#>   binomial diet_plant diet_vertibrate diet_invertebrate
#>   <chr>           <dbl>           <dbl>           <dbl>
#> 1 Abditomys_latidens 100             0             0
```

```

#> 2 Abeomelomys_sevia          78          3          19
#> 3 Abrawayaomys_ruschii       88          1          11
anti_join(diet, life_habit, by = "binomial")
#> # A tibble: 2 x 4
#>   binomial      diet_plant diet_vertibrate diet_invertibrate
#>   <chr>          <dbl>          <dbl>          <dbl>
#> 1 Abrocoma_bennettii      100            0            0
#> 2 Abrocoma_boliviensis    100            0            0

```

729 3.5.3 Filtering matching observations between two tables with filter- 730 ing joins

731 So-called filtering joins return row from the first table that are matched (or not, for
732 anti_join()) in the second.

```

semi_join(diet, life_habit, by = "binomial")
#> # A tibble: 3 x 4
#>   binomial      diet_plant diet_vertibrate diet_invertibrate
#>   <chr>          <dbl>          <dbl>          <dbl>
#> 1 Abditomys_latidens      100            0            0
#> 2 Abeomelomys_sevia       78            3            19
#> 3 Abrawayaomys_ruschii    88            1            11
anti_join(diet, life_habit, by = "binomial")
#> # A tibble: 2 x 4
#>   binomial      diet_plant diet_vertibrate diet_invertibrate
#>   <chr>          <dbl>          <dbl>          <dbl>
#> 1 Abrocoma_bennettii      100            0            0
#> 2 Abrocoma_boliviensis    100            0            0

```

733 Chapter 4

734 Working with lists and iteration



Every use case is ridiculous
until it happens to you.

735

```
# load the tidyverse  
library(tidyverse)
```

736 4.1 List columns with tidyr

737 4.1.1 Nesting data

738 It may become necessary to indicate the groups of a tibble in a somewhat more explicit
739 way than simply using `dplyr::group_by`. `tidyr` offers the option to create nested tib-
740 bles, that is, to store complex objects in the columns of a tibble. This includes other tib-
741 bles, as well as model objects and plots.

742 *NB:* Nesting data is done using `tidyr::nest`, which is different from the similarly named
743 `tidyr::nesting`.

744 The example below shows how *Phylacine* data can be converted into a nested tibble.

```
# get phylacine data
data = read_csv("data/phylacine_traits.csv")
data = data %>%
  `colnames<-` (str_to_lower(colnames(.))) %>%
  `colnames<-` (str_remove(colnames(.), "(.1.2)")) %>%
  `colnames<-` (str_replace_all(colnames(.), "\\.", "_"))

# nest phylacine by order
nested_data = data %>%
  group_by(order) %>%
  nest()

nested_data
#> # A tibble: 29 x 2
#> # Groups:   order [29]
#>   order      data
#>   <chr>      <list>
#> 1 Rodentia   <tibble [2,306 x 23]>
#> 2 Chiroptera <tibble [1,162 x 23]>
#> 3 Carnivora  <tibble [313 x 23]>
#> 4 Pilosa     <tibble [34 x 23]>
#> 5 Diprotodontia <tibble [183 x 23]>
#> 6 Cetartiodactyla <tibble [392 x 23]>
#> # ... with 23 more rows

# get column class
sapply(nested_data, class)
#>   order      data
#> "character"    "list"
```

745 The data is now a nested data frame. The class of each of its columns is respectively, a
746 character (order name) and a list (the data of all mammals in the corresponding order).

747 While `nest` can be used without first grouping the tibble, it's just much easier to group
748 first.

749 4.1.2 Unnesting data

750 A nested tibble can be converted back into the original, or into a processed form, using
751 `tidyr::unnest`. The original groups are retained.

```
# use unnest to recover the original data frame
unnest(nested_data, cols = "data") %>%
  head()
#> # A tibble: 6 x 24
#> # Groups:   order [1]
```

```
#>   order binomial family genus species terrestrial marine freshwater aerial
#>   <chr> <chr>      <chr> <chr> <chr>          <dbl>  <dbl>          <dbl>  <dbl>
#> 1 Rode~ Abditom~ Murid~ Abdi~ latide~          1      0              0      0
#> 2 Rode~ Abeomel~ Murid~ Abeo~ sevia          1      0              0      0
#> 3 Rode~ Abraway~ Crice~ Abra~ ruschii        1      0              0      0
#> 4 Rode~ Abrocom~ Abroc~ Abro~ bennet~        1      0              0      0
#> 5 Rode~ Abrocom~ Abroc~ Abro~ bolivi~        1      0              0      0
#> 6 Rode~ Abrocom~ Abroc~ Abro~ budini         1      0              0      0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> #   mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> #   mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>

# unnesting preserves groups
groups(unnest(nested_data, cols = "data"))
#> [[1]]
#> order
```

752 The `unnest_longer` and `unnest_wider` variants of `unnest` are maturing functions, that
 753 is, not in their final form. They allow interesting variations on unnesting — these are
 754 shown here but advised against. Unnest the data first, and then convert it to the form
 755 needed.

756 4.1.3 Working with list columns

757 The class of a list column is `list`, and working with list columns (and lists, and list-like
 758 objects such as vectors) makes iteration necessary, since this is one of the only ways to
 759 operate on lists.

760 Two examples are shown below when getting the class and number of rows of the nested
 761 tibbles in the list column.

```
# how many rows in each nested tibble?
for (i in seq_along(nested_data$data[1:10])) {
  print(nrow(nested_data$data[[i]]))
}
#> [1] 2306
#> [1] 1162
#> [1] 313
#> [1] 34
#> [1] 183
#> [1] 392
#> [1] 460
#> [1] 57
#> [1] 20
#> [1] 465
```

```

# what is the class of each element?
lapply(X = nested_data$data[1:3], FUN = class)
#> [[1]]
#> [1] "tbl_df"      "tbl"        "data.frame"
#>
#> [[2]]
#> [1] "tbl_df"      "tbl"        "data.frame"
#>
#> [[3]]
#> [1] "tbl_df"      "tbl"        "data.frame"

```

762 Functionals

763 The second example uses `lapply`, and this is a *functional*. *Functionals* are functions that
 764 take another function as one of their arguments. Base R functionals include the `*apply`
 765 family of functions: `apply`, `lapply`, `vapply` and so on.

766 4.2 Iteration with map

767 The tidyverse replaces traditional loop-based iteration with *functionals* from the `purrr`
 768 package.

769 Why use purrr

770 A good reason to use `purrr` functionals instead of base R functionals is their consistent
 771 and clear naming, which always indicates how they should be used. This is explained in
 772 the examples below. How `map` is different from `for` and `lapply` are best explained in the
 773 **Advanced R Book**.

774 4.2.1 Basic use of map

775 `map` works very similarly to `lapply`, where `.x` is object on whose elements to apply the
 776 function `.f`.

```

# get the number of rows in data
map(.x = nested_data$data, .f = nrow) %>%
  head()
#> [[1]]
#> [1] 2306
#>
#> [[2]]
#> [1] 1162
#>
#> [[3]]
#> [1] 313

```



```

#>
#> [[4]]
#> [1] 34
#>
#> [[5]]
#> [1] 183
#>
#> [[6]]
#> [1] 392

```

777 map works on any list-like object, which includes vectors, and always returns a list. map
 778 takes two arguments, the object on which to operate, and the function to apply to each
 779 element.

```

# get the square root of each integer 1 - 10
some_numbers = 1:3
map(some_numbers, sqrt)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 1.41
#>
#> [[3]]
#> [1] 1.73

```

780 4.2.2 map variants returning vectors

781 Though map always returns a list, it has variants named map_* where the suffix indicates
 782 the return type. map_chr, map_dbl, map_int, and map_lgl return character, double (nu-
 783 meric), integer, and logical vectors.

```

# use map_dbl to get the mean mass in each order
map_dbl(nested_data$data, function(df){
  mean(df$mass_g)
})
#> [1] 4.86e+02 4.91e+01 4.79e+04 7.86e+05 4.02e+04 1.85e+06 6.68e+03 3.06e+02
#> [9] 1.61e+02 4.06e+01 7.48e+02 1.45e+03 2.36e+05 3.37e+01 1.74e+02 9.58e+05
#> [17] 9.03e+02 4.70e+06 1.13e+03 2.84e+03 2.23e+01 1.12e+06 1.83e+02 5.94e+05
#> [25] 1.22e+04 9.44e+03 1.65e+06 4.45e+01 5.24e+04

# map_chr will convert the output to a character
# here we get the most common IUCN status of each order
map_chr(nested_data$data, function(df){

  count(df, iucn_status) %>%
    arrange(-n) %>%

```

```

    summarise(common_status = first(iucn_status)) %>%
    pull(common_status)
  })
#> [1] "LC" "LC" "LC" "EP" "LC" "LC" "LC" "LC" "LC" "LC" "LC" "LC" "EP" "VU" "LC"
#> [16] "EP" "LC" "EP" "LC" "LC" "NT" "VU" "LC" "EP" "VU" "CR" "EP" "LC" "LC"

# map_lgl returns TRUE/FALSE values
some_numbers = c(NA, 1:3, NA, NaN, Inf, -Inf)
map_lgl(some_numbers, is.na)
#> [1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE

```

784 4.2.3 map variants returning data frames

785 map_df returns data frames, and by default binds dataframes by rows, while map_dfr
 786 does this explicitly, and map_dfc does returns a dataframe bound by column.

```

# get the first two rows of each dataframe
map_df(nested_data$data[1:3], head, n = 2)
#> # A tibble: 6 x 23
#>   binomial family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr>          <dbl>  <dbl>        <dbl>  <dbl>
#> 1 Abditom~ Murid~ Abdi~ latide~          1    0            0    0
#> 2 Abeomel~ Murid~ Abeo~ sevia          1    0            0    0
#> 3 Acerodo~ Ptero~ Acer~ celebe~          0    0            0    1
#> 4 Acerodo~ Ptero~ Acer~ humilis          0    0            0    1
#> 5 Acinony~ Felid~ Acin~ jubatus          1    0            0    0
#> 6 Ailurop~ Ursid~ Ailu~ melano~          1    0            0    0
#> # ... with 15 more variables: life_habit_method <chr>, life_habit_source <chr>,
#> #   mass_g <dbl>, mass_method <chr>, mass_source <chr>, mass_comparison <chr>,
#> #   mass_comparison_source <chr>, island_endemicity <chr>, iucn_status <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertibrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>

```

787 map accepts arguments to the function being mapped, such as in the example above,
 788 where head() accepts the argument n = 2.

789 map_dfr behaves the same as map_df.

```

# the same as above but with a pipe
nested_data$data[1:5] %>%
  map_dfr(head, n = 2)
#> # A tibble: 10 x 23
#>   binomial family genus species terrestrial marine freshwater aerial
#>   <chr>      <chr> <chr> <chr>          <dbl>  <dbl>        <dbl>  <dbl>
#> 1 Abditom~ Murid~ Abdi~ latide~          1    0            0    0
#> 2 Abeomel~ Murid~ Abeo~ sevia          1    0            0    0
#> 3 Acerodo~ Ptero~ Acer~ celebe~          0    0            0    1
#> 4 Acerodo~ Ptero~ Acer~ humilis          0    0            0    1

```

```
#> 5 Acinony~ Felid~ Acin~ jubatus      1      0      0      0
#> 6 Ailurop~ Ursid~ Ailu~ melano~      1      0      0      0
#> # ... with 4 more rows, and 15 more variables: life_habit_method <chr>,
#> #   life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

map_dfc binds the resulting 3 data frames of two rows each by column, and automatically repairs the column names, adding a suffix to each duplicate.

4.2.4 Working with list columns using map

The various map versions integrate well with list columns to make synthetic/summary data. In the example, the `dplyr::mutate` function is used to add three columns to the nested tibble: the number of rows, the mean mileage, and the name of the first car.

In each of these cases, the vectors added are generated using `purrr` functions.

```
# get the number of rows per dataframe, the mean mileage, and the first car
nested_data = nested_data %>%
  mutate(
    # use the int return to get the number of rows
    n_rows = map_int(data, nrow),

    # double return for mean mileage
    mean_mass = map_dbl(data, function(df) {mean(df$mass_g)}),

    # character return to get the heaviest member
    first_animal = map_chr(data, function(df) {
      arrange(df, -mass_g) %>%
        . $binomial %>%
        first()
    })
  )

# examine the output
nested_data
#> # A tibble: 29 x 5
#> # Groups:   order [29]
#>   order      data      n_rows mean_mass first_animal
#>   <chr>    <list>    <int>    <dbl> <chr>
#> 1 Rodentia <tibble [2,306 x 23]> 2306    486. Nechoerus_aesopi
#> 2 Chiroptera <tibble [1,162 x 23]> 1162    49.1 Acerodon_jubatus
#> 3 Carnivora <tibble [313 x 23]> 313   47905. Mirounga_leonina
#> 4 Pilosa <tibble [34 x 23]> 34  785958. Megatherium_americanum
```

```
#> 5 Diprotodontia <tibble [183 x 23]>      183    40202. Diprotodon_optatum
#> 6 Cetartiodactyla <tibble [392 x 23]>      392  1854811. Balaenoptera_musculus
#> # ... with 23 more rows
```

4.2.5 Selective mapping using map variants

map_at and map_if work like other *_at and *_if functions. Here, map_if is used to run a linear model only on those tibbles which have sufficient data. The predicate is specified by .p.

In this example, the nested tibble is given a new column using dplyr::mutate, where the data to be added is a mixed list.

```
# split data by order number and run an lm only if there are more than 100 rows
nested_data = nest(data, data = -order)

nested_data = mutate(nested_data,
  model = map_if(.x = data,

    # this is the predicate
    # which elements should be operated on?
    .p = function(x){
      nrow(x) > 100
    },

    # this is the function to use
    # if the predicate is satisfied
    .f = function(x){
      lm(mass_g ~ diet_plant, data = x)
    })

# check the data structure
nested_data %>% head()
#> # A tibble: 6 x 3
#>   order      data      model
#>   <chr>    <list>    <list>
#> 1 Rodentia <tibble [2,306 x 23]> <lm>
#> 2 Chiroptera <tibble [1,162 x 23]> <lm>
#> 3 Carnivora <tibble [313 x 23]> <lm>
#> 4 Pilosa <tibble [34 x 23]> <tibble [34 x 23]>
#> 5 Diprotodontia <tibble [183 x 23]> <lm>
#> 6 Cetartiodactyla <tibble [392 x 23]> <lm>
```

Some elements of the column model are tibbles, which have not been operated on because they have fewer than 100 rows (species). The remaining elements are lm objects.

805 4.3 More map variants

806 map also has variants along the axis of how many elements are operated upon. map2 op-
 807 erates on two vectors or list-like elements, and returns a single list as output, while pmap
 808 operates on a list of list-like elements. The output has as many elements as the input lists,
 809 which must be of the same length.

810 4.3.1 Mapping over two inputs with map2

811 map2 has the same variants as map, allowing for different return types. Here map2_int
 812 returns an integer vector.

```
# consider 2 vectors and replicate the simple vector addition using map2
map2_int(.x = 1:5,
         .y = 6:10,
         .f = sum)
#> [1] 7 9 11 13 15
```

813 map2 doesn't have _at and _if variants.

814 One use case for map2 is to deal with both a list element and its index, as shown in the
 815 example. This may be necessary when the list index is removed in a split or nest. This
 816 can also be done with imap, where the index is referred to as .y.

```
# make a named list for this example
this_list = list(a = "first letter",
                b = "second letter")
```

```
# a not particularly useful example
map2(this_list, names(this_list),
     function(x, y) {
       glue::glue('{x} : {y}')
     })
#> $a
#> first letter : a
#>
#> $b
#> second letter : b
```

```
# imap can also do this
imap(this_list,
     function(x, .y){
       glue::glue('{x} : {y}')
     })
#> $a
#> first letter : a
#>
```

```
#> $b
#> second letter : b
```

817 4.3.2 Mapping over multiple inputs with pmap

818 pmap instead operates on a list of multiple list-like objects, and also comes with the same
819 return type variants as map. The example shows both aspects of pmap using pmap_chr.

```
# operate on three different lists
list_01 = as.list(1:3)
list_02 = as.list(letters[1:3])
list_03 = as.list(rainbow(3))

# print a few statements
pmap_chr(list(list_01, list_02, list_03),
         function(l1, l2, l3){
           glue::glue('number {l1}, letter {l2}, colour {l3}')
         })
#> [1] "number 1, letter a, colour #FF0000FF"
#> [2] "number 2, letter b, colour #00FF00FF"
#> [3] "number 3, letter c, colour #0000FFFF"
```

820 4.4 Combining map variants and tidyverse functions

821 The example below shows a relatively complex data manipulation pipeline. Such
822 pipelines must either be thought through carefully in advance, or checked for required
823 output on small subsets of data, so as not to consume excessive system resources.

824 In the pipeline:

- 825 1. The tibble becomes a nested dataframe by order (using `tidyr::nest`),
- 826 2. If there are enough data points (> 100), a linear model of mass ~ plant diet is fit
827 (using `purrr::map_if`, and `stats::lm`),
- 828 3. The model coefficients are extracted if the model was fit (using `purrr::map` &
829 `dplyr::case_when`),
- 830 4. The model coefficients are converted to data for plotting (using `purrr::map`, `tibble::tibble`, & `tidyr::pivot_wider`),
- 831 5. The raw data is plotted along with the model fit, taking the title from the nested data
832 frame (using `purrr::pmap` & `ggplot2::ggplot`).

```
nested_data <-
  data %>%
  tidyr::nest(data = -order) %>%
  mutate(data,
         model = map_if(.x = data,
                        # this is the predicate
```

```

# which elements should be operated on?
.p = function(x){
  nrow(x) > 100
},

# this is the function to use
# if the predicate is satisfied
.f = function(x){
  lm(mass_g ~ diet_plant, data = x)
}) %>%

mutate(m_coef = map(model,

# use case when to get model coefficients
function(x) {
  dplyr::case_when(
    "lm" %in% class(x) ~ {
      list(coef(x))
    },
    TRUE ~ {
      list(c(NA,NA))
    }
  )},

# work on the two element double vector of coefficients
m_coef = map(m_coef, function(x){
  tibble(coef = unlist(x),
    param = c("intercept", "diet_plant")) %>%
    pivot_wider(names_from = "param",
      values_from = "coef")
}),

# work on the raw data and the coefficients
plot = pmap(list(order, data, m_coef), function(ord, x, y){

# pay no attention to the ggplot for now
ggplot2::ggplot()+
  geom_point(data = x,
    aes(diet_plant, mass_g),
    size = 0.1)+
  scale_y_log10()+
  labs(title = glue::glue('order: {ord}'))
})
)

```

834 4.5 A return to map variants

835 Lists are often nested, that is, a list element may itself be a list. It is possible to map a
836 function over elements as a specific depth.

837 In the example, phylacine is split by order, and then by IUCN status, creating a two-level
838 list, with the second layer operated on.

```
# use map to make a 2 level list
this_list = split(data, data$order) %>%
  map(function(df){ split(df, df$iucn_status) })

# map over the second level to count the number of
# species in each order in each IUCN class
# display only the first element
map_depth(this_list[1], 2, nrow)
#> $Afrosoricida
#> $Afrosoricida$CR
#> [1] 1
#>
#> $Afrosoricida$DD
#> [1] 4
#>
#> $Afrosoricida$EN
#> [1] 7
#>
#> $Afrosoricida$EP
#> [1] 2
#>
#> $Afrosoricida$LC
#> [1] 32
#>
#> $Afrosoricida$NT
#> [1] 3
#>
#> $Afrosoricida$VU
#> [1] 8
```

839 4.5.1 Iteration without a return

840 `map` and its variants have a return type, which is either a list or a vector. However, it is
841 often necessary to iterate a function over a list-like object for that function's side effects,
842 such as printing a message to screen, plotting a series of figures, or saving to file.

843 `walk` is the function for this task. It has only the variants `walk2`, `iwalk`, and `pwalk`, whose
844 logic is similar to `map2`, `imap`, and `pmap`. In the example, the function applied to each list
845 element is intended to print a message.


```

this_list = split(data, data$order)

iwalk(this_list,
      function(df, .y){
        print(glue::glue('{nrow(df)} species in order {.y}'))
      })
#> 57 species in order Afrosoricida
#> 313 species in order Carnivora
#> 392 species in order Cetartiodactyla
#> 1162 species in order Chiroptera
#> 39 species in order Cingulata
#> 74 species in order Dasyuromorphia
#> 2 species in order Dermoptera
#> 97 species in order Didelphimorphia
#> 183 species in order Diprotodontia
#> 465 species in order Eulipotyphla
#> 5 species in order Hyracoidea
#> 94 species in order Lagomorpha
#> 3 species in order Litopterna
#> 19 species in order Macroscelidea
#> 1 species in order Microbiotheria
#> 7 species in order Monotremata
#> 2 species in order Notoryctemorphia
#> 3 species in order Notoungulata
#> 7 species in order Paucituberculata
#> 24 species in order Peramelemorphia
#> 29 species in order Perissodactyla
#> 9 species in order Pholidota
#> 34 species in order Pilosa
#> 460 species in order Primates
#> 18 species in order Proboscidea
#> 2306 species in order Rodentia
#> 20 species in order Scandentia
#> 5 species in order Sirenia
#> 1 species in order Tubulidentata

```

846 4.5.2 Modify rather than map

847 When the return type is expected to be the same as the input type, that is, a list returning
 848 a list, or a character vector returning the same, `modify` can help with keeping strictly to
 849 those expectations.

850 In the example, simply adding 2 to each vector element produces an error, because the
 851 output is a numeric, or double. `modify` helps ensure some type safety in this way.

```
vec = as.integer(1:10)
```

```

tryCatch(
  expr = {

    # this is what we want you to look at

    modify(vec, function(x) { (x + 2) })

  },

  # do not pay attention to this
  error = function(e){
    print(toString(e))
  }
)
#> [1] "Error: Can't coerce element 1 from a double to a integer\n"

```

852 Converting the output to an integer, which was the original input type, serves as a solution.

```

modify(vec, function(x) { as.integer(x + 2) })
#> [1] 3 4 5 6 7 8 9 10 11 12

```

853 A note on invoke

854 `invoke` used to be a wrapper around `do.call`, and can still be found with its family of
 855 functions in `purrr`. It is however retired in favour of functionality already present in `map`
 856 and `rlang::exec`, the latter of which will be covered in another session.

857 4.6 Other functions for working with lists

858 `purrr` has a number of functions to work with lists, especially lists that are not nested
 859 list-columns in a tibble.

860 4.6.1 Filtering lists

861 Lists can be filtered on any predicate using `keep`, while the special case `compact` is ap-
 862 plied when the empty elements of a list are to be filtered out. `discard` is the opposite of
 863 `keep`, and keeps only elements not satisfying a condition. Again, the predicate is specified
 864 by `.p`.

```

# a list containing numbers
this_list = list(a = 1, b = -1, c = 2, d = NULL, e = NA)

# remove the empty element
# this must be done before using keep on the list
this_list = compact(this_list)

# use discard to remove the NA
this_list = discard(this_list, .p = is.na)

```

```

# keep list elements which are positive
keep(this_list, .p = function(x){ x > 0 })
#> $a
#> [1] 1
#>
#> $c
#> [1] 2

```

865 head_while is bit of an odd case, which returns all elements of a list-like object in se-
 866 quence until the first one fails to satisfy a predicate, specified by .p.

```

1:10 %>%
  head_while(.p = function(x) x < 5)
#> [1] 1 2 3 4

```

867 4.6.2 Summarising lists

868 The purrr functions every, some, has_element, detect, detect_index, and
 869 vec_depth help determine whether a list passes a certain logical test or not. These are
 870 seldom used and are not discussed here.

871 4.6.3 Reduction and accumulation

872 reduce helps combine elements along a list using a specific function. Consider the ex-
 873 ample below where list elements are concatenated into a single vector.

```

this_list = list(a = 1:3, b = 3:4, c = 5:10)

reduce(this_list, c)
#> [1] 1 2 3 3 4 5 6 7 8 9 10

```

874 This can also be applied to data frames. Consider some random samples of mtcars, each
 875 with only 5 cars removed. The objective is to find the cars present in all 10 samples.

876 The way reduce works in the example below is to take the first element and find its inter-
 877 section with the second, and to take the result and find its intersection with the third and
 878 so on.

```

# sample mtcars
mtcars = as_tibble(mtcars, rownames = "car")

sampled_data = map(1:10, function(x){
  sample_n(mtcars, nrow(mtcars)-5)
})

# get cars which appear in all samples
sampled_data = reduce(sampled_data,
  dplyr::inner_join)

```

879 `accumulate` works very similarly, except it retains the intermediate products. The first
 880 element is retained as is. `accumulate2` and `reduce2` work on two lists, following the
 881 same logic as `map2` etc. Both functions can be used in much more complex ways than
 882 demonstrated here.

```
# make a list
this_list = list(a = 1:3, b = 3:6, c = 5:10, d = c(1,2,5,10,12))

# a multiple accumulate can help
accumulate(this_list, union, .dir = "forward")
#> $a
#> [1] 1 2 3
#>
#> $b
#> [1] 1 2 3 4 5 6
#>
#> $c
#> [1] 1 2 3 4 5 6 7 8 9 10
#>
#> $d
#> [1] 1 2 3 4 5 6 7 8 9 10 12
```

883 4.6.4 Miscellaneous operation

884 `purrr` offers a few more functions to work with lists (or list like objects). `prepend` works
 885 very similarly to `append`, except it adds to the head of a list. `splice` adds multiple objects
 886 together in a list. `splice` will break the existing list structure of input lists.

887 `flatten` has a similar behaviour, and converts a list of vectors or list of lists to a single
 888 list-like object. `flatten_*` options allow the output type to be specified.

```
this_list = list(a = rep("a", 3),
                 b = rep("b", 4))

this_list
#> $a
#> [1] "a" "a" "a"
#>
#> $b
#> [1] "b" "b" "b" "b"

# use flatten_chr to get a character vector
flatten_chr(this_list)
#> [1] "a" "a" "a" "b" "b" "b" "b"
```

889 `transpose` shifts the index order in multi-level lists. This is seen in the example, where
 890 the `iucn_status` goes from being the index of the second level to the index of the first.

```

this_list = split(data, data$order) %>%
  map(function(df) {split(df, df$iucn_status)})

# from a list of lists where species are divided by order and then
# iucn_status, this is now a list of lists where species are
# divided by status and then order
transpose(this_list[1])

```

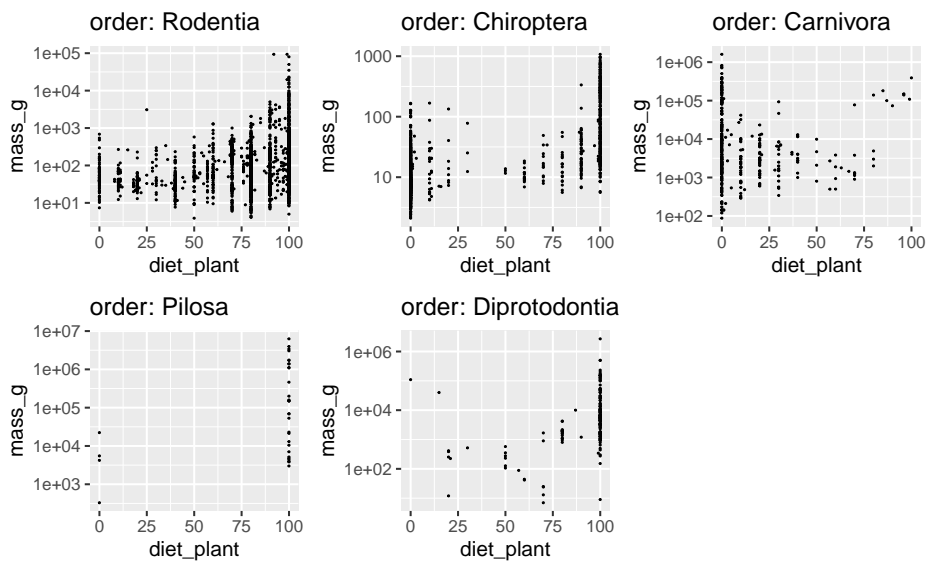
891 4.7 Lists of ggplots with patchwork

892 The patchwork library helps compose ggplots, which will be properly introduced in the
 893 next session. patchwork usually works on lists of ggplots, which can come from a stan-
 894 dalone list, or from a list column in a nested dataframe. The example below shows the
 895 latter, with the data data frame from earlier.

```

# use patchwork on list
patchwork::wrap_plots(nested_data$plot[1:5])

```



896

Chapter 5

ggplot2 and the grammar of graphics

By Raphael Scherrer, data from Anne-Marie Veenstra-Skirl

Every use case is ridiculous
until it happens to you.

In this tutorial we will learn how to make nice graphics using `ggplot2`, perhaps the most well-known member of the tidyverse. So well-known, in fact, that people often know `ggplot2` before they get to know about the tidyverse. We will first learn about the philosophy behind `ggplot2` and then follow that recipe to build more complex customized plots through some examples.

5.1 Introduction

5.1.1 What is ggplot2 and why use it?

There are many ways of making graphics in base R. For example, `plot` is used for scatterplots, `hist` is used for histograms, `boxplot` is self-explanatory, and `image` can be used for heatmaps. However, those functions are often developed by different people with different logics in mind, which can make them inconsistent with each other, e.g. one has to learn what the arguments of each function are and switching from one type of visualization to another may not be very easy. `ggplot2` is aimed at solving this problem and making plotting *flexible*, allowing to build virtually any graph using a common standard, the *grammar of graphics* (which is what “gg” stands for). By building on a single reference grammar, `ggplot2` fits nicely into the tidyverse, and as part of it, it also follows the same rule as `tidyr`, `dplyr` or `purrr`, making the integration between all those packages very smooth.

5.1.2 What is the grammar of graphics?

The grammar of graphics is a system of rules on how to structure plots such that almost any graph can be made through combinations of a limited set of simpler elements, just as you can make any sentence by combining together letters from an alphabet. `ggplot2` is the implementation of this philosophy in R, and comes with a limited set of *layers*, that you can pick and combine into an impressive variety of graphics, all based on the same syntax. But what are those elements?

Here is the backbone of a `ggplot` statement (I will from now on use “`ggplot`” to refer to an object of class `ggplot`, the output of the `ggplot` function and the object that contains our graphic), taken from the book R for Data Science:

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

This pseudocode snippet illustrates a fundamental aspect of `ggplot2`, which is that plots are built by *successive* commands, each corresponding to a layer, assembled together using the `+` operator. This might seem less practical than having a whole plot made in a single call to the `plot` function, but it is this modularity that actually gives `ggplot2` its flexibility.

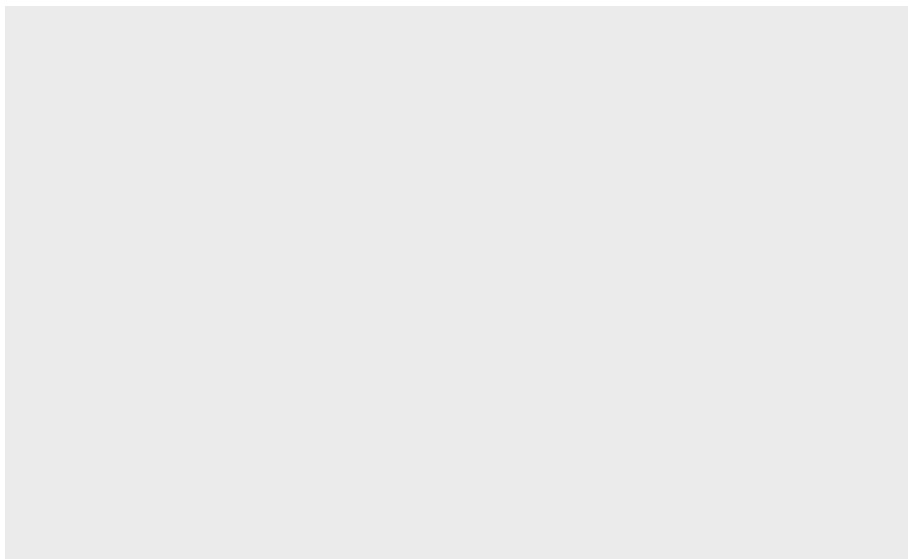
This means that in `ggplot2` you will typically need multiple commands to make a plot. All `ggplots` are made of at least the two following basic ingredients:

- A call to the `ggplot` function, with the relevant data frame passed to it (this data frame contains our data to plot)

- A geom layer, specifying the type of plot to be shown. Variables from the data are mapped onto the graphical properties of this layer, called *aesthetics*.

That means that:

```
library(tidyverse)
ggplot(mtcars)
```



will not show anything. A `ggplot` object is there, but it has no layers yet.

Plots can then be customized with statistical transformations, re-positioning, changes in coordinate system, facetting, and more. We will now go through the different elements.

5.1.3 Quick plot

Note that the `qplot` function, which stands for “quick plot”, will show a plot when called on your dataset. It is a wrapper around `ggplot2` layers that allows to quickly get a visualization, just like using `plot` from base R. However, it is less flexible than combining your `ggplot` yourself, so here we will make sure that you understand how the different layers are assembled.

5.2 But first, the data

In this chapter we will use the data from `bacterial_experiment.csv`, forged by Annie for us to use. This dataset resembles Annie’s experiment where she created mutator strains of bacteria (that is, bacteria that mutate at a much higher rate than usual) and tracked their growth through time and at different concentrations of an agent supposed to activate the full “mutation potential” of those strains.

```
data <- read_csv("data/bacterial_experiment.csv")
data
#> # A tibble: 310 x 7
#>   strain assay conc ratio time      cfu OD600
#>   <chr>   <chr> <dbl> <dbl> <chr>    <dbl> <dbl>
#> 1 strain 1 test 1      1  8.58 T0      3200000000 0.319
#> 2 strain 1 test 1      1  8.58 T1      1293846908 0.911
#> 3 strain 1 test 1      1  6.11 T0      370110830 0.287
#> 4 strain 1 test 1      1  6.11 T1      1480443320 0.9
#> 5 strain 1 test 1      1 11.8 T0      377928804 0.321
#> 6 strain 1 test 1      1 11.8 T1      1511715216 0.914
#> # ... with 304 more rows
```

965 The different strains of bacteria were grown in two different assays, whose details are ir-
 966 relevant for the purpose of this tutorial. `cfu` is the number of colony forming units while
 967 `OD600` is the optical density at 600nm wavelength; both are estimates of bacterial popula-
 968 tion density. `ratio` represents the ratio in mutants between two time points, T0 and T1
 969 (encoded in time).

970 In this table, the unit of observation is the time point (T0 and T1 are on different rows),
 971 therefore the values of `ratio`, which are attributed to each T0-T1 pair, are duplicated to
 972 yield one value per time point. To make our life easier with later plotting and to stay within
 973 the *tidy* spirit of the tidyverse (where one table should have one unit of observation), we
 974 use the tools we have already learnt to make a ratio-wise table:

```
data2 <- data %>%
  pivot_wider(names_from = "time", values_from = c("cfu", "OD600"))
data2
#> # A tibble: 155 x 8
#>   strain assay conc ratio   cfu_T0   cfu_T1 OD600_T0 OD600_T1
#>   <chr>   <chr> <dbl> <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
#> 1 strain 1 test 1      1  8.58 3200000000 1293846908    0.319    0.911
#> 2 strain 1 test 1      1  6.11 370110830 1480443320    0.287    0.9
#> 3 strain 1 test 1      1 11.8 377928804 1511715216    0.321    0.914
#> 4 strain 1 test 1      1  7.78 369871771 1479487084    0.299    0.92
#> 5 strain 1 test 1      5 10.5 3800000000 1505539596    0.295    0.922
#> 6 strain 1 test 1      5  8.29 322488344 1289953376    0.275    0.88
#> # ... with 149 more rows
```

975 5.3 Geom layers

976 The geom object is the core visual layer of a plot, and it defines the type of plot being made,
 977 e.g. `geom_point` will add points, `geom_line` will add lines, etc. There are tons of geoms to
 978 pick from, depending on the type of figure you want to make, and new geoms are regularly
 979 added in extensions to `ggplot2` (links at the end of this chapter).

980 All geoms have aesthetics, or graphical parameters, that may be specified. Those include

x and y coordinates, color, transparency, etc. Some aesthetics are mandatory for some geoms, e.g. `geom_point` needs x and y coordinates of the points to plot. Other aesthetics are optional, e.g. if `color` is unspecified, all the points will look black. Some geoms even have no mandatory aesthetics, such as `geom_abline`, which will plot a diagonal running through the origin and with slope one if its `intercept` and `slope` are unspecified.

Aesthetics are specified in two ways: (1) variables from the data can be mapped to them using the `aes` function, or (2) they can take fixed values.

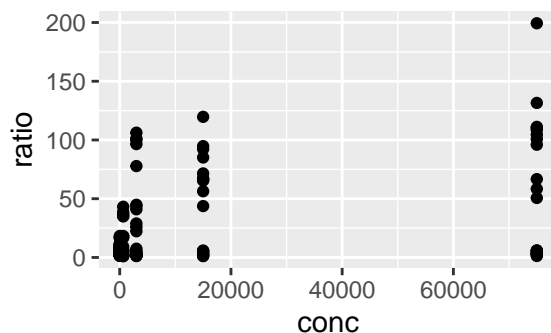
Some of the main aesthetics to know, besides geom-specific coordinates (e.g. x, y), include: `color`, `fill` (color used to fill surfaces), `group` (used e.g. to plot multiple lines with similar aspect on the same plot), `alpha` (transparency), `size`, `linetype`, `shape`, and `label` (for showing text).

Note that in most functions across the tidyverse both US and UK English can be used, e.g. `colour` is also a valid aesthetics, and `dplyr::summarize` is equivalent to `dplyr::summarise`.

5.3.1 Mapping variables to aesthetics

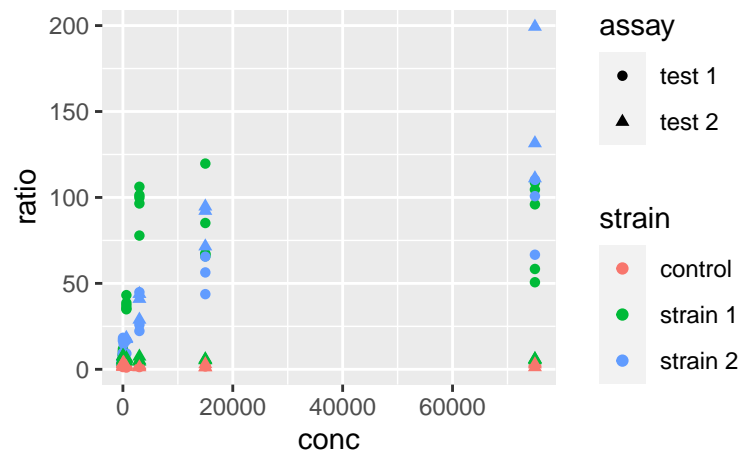
Variables are mapped to aesthetics using the `aes` function. Here is a basic scatterplot example showing `ratio` against `conc`:

```
ggplot(data2) +  
  geom_point(mapping = aes(x = conc, y = ratio))
```



We can use the other available aesthetics to show more aspects of the data, or to see patterns a bit more clearly. For example, we can color-code the points based on their strain, and change their shape based on the type of assay:

```
ggplot(data2) +  
  geom_point(mapping = aes(x = conc, y = ratio, color = strain, shape = assay))
```

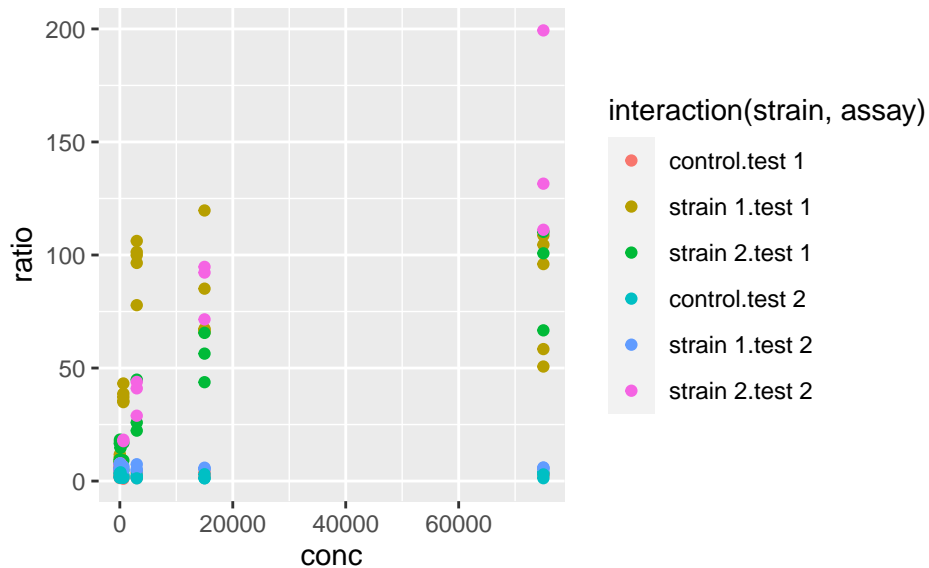


1002

1003 Do you want to map several variables to a single aesthetic? Then interaction from base
 1004 R can be used within a `ggplot`:

```
ggplot(data2) +
  geom_point(
    mapping = aes(x = conc, y = ratio, color = interaction(strain, assay))
  )
```

1005



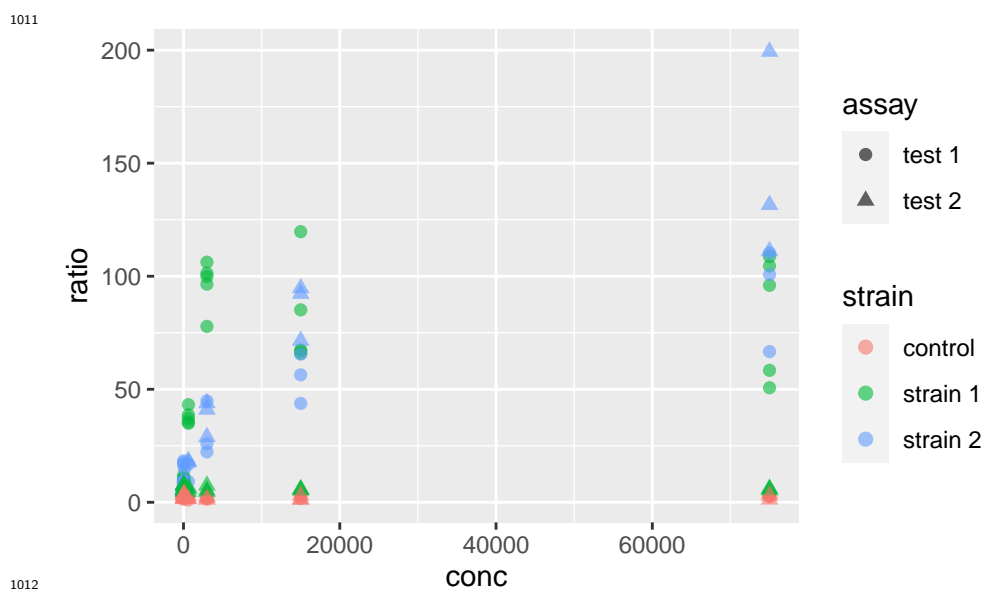
1006

1007 5.3.2 Fixed aesthetics

1008 Fixed graphical parameters (i.e. that are not mapped to a variable) should be added as
 1009 arguments of the geom *outside* the aes command. For example, to make *all* points a little

1010 bigger and more transparent, we can use

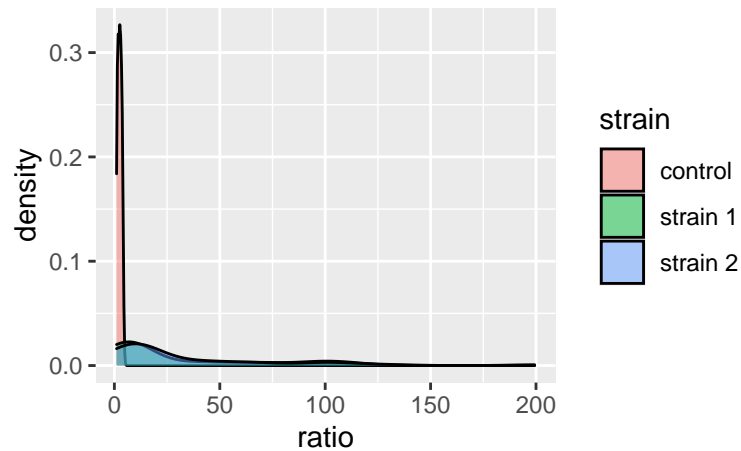
```
1011 ggplot(data2) +
  geom_point(
    mapping = aes(x = conc, y = ratio, color = strain, shape = assay),
    size = 2, alpha = 0.6
  )
```



1013 5.3.3 Statistical transformation

1014 Statistical transformations, or `stat` functions, can be applied to the data within a `geom`
 1015 call. Actually, statistical transformations are *always* applied within a `geom` call, but most
 1016 of the time the identity function is used. To illustrate, consider the following plot showing
 1017 a distribution of `ratio` for different strains:

```
ggplot(data2) +
  geom_density(aes(x = ratio, fill = strain), alpha = 0.5)
```

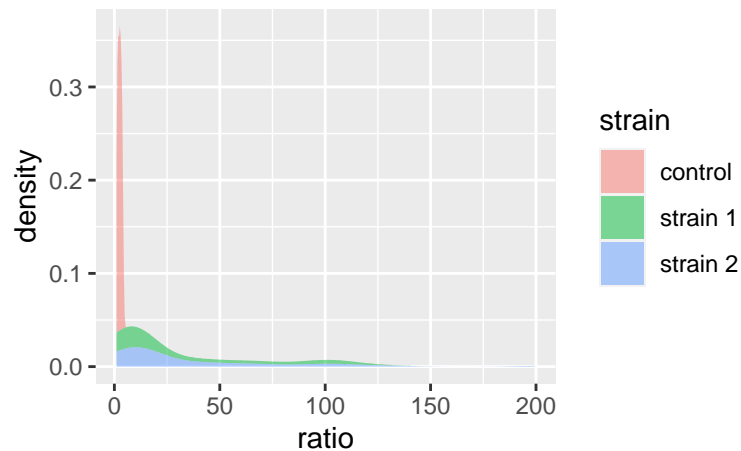


1018

1019 Here, the density axis is not part of the original dataset data2; it was computed from
 1020 the data, for each value of ratio, by using a density-estimation algorithm. This shows
 1021 that `stat_density` (and not `stat_identity`) is the default stat used in `geom_density`.
 1022 Every geom comes with its default stat.

1023 Similarly, `stat` functions can be used in place of `geom` because every `stat` has a default
 1024 `geom` associated to it. So, we can call:

```
ggplot(data2) +  
  stat_density(aes(x = ratio, fill = strain), alpha = 0.5)
```

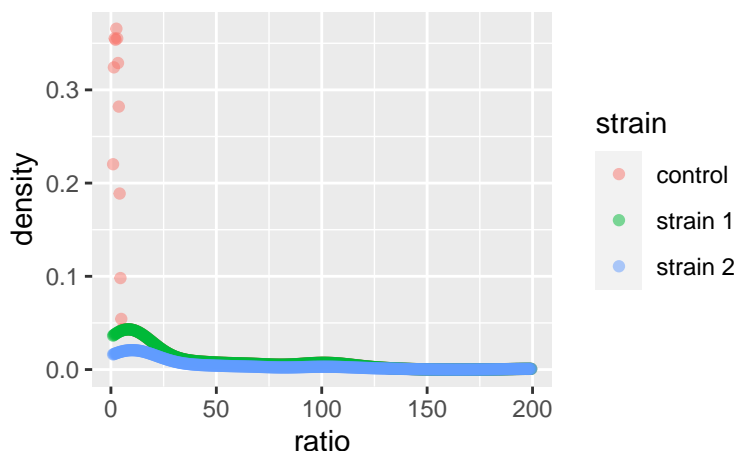


1025

1026 which has `geom_density` as default geom.

1027 It is possible to override the default stat using the `stat` argument of `geom`, and con-
 1028 versely, it is possible to change the default geom associated with a given stat. For ex-
 1029 ample, say we want to plot our densities as points. Then,

```
ggplot(data2) +  
  stat_density(aes(x = ratio, color = strain), alpha = 0.5, geom = "point")
```



1030

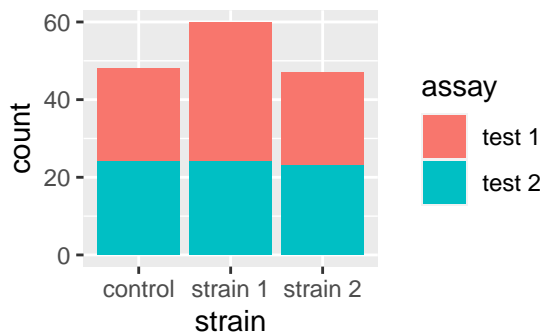
1031 does the job (note that we replaced `fill` with `color` because our points do not have a
1032 surface to fill).

1033 Note that default `geom-stat` combinations are usually well thought of (density plots are
1034 a good example). Therefore, it is often not necessary to play with stats. It may matter in
1035 some specific cases, e.g. when using `geom_bar`, but we do not cover that here (you can
1036 check out the dedicated chapter in R for Data Science for an example).

1037 5.3.4 Position

1038 The `position` argument of geoms allows to adjust the positioning of the geom's elements.
1039 It has a few variants, but the possibilities depend on the geom used. We illustrate those
1040 available to `geom_bar`. By default, `geom_bar` uses the `stat_count` statistical transfor-
1041 mation, meaning that it will show us the number of observations into each category of a
1042 factor, e.g. `strain`, splitted into categories of another factor, e.g. `assay`:

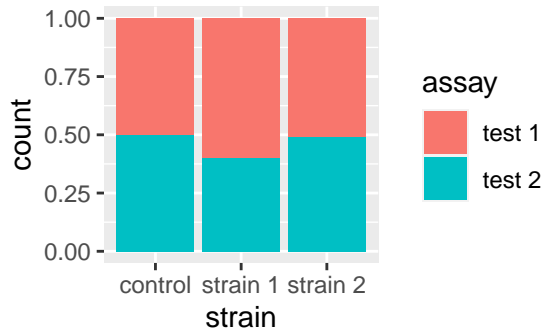
```
ggplot(data2) +  
  geom_bar(aes(x = strain, fill = assay))
```



1043

1044 If we wanted to visualize proportions instead of numbers, we could use the `fill` value of
1045 the `position` argument:

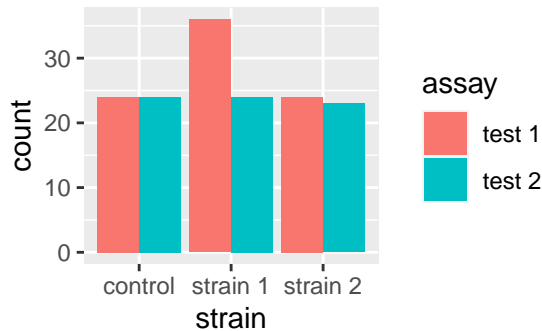
```
ggplot(data2) +
  geom_bar(aes(x = strain, fill = assay), position = "fill")
```



1046

1047 Alternatively we could use the `dodge` option to show the different categories side-by-side:

```
ggplot(data2) +
  geom_bar(aes(x = strain, fill = assay), position = "dodge")
```



1048

1049 Those are only two examples of what can be done. Just remember that `position` exists
 1050 and look into the documentation of your geom of interest to see what position adjustments
 1051 are available! (Check out `geom_jitter` as a nice wrapper around `geom_point` with a
 1052 `jitter` position adjustment, perfect to overlay with boxplots or violin plots.)

1053 5.3.5 Other geoms

1054 The most common geoms you may encounter are:

- 1055 • `geom_point` for scatter plots and `geom_jitter` for the dodged equivalent
- 1056 • `geom_bar` for a barplot
- 1057 • `geom_text` for a scatter plot of labels
- 1058 • `geom_histogram` and `geom_density`, self-explanatory
- 1059 • `geom_boxplot` and `geom_violin`
- 1060 • `geom_line`, `geom_path` (a line never goes backwards along the x-axis, while a
 1061 path can) and `geom_smooth` (local regression smoothing)
- 1062 • `geom_segment`, `geom_hline`, `geom_vline` and `geom_abline` that may come
 1063 handy as annotations

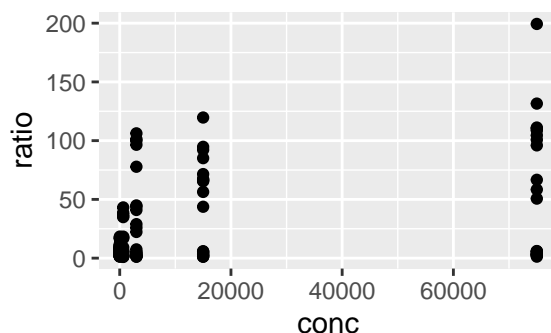
1064 • `geom_tile` for heatmaps

1065 There are literally tons of geoms and ways to use them. In this tutorial, we emphasize
 1066 the understanding of the grammar and how to assemble the different ingredients, rather
 1067 than the ingredients themselves. For this reason, here we are not giving an exhaustive
 1068 sample of each geom and what they look like. So, keep this list of names in mind as a
 1069 reminder that whatever plot you want to make, there probably is a geom for it. To explore
 1070 a gallery of examples, check out the R graph gallery.

1071 5.3.6 Extra on aesthetics

1072 It is possible to use the `+` operators, not only to add layers but also to modify previous lay-
 1073 ers. You might wonder why not to write the layer correctly in the first place. This starts
 1074 making more sense in cases e.g. where a plot can be modified in different ways. For ex-
 1075 ample, consider this plot:

```
ggplot(data2, aes(x = conc, y = ratio)) +  
  geom_point()
```



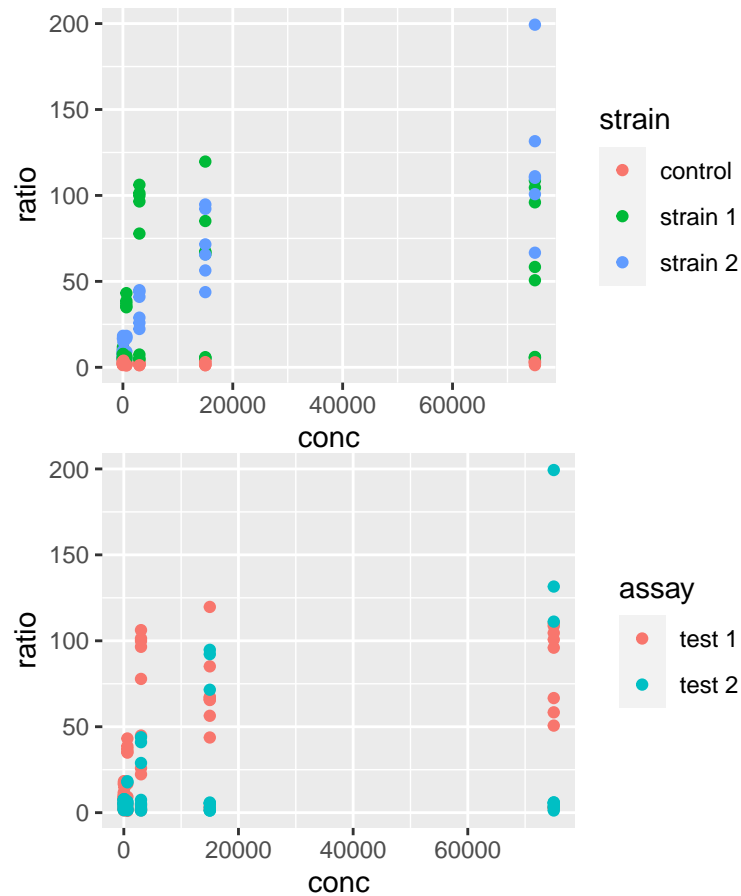
1076

1077 We may want to color-code the points based on `strain` or `assay`, or both, thus requiring
 1078 two plots building on this single one. An important property of `ggplot` objects is that they
 1079 can be assigned to variables, e.g.

```
p <- ggplot(data2, aes(x = conc, y = ratio)) +  
  geom_point()
```

1080 Note that we have to call the object `p` for the plot to be displayed. If we just assign the plot
 1081 to `p`, the plot does not show. We can subsequently add differential aesthetics to different
 1082 copies of `p`:

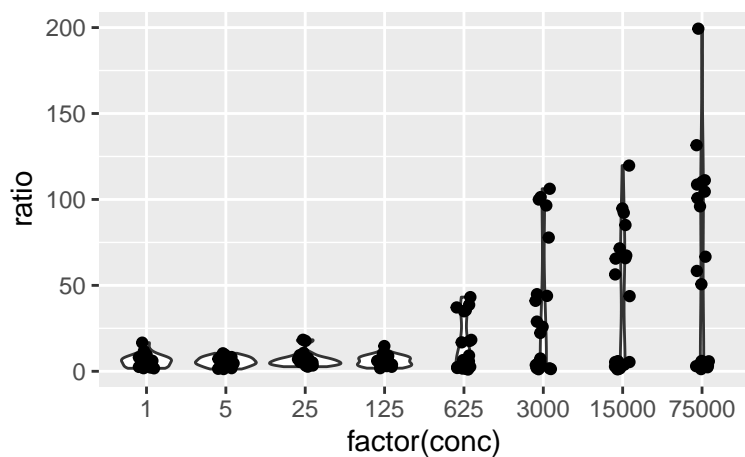
```
p + aes(color = strain)  
p + aes(color = assay)
```



5.3.7 Plot-wide aesthetics and multiple geoms

In the last example, by adding new aesthetics mapping to the `ggplot` using the `+` operator, we did not add these aesthetics *specifically* to the `geom_point` layer, but to all the geoms present in the plot. Similarly, one can pass aesthetic mappings to the `ggplot` command directly, not necessarily with the `geom` statement. This saves some typing when geoms taking the same aesthetics are used, e.g. `geom_violin` and `geom_jitter`:

```
ggplot(data2, aes(x = factor(conc), y = ratio)) +
  geom_violin() +
  geom_jitter(width = 0.1)
# x is made categorical here
```

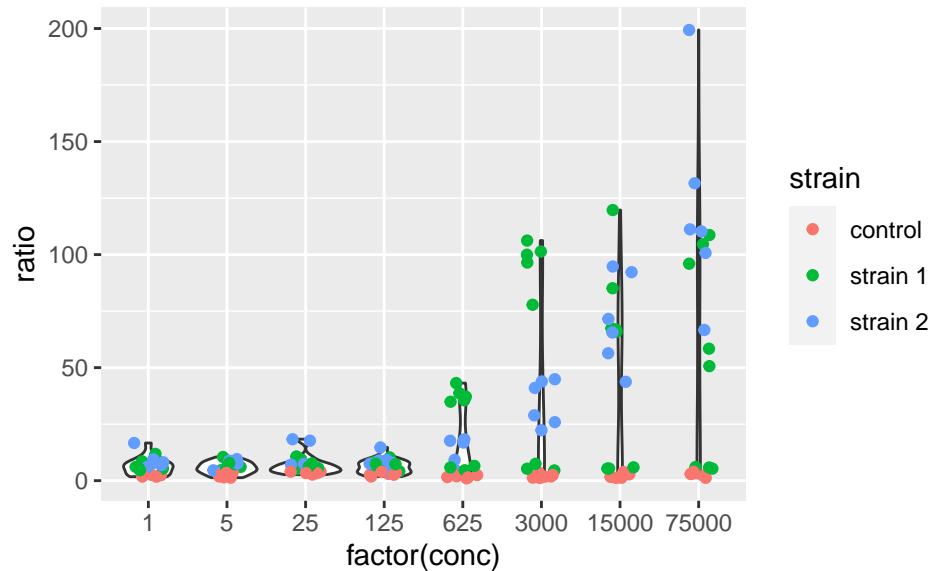


1091

1092 This shows a nice example of multiple geoms combined in a single plot. If, however, the
 1093 aesthetics used in some geoms are geom-specific, better pass them to their respective
 1094 geom. For example, if you want to color only the points but not the violins, use:

```
ggplot(data2, aes(x = factor(conc), y = ratio)) +  
  geom_violin() +  
  geom_jitter(mapping = aes(color = strain), width = 0.2)
```

1095



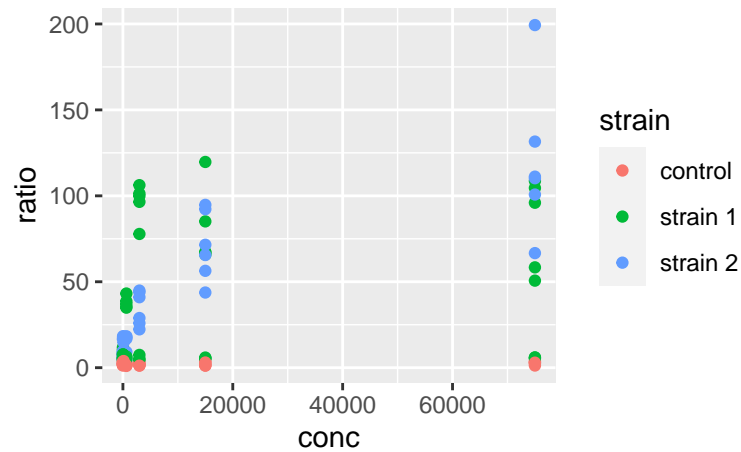
1096

1097 5.3.8 Multiple geoms with different datasets

1098 Just as aesthetics can vary from geom to geom, so do datasets. In other words, the dataset
 1099 does not have to be passed to the ggplot command necessarily, and can be passed to a

1100 geom instead, for example:

```
ggplot() +
  geom_point(data2, mapping = aes(x = conc, y = ratio, color = strain))
```



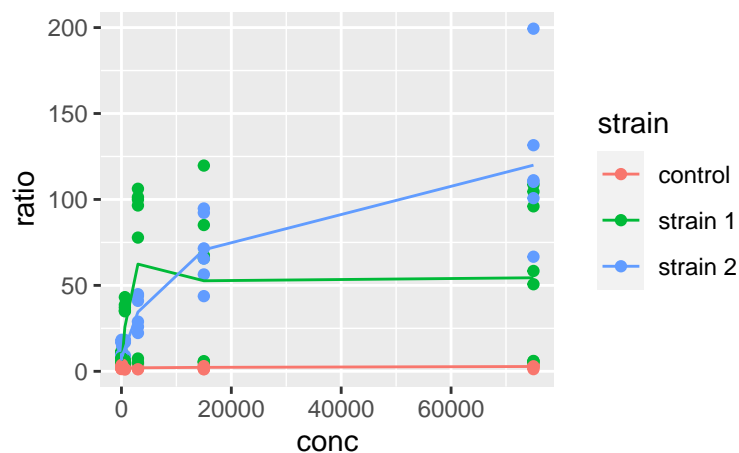
1101

1102 This means that different geoms can be based on different datasets. This allows quite
 1103 some complexification of the plots and illustrates very well the usefulness of the other
 1104 packages of the tidyverse. Say, for example, that we want to add to this plot a line going
 1105 through the means at each value of conc. These mean values are not yet present in our
 1106 dataset, and we need to come up with a mean-wise dataset. `dplyr` is our friend for this
 1107 task:

```
data3 <- data2 %>%
  group_by(conc, strain) %>%
  summarize(ratio = mean(ratio))
data3
#> # A tibble: 24 x 3
#> # Groups:   conc [8]
#>   conc strain  ratio
#>   <dbl> <chr>  <dbl>
#> 1     1 control   2.21
#> 2     1 strain 1   7.09
#> 3     1 strain 2   9.16
#> 4     5 control   2.50
#> 5     5 strain 1   7.17
#> 6     5 strain 2   6.89
#> # ... with 18 more rows
```

1108 Let us now add an extra layer of information based on this latest, summary dataset:

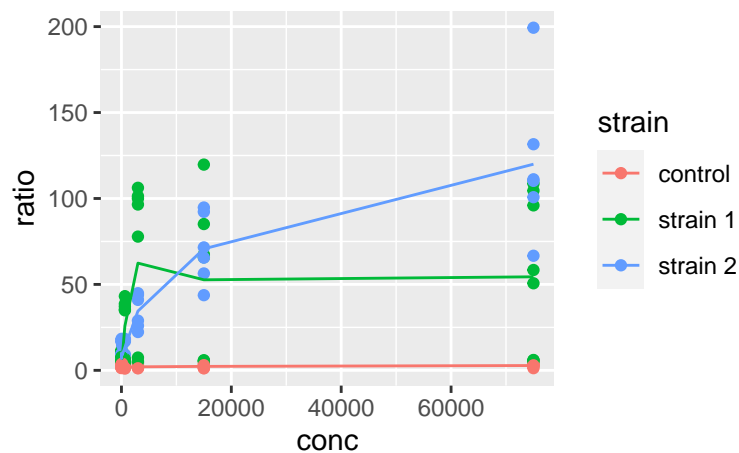
```
ggplot() +
  geom_point(data = data2, mapping = aes(x = conc, y = ratio, color = strain)) +
  geom_line(data = data3, mapping = aes(x = conc, y = ratio, color = strain))
```



1109

1110 Here, we could save some typing by writing:

```
ggplot(data2, mapping = aes(x = conc, y = ratio, color = strain)) +
  geom_point() +
  geom_line(data = data3)
```



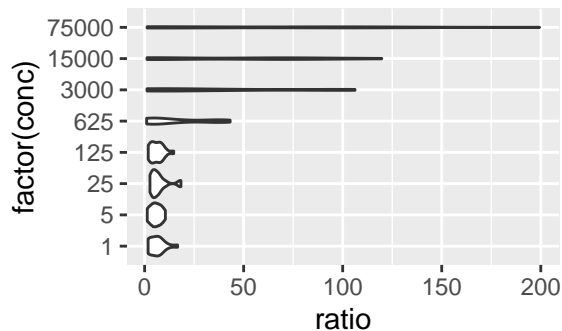
1111

1112 where `geom_line` inherits the same aesthetic mapping as `geom_point`. But then, you
 1113 have to make sure that `data3` contains all the aesthetics that the `ggplot` call expects to
 1114 see in each of its geoms (here `x`, `y` and `color`).

1115 5.4 Coordinate-system

1116 The default way that the plotting window is organized is an orthogonal space with a hor-
 1117 izontal x-axis and a vertical y-axis. Use the `coord` commands to deviate from this. For
 1118 example, `coord_flip` will flip the axes:

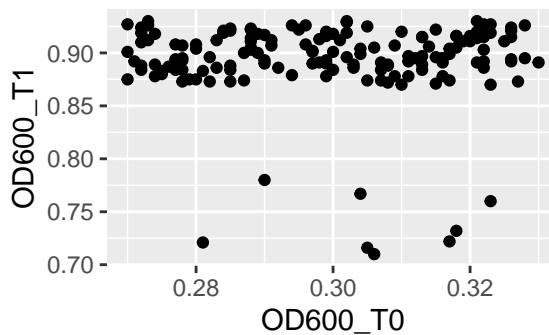
```
ggplot(data2, aes(x = factor(conc), y = ratio)) +
  geom_violin() +
  coord_flip()
```



1119

1120 while `coord_fixed` will fix the aspect ratio between the axes, thus showing them on the
 1121 same scale. For example, the following plot of the optical density between two time points,

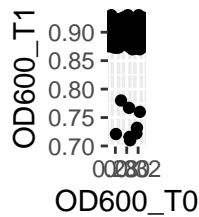
```
ggplot(data2, aes(x = OD600_T0, y = OD600_T1)) +
  geom_point()
```



1122

1123 becomes:

```
ggplot(data2, aes(x = OD600_T0, y = OD600_T1)) +
  geom_point() +
  coord_fixed()
```



1124

1125 when both axes are shown on the same scale.

1126 Other coordinate systems exist, depending on the need, including `coord_polar` for ra-

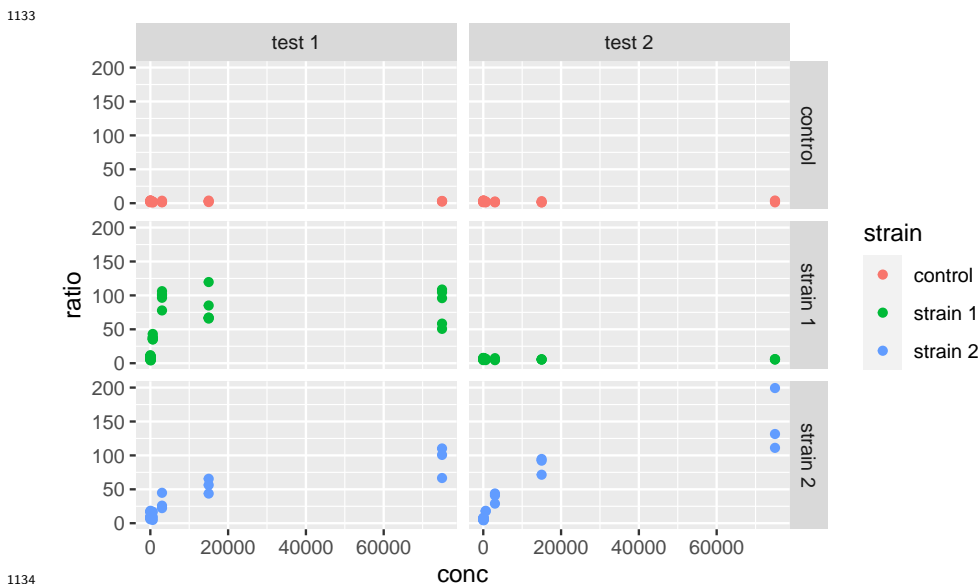
1127 dial plots or `coord_quickmap`, tailored at latitude-longitude plotting.

1128 5.5 Facetting

1129 One of the most powerful features of `ggplot2` is its easy way of splitting a plot into multi-
1130 ple subplots, or *facets*.

1131 There are two functions for facetting: `facet_grid` and `facet_wrap`. `facet_grid` will
1132 arrange the plot in rows and columns depending on variables that the user defines:

```
ggplot(data2, aes(x = conc, y = ratio, color = strain)) +  
  geom_point() +  
  facet_grid(strain ~ assay)
```

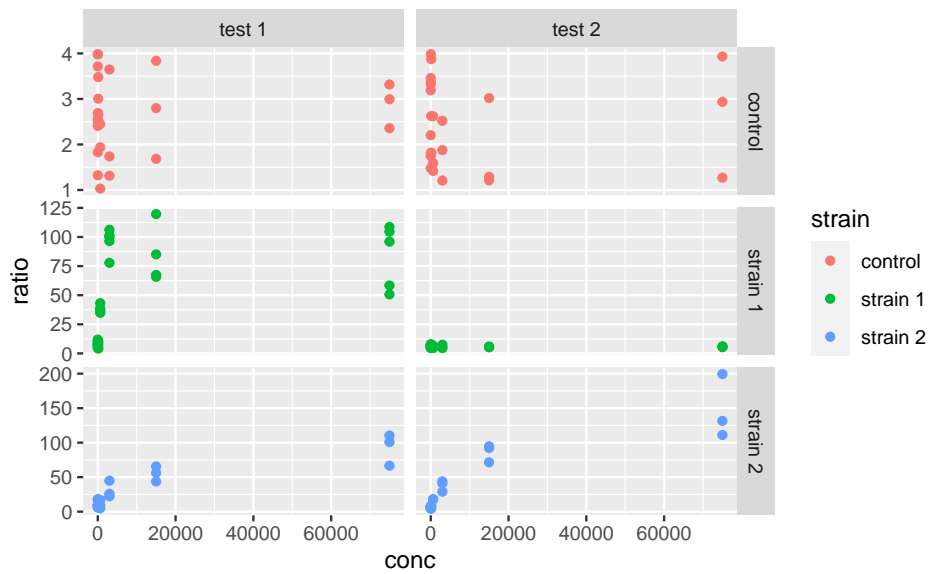


1135 Here the tilde (`~`) symbolizes a *formula*, a type of expression in R with a left and right-hand
1136 side, which here are interpreted as variables to use for rows and columns, respectively. If
1137 using only one variable for facetting, use `.` or nothing on the other side of the tilde.

1138 Note that facets are plotted on the same scale. We can use the `scales` argument to allow
1139 free scales, for example:

```
ggplot(data2, aes(x = conc, y = ratio, color = strain)) +  
  geom_point() +  
  facet_grid(strain ~ assay, scales = "free_y")
```

1140

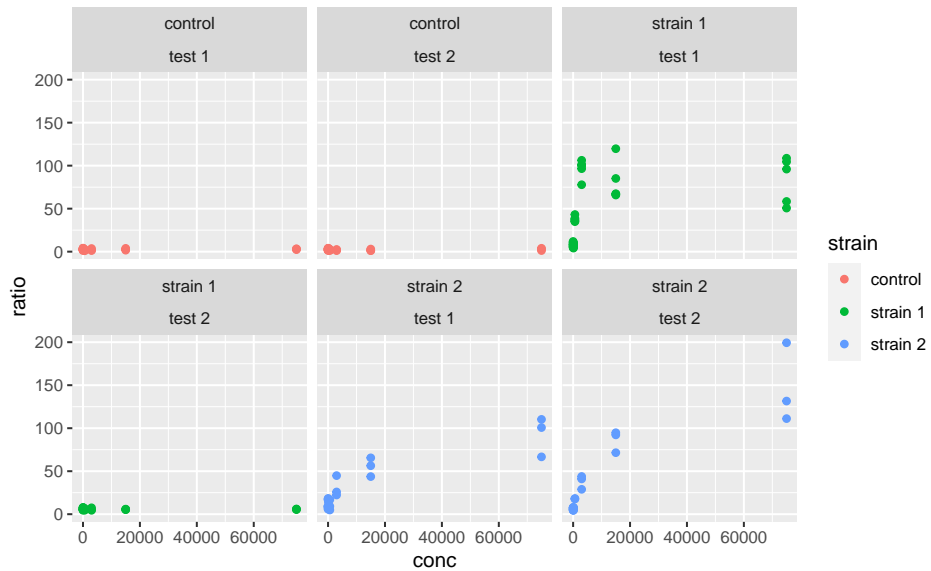


1141

1142 `facet_wrap` is similar to `facet_grid`, except that it does not organize the facets in rows
 1143 and columns but rather as an array of facets that fill the screen by row, like when filling a
 1144 matrix with numbers:

```
ggplot(data2, aes(x = conc, y = ratio, color = strain)) +  
  geom_point() +  
  facet_wrap(strain ~ assay)
```

1145



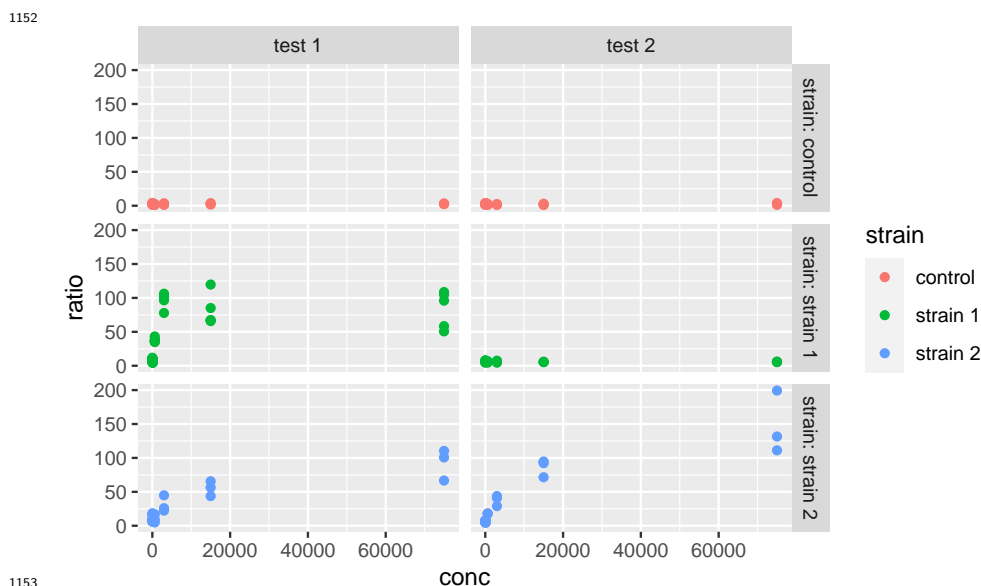
1146

1147 where the position of the variables relative to the `~` becomes irrelevant.

Note that a faceted ggplot is still *one* ggplot, not a combination of ggplots, which we will cover later.

Custom-labelling the strips of the facets is done with the `labeller` argument. The way this is used is a little complicated, but essentially looks like this:

```
ggplot(data2, aes(x = conc, y = ratio, color = strain)) +
  geom_point() +
  facet_grid(strain ~ assay, labeller = labeller(.rows = label_both))
```



Here, the `label_both` function is applied to the variable facetting by row, which is `strain`. `label_both` tells the `labeller` to label the strips with the name of the variable (`strain`) followed by its value, separated by a colon. We will not cover labelling in details here, but keep in mind that the `labeller` argument is what to play with, and that it takes the output of the `labeller` function as input, which itself takes labelling functions, such as `label_both`, as arguments. Other labelling functions include `label_value`, which just shows the value in the strip (that is the default) and `label_parsed`, which is used for showing mathematical expressions in strip labels (e.g. greek letters, exponents etc.). It is possible to provide custom names too. For more information on customizing facet strip labels, visit this link.

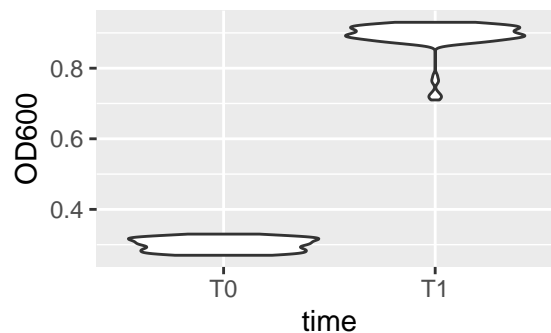
Note: I made a package called `ggsim`, yet another extension of `ggplot2` with a few functions coming handy for simulation data. One of the functions, `facetize`, is aimed at making your life easier when labelling the strips of your facets (i.e. not going into the nitty gritty of the `labeller` function), especially when some facets include parsing mathematical expressions. Feel free to install it from GitHub by using:

```
devtools::install_github("rscherrer/ggsim")
```

1169 5.6 The right format for the dataset

1170 One question that may come to your mind is: what is the right format of a dataset for use in
 1171 `ggplot`, especially since it is part of the `tidyverse`? The answer is: it depends, and this is
 1172 where the intergration with other `tidyverse` tools makes our life easier. If, for example, we
 1173 want to use a variable for facetting or as an aesthetics, it is important to have this variable
 1174 as a single column. For example, in the original data dataset, we could have compared
 1175 the optical density between the two time point:

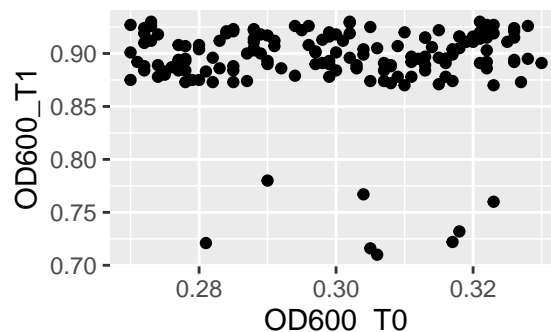
```
1176 ggplot(data, aes(x = time, y = OD600)) +  
  geom_violin()
```



1176

1177 where `time` is both an aesthetic (`x`) and its own column. However, if we want to plot the
 1178 optical density of time point T1 *versus* that of time point T0, then we need these two time
 1179 points in separate columns, which is exactly what `OD600_T0` and `OD600_T1`, in the `data2`
 1180 dataset, are (remember we got those using `tidyr::pivot_wider`):

```
ggplot(data2, aes(x = OD600_T0, y = OD600_T1)) +  
  geom_point()
```



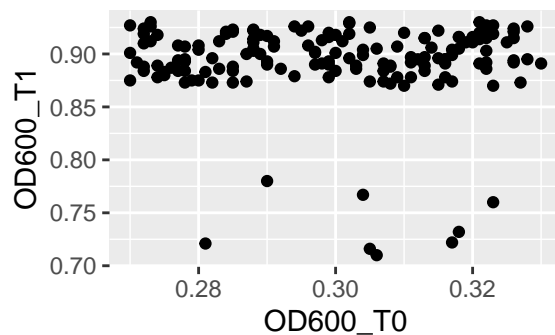
1181

1182 5.7 Plotting as part of a pipeline

1183 What we just saw means that sometimes reformatting of a dataset is needed (e.g. using
 1184 `pivot_longer` or `pivot_wider` from `tidyr`) to get this one plot done that requires re-
 1185 shaping. If you do not want to spend space storing a reformatted data frame into a whole

new object, just to make a single plot, you can use `ggplot` as final part of a tidyverse pipeline. For example, starting from the original data:

```
data %>%
  pivot_wider(names_from = "time", values_from = c("cfu", "OD600")) %>%
  ggplot(aes(x = OD600_T0, y = OD600_T1)) +
  geom_point()
```



Notice the use of the pipe `%>%` to pass the resulting data frame on to the `ggplot` command. Because `ggplot` is called with a pipe, its first argument is already passed (it is the data frame coming through the pipe), so we only need to pass the second argument, i.e. the aesthetics mapping, to the `ggplot` function.

5.8 Customization

Now that we saw everything there is to know about structuring a `ggplot`, it is time to learn how to polish it (the easiest and most rewarding part!).

5.8.1 Scales

Every aesthetics can be scaled. This includes specifying what values an aesthetics can take (e.g. what colors to pick, or what range of transparencies to use), possible break points along the legend, or legend titles and labels, among others. Use the `scale_*` family of functions for that. There are many such functions, because many aesthetics can be modified, but the logic behind their naming is always the same:

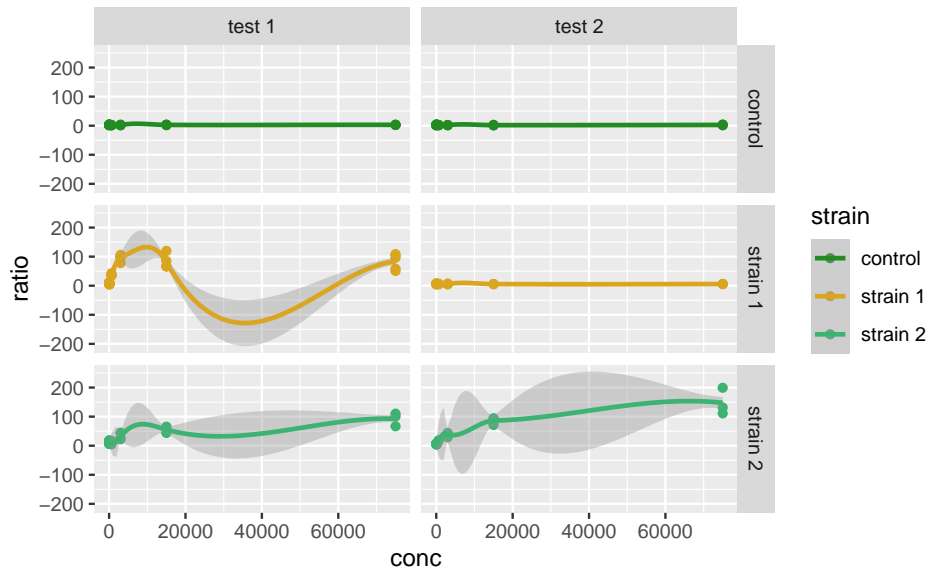
```
scale_<AESTHETIC>_<TYPE>
```

where `<AESTHETIC>` is replaced by the aesthetic you want to scale (e.g. `color`, `size`, `alpha`) and `<TYPE>` is the type of variable that is mapped to this aesthetic (common types are `continuous`, `discrete` and `manual`). Some scaling functions do not take a `<TYPE>` but just an `<AESTHETIC>` in their name, e.g. `scale_alpha`.

In our example, if we color-code points according to their `strain`, which is a categorical variable, we can use `scale_color_manual` (aka `scale_colour_manual`) to manually pick the colors we want:

```
ggplot(data2, aes(x = conc, y = ratio, color = strain)) +
  geom_point() +
  geom_smooth() + # just to spice up our use of geoms
  facet_grid(strain ~ assay) +
  scale_color_manual(values = c("forestgreen", "goldenrod", "mediumseagreen"))
```

1210

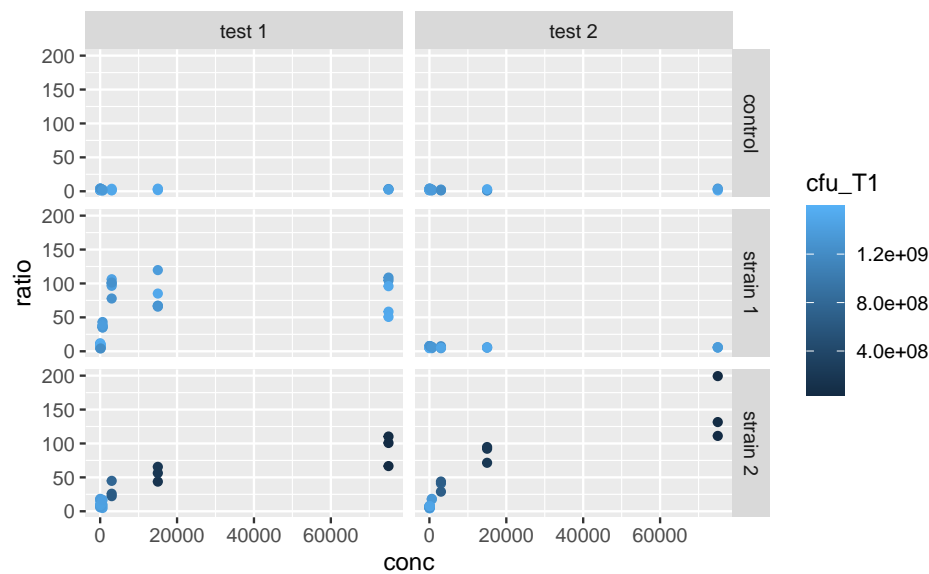


1211

1212 Alternatively, we could color-code the points based on their number of CFU at time point
 1213 T1, cfu_T1, which is a continuous variable, using `scale_color_continuous`. Without
 1214 scaling:

```
ggplot(data2, aes(x = conc, y = ratio, color = cfu_T1)) +
  geom_point() +
  facet_grid(strain ~ assay)
```

1215

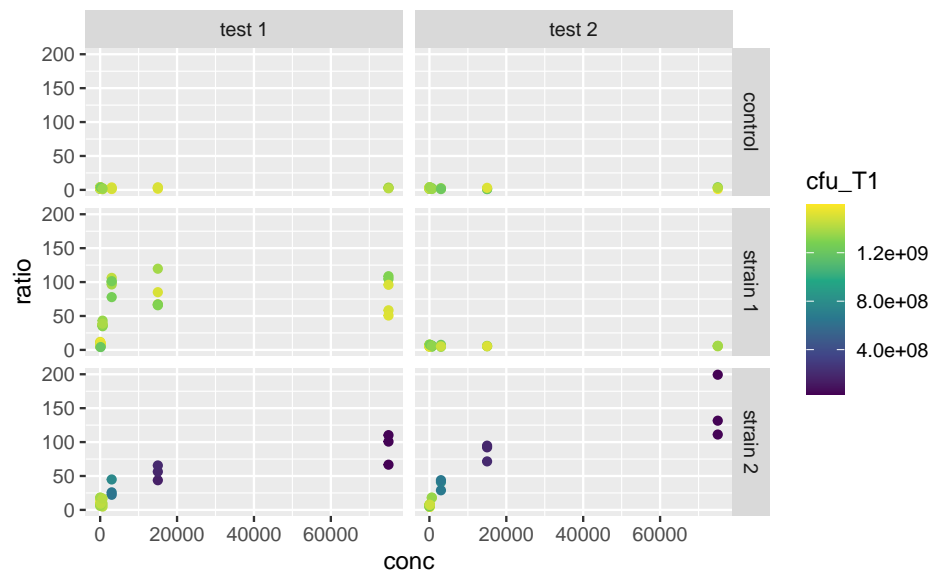


1216

1217 With scaling:

```
ggplot(data2, aes(x = conc, y = ratio, color = cfu_T1)) +
  geom_point() +
  facet_grid(strain ~ assay) +
  scale_color_continuous(type = "viridis")
```

1218



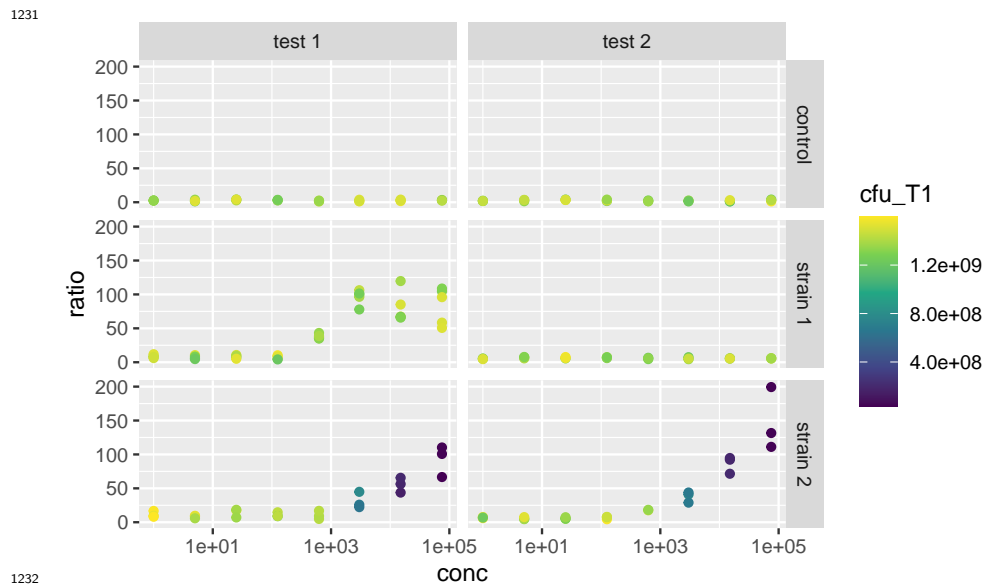
1219

1220 The arguments that are taken by the `scale_` function really depend on the use case,
 1221 e.g. `scale_color_manual` expects discrete values, `scale_color_continuous`

expects a type of built-in continuous color gradient, and `scale_color_gradient` expects a low and high color boundaries (and also a mid-gradient color in the case of `scale_color_gradient2`). But the logic shown here is similar across many aesthetics, e.g. `scale_alpha_continuous` and `scale_size_continuous` work in similar ways, both taking a range argument. So, lots of scaling functions to play with, of which we do not provide an exhaustive list here.

Mandatory aesthetics, such as `x` and `y`, also have their scaling functions. If `x` or `y` is continuous, one can e.g. use `scale_x_log10` to show this axis on a logarithmic scale, without having to log-transform the data before plotting, e.g.

```
ggplot(data2, aes(x = conc, y = ratio, color = cfu_T1)) +
  geom_point() +
  facet_grid(strain ~ assay) +
  scale_color_continuous(type = "viridis") +
  scale_x_log10()
```



More on re-scaling legend titles and labels further down.

5.8.2 Labels

The functions `ggtitle`, `xlab`, `ylab` and `labs` allow you to customize the labels shown for each aesthetics (remember that the `x`- and `y`-axes are aesthetics too), and for the main title of the plot. On to a full-fledged example:

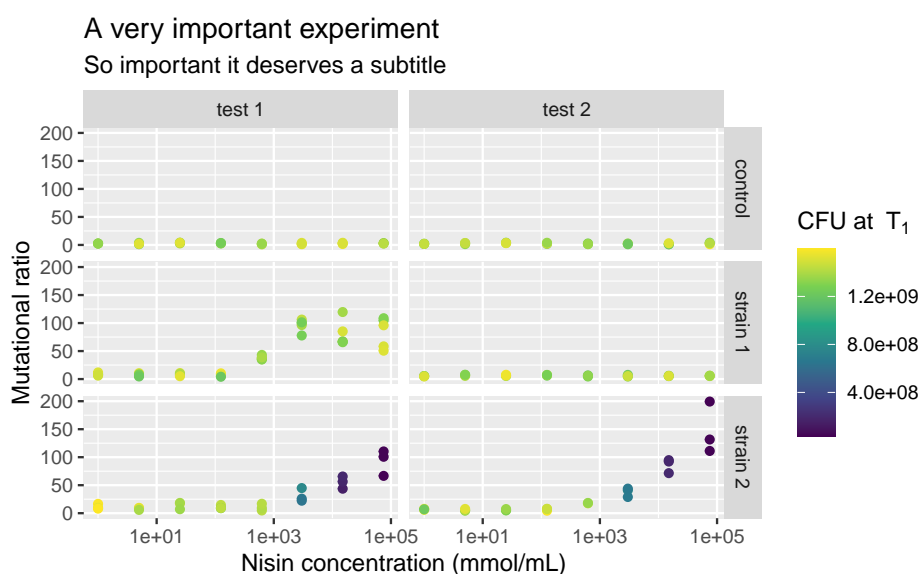
```
p <- ggplot(data2, aes(x = conc, y = ratio, color = cfu_T1)) +
  geom_point() +
  facet_grid(strain ~ assay) +
```

```

scale_color_continuous(type = "viridis") +
scale_x_log10() +
xlab("Nisin concentration (mmol/mL)") +
ylab("Mutational ratio") +
labs(color = parse(text = "'CFU at '~T[1]")) + # plotmath expression
ggtitle(
  "A very important experiment",
  "So important it deserves a subtitle"
)
p

```

1238



1239

1240 Note that `xlab` and `ylab` are wrappers around `labs`, meaning that we could have provided
 1241 labs with `x = ...` and `y = ...` in addition to `color = ...`, its arguments just
 1242 need to take the names of the aesthetics. If you want no labels, use e.g. `xlab(NULL)` or
 1243 `ylab(NULL)`.

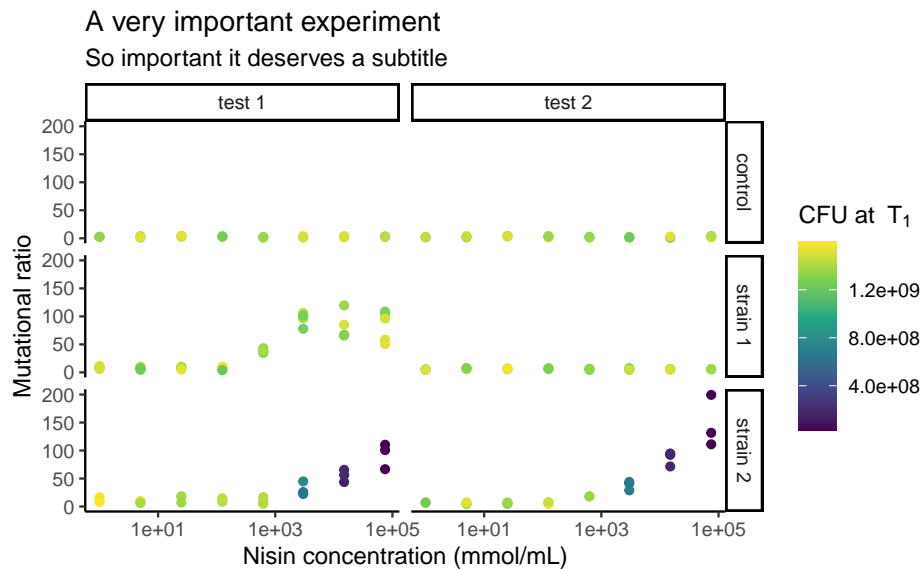
1244 Also notice the use of `parse` to display mathematical notations using the `plotmath` syntax.
 1245 This is not part of the tidyverse though, so it is a story for another day, feel free to
 1246 look it up (type `?bquote`)!

1247 5.8.3 Themes

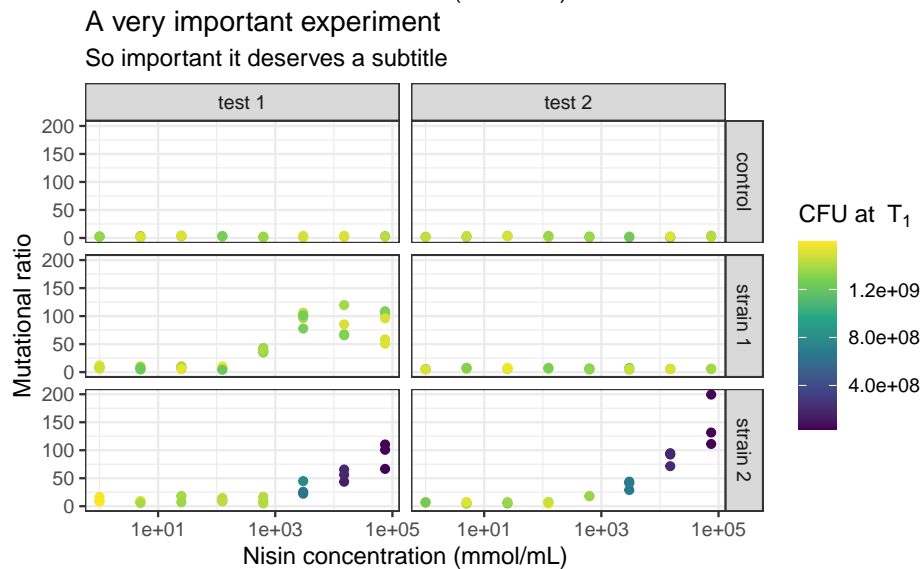
1248 You may be already frustrated that all plots have this same grey default `ggplot2`
 1249 background. Of course, it is possible to change this too by playing with the theme
 1250 functions. There are other built-in themes than the default grey one, such as `theme_bw`
 1251 or `theme_classic`:

```
p + theme_classic()
p + theme_bw()
```

1252



1253



1254

1255 The individual elements of the theme, e.g. the background grid or the color of the panel,
1256 can be customized using the arguments in the theme function. The theme function can
1257 also be used to modify stuff related to the legend or the axes of the plots. For example:

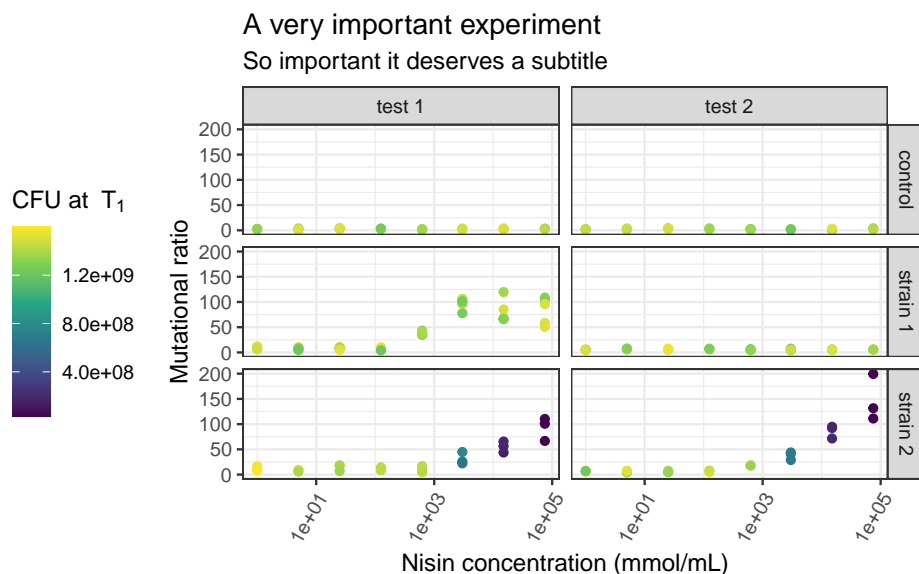
```
p <- p +
  theme_bw() +
  theme(
```



```

legend.position = "left",
axis.text.x = element_text(angle = 60, hjust = 1)
)
p

```



1258

Here, `legend.position` is sort of self-explanatory, but `axis.text.x` is a bit more subtle. Some elements of the theme, such as the text of the axes, need a series of graphical parameters in order to be modified, and the graphical parameters that can be used depend on the type of object those theme elements are (are they `text`, `rect` or `line`?). We use the `element_*` family of functions to pass those graphical parameters to our theme elements of interest. Here, we use `element_text` to transform the text on the x-axis by rotating it by an angle of 60 degrees, and then align each label to the right (`hjust` stands for “horizontal justification”). Again, lots of combinations are possible. Explore!

1267 5.8.4 Legend

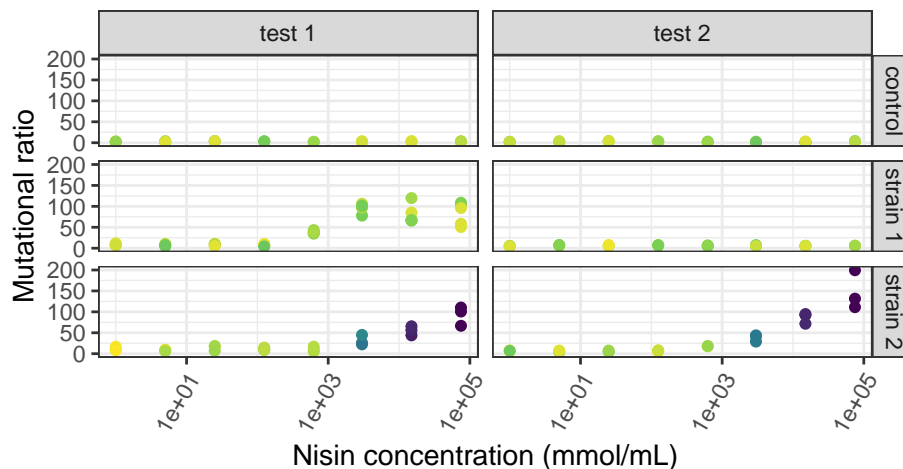
The one thing I Google the most, without a doubt, is “custom legend in ggplot”, because I always forget how to choose which legend to show, e.g. if I want to display the color legend but not the alpha legend. So here it is: to hide *all* the legends, use:

```
p + theme(legend.position = "none")
```

1271

A very important experiment

So important it deserves a subtitle



1272

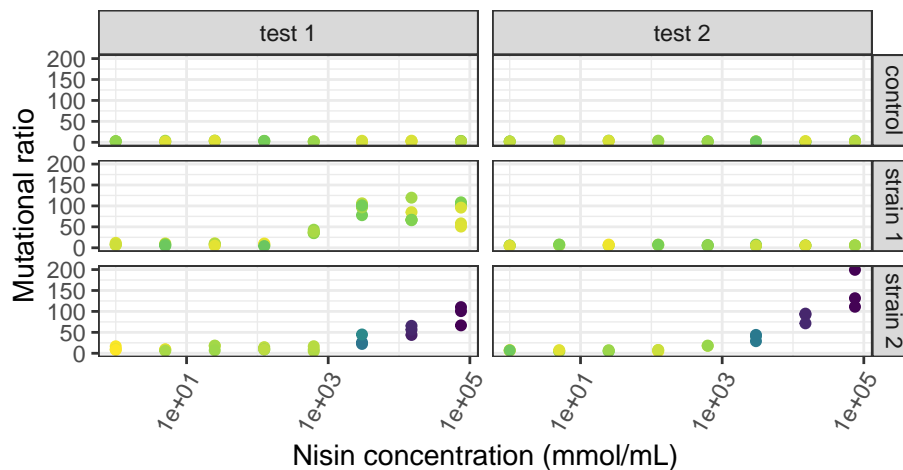
1273 And to selectively hide *some* legends, use guides:

```
p + guides(color = FALSE)
```

1274

A very important experiment

So important it deserves a subtitle

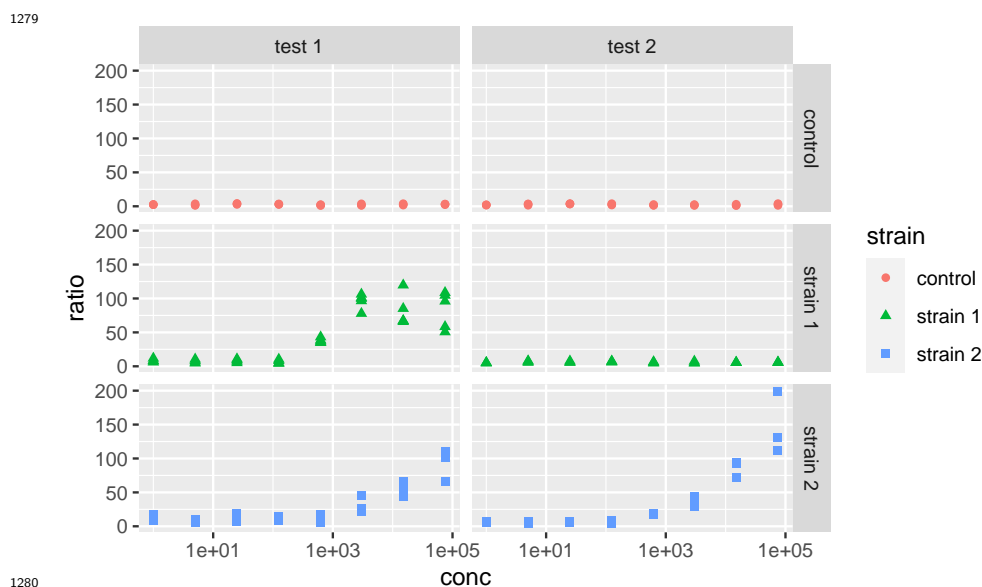


1275

1276 It is also important to remember that ggplot2 will try to combine legends together when-
 1277 ever it can. If the same variable is mapped to two different aesthetics, e.g. shape and color,
 1278 only one legend will appear:

```
ggplot(data2, aes(x = conc, y = ratio, color = strain, shape = strain)) +  
  geom_point() +
```

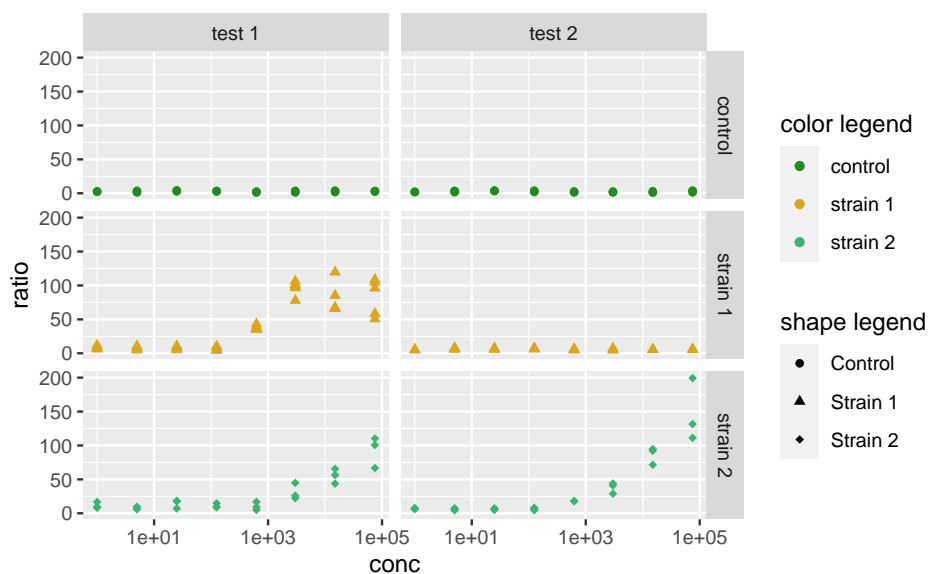
```
facet_grid(strain ~ assay) +
scale_x_log10()
```



1281 But this behavior can be controlled. You can use the arguments of the `scale_` functions
 1282 to pass custom titles and labels to the legends. And if the legends mapping to the same
 1283 variable have different titles or labels, they will be shown separately:

```
ggplot(data2, aes(x = conc, y = ratio, color = strain, shape = strain)) +
  geom_point() +
  facet_grid(strain ~ assay) +
  scale_x_log10() +
  scale_color_manual(
    "color legend", values = c("forestgreen", "goldenrod", "mediumseagreen")
  ) +
  scale_shape_manual(
    "shape legend", values = c(16, 17, 18),
    labels = c("Control", "Strain 1", "Strain 2")
  )
```

1284



Note that you can also use this trick to combine different legends together, by giving them the same titles and labels.

5.9 Combining plots

This was more or less what you need to know to be operational when plotting *single* ggplots. But what if the facetting option is not enough, and you want to combine multiple plots into a single figure? `ggplot2` itself does not do that, but the good news is, there are many packages that do. Those include `patchwork`, `cowplot`, `grid`, `gridExtra`, `egg` or `aplot` (and probably more).

One term that these packages often use is `grob`. A `grob` is a `ggplot`-like object, such as a `ggplot` but could also be a single text label in the middle of a plotting window. These packages essentially assemble grobs together.

`patchwork` is personally my favorite so I will focus on this one here. It has the advantage to automatically align the frames of the different plots across the different subplots (I found that this is not entirely true when combining `ggtree` objects with other plots, `aplot` is better for this specific case). It also has an excellent, succinct documentation.

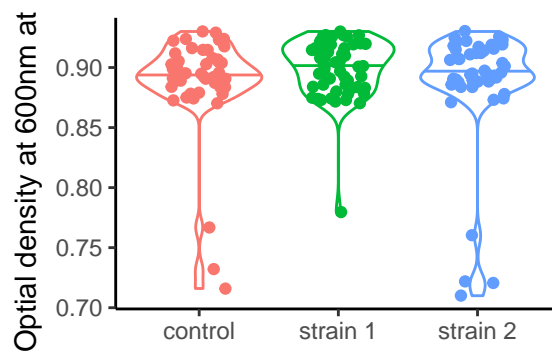
Let us look at an example, where we assign the previous plot to `p1` and make a new plot to combine it with, called `p2`:

```
p1 <- p
p2 <- ggplot(data2, aes(x = strain, y = OD600_T1, color = strain)) +
  geom_violin(draw_quantiles = 0.5) +
  geom_jitter(width = 0.2) +
  theme_classic() +
  xlab(NULL) +
```

```

ylab(parse(text = "'Optial density at 600nm at '~T[1]")) +
theme(legend.position = "none")
p2

```



1303

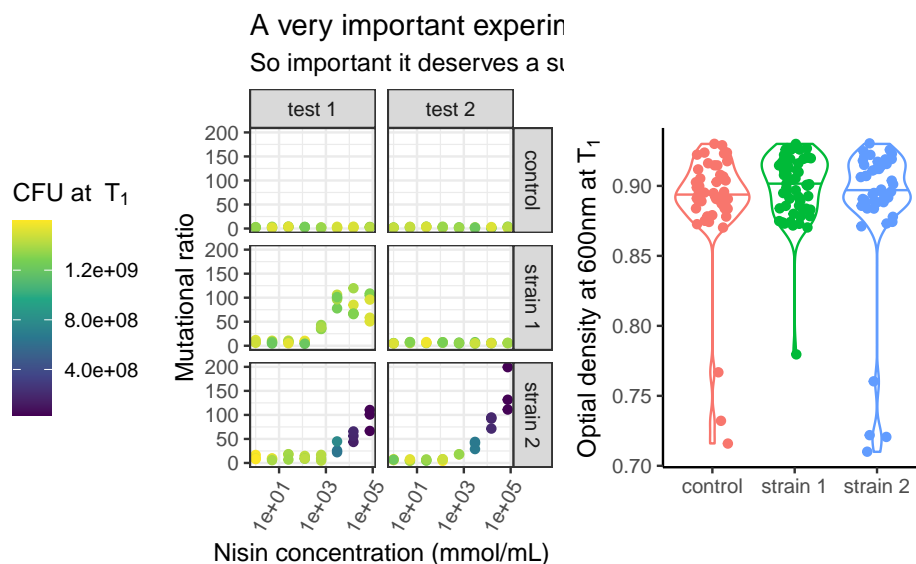
1304 In patchwork, we would combine both using:

```

library(patchwork)
p1 + p2

```

1305



1306

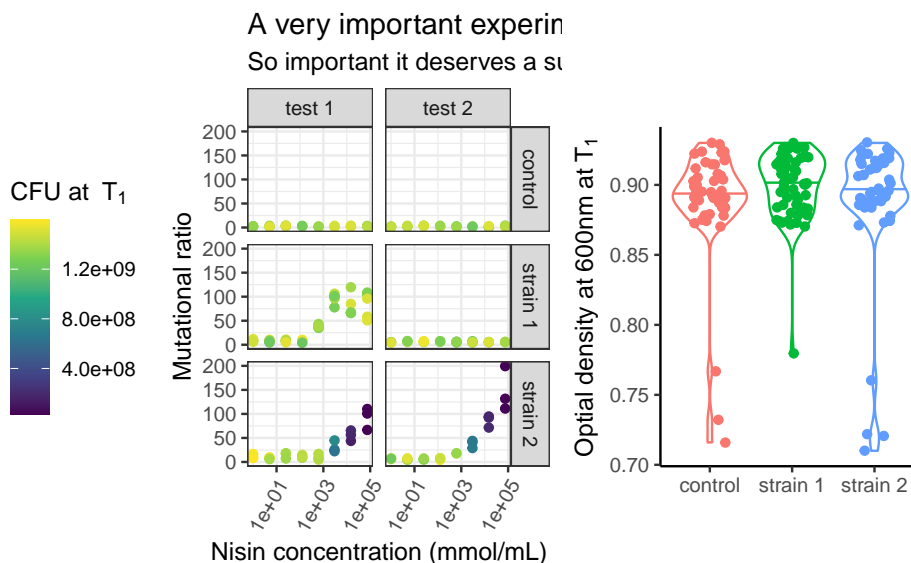
1307 patchwork uses operators such as +, / or | to assemble the plots in various layouts. It
 1308 looks simple, but a caveat of this approach is that it may become tedious when assembling,
 1309 e.g. 15 small plots, or plots from a list of unknown length. The programmatic equivalent
 1310 of the above example is:

```

wrap_plots(p1, p2) # or even more programmatic, wrap_plots(list(p1, p2))

```

1311



More customization can be added to the previous combination of plots, such as layout specifications, e.g. controlling the position and dimension of the different plots, or annotations, e.g. global title, labelling each plot or capturing the legends of all the plots and show it as one global legend). But this is a `ggplot2` tutorial and we just want you to know that `patchwork` and friends exist, so go check them out to know more about what they can do!

5.10 Saving a plot

Last but not least, ggplots have their own saving function: `ggsave` (it also works on combinations of ggplots made by `patchwork` or `cowplot`), which guesses the extension of your figure (e.g. `.png` or `.pdf`) from the file name you provide. You can also give it specific width, height and dpi (resolution) parameter values.

5.11 High throughput plotting workflow

As we mentioned in the part about combining plots, sometimes we want to do things many times (in my case I often make 100 times the same figure, just for different replicate simulations). Of course we would not copy and paste many times the same snippet of code, or write 100 times `+` to assemble some plots (by now we are advanced R users, after all). This is where we can make use, again, of the combination of tidyverse tools, and especially `purrr`.

Let us make a function that plots the number of CFU against the optical density, faceted by time point (so, that function expects a time point-wise dataset, such as `data`):

```
plot_this <- function(data) {
```

```

ggplot(data, aes(x = OD600, y = cfu, color = cfu)) +
  geom_point() +
  facet_grid(. ~ time) +
  theme_classic() +
  scale_color_continuous(type = "viridis") +
  theme(legend.position = "none") +
  xlab(parse(text = "'OD at 600nm at '~T[1]")) +
  ylab("CFU")
}

```

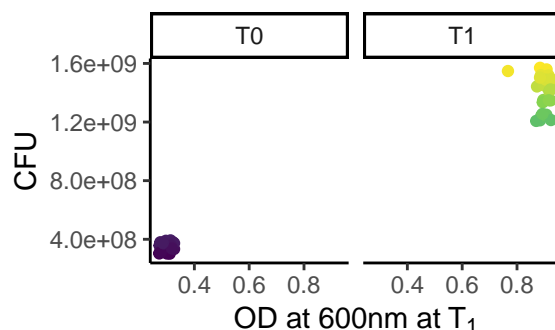
1333 Note that this does not plot anything, it is just a function that will if called on a dataset.

1334 The objective is to apply this function to each strain-assay combination, thus getting
 1335 *one plot* per combination. We can check that this function works as expected for a single
 1336 combination using our friend `dplyr`:

```

data %>%
  filter(strain == "control", assay == "test 1") %>%
  plot_this()

```



1337

1338 which works because `plot_this` takes a data frame as first argument.

1339 Now that we are happy with our single-plot function, we `tidyr::nest` our data frame
 1340 into all the relevant combinations of `strain` and `assay`, and we `purrr::map` through
 1341 the resulting list-column to produce many ggplots in one go:

```

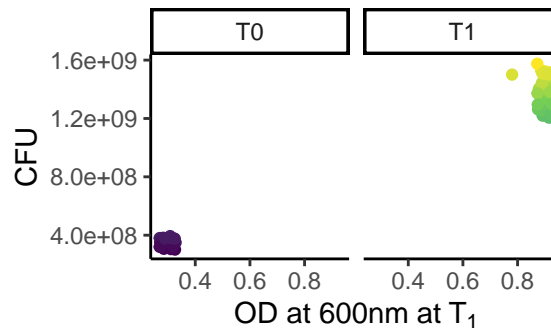
newdata <- data %>%
  group_by(assay, strain) %>%
  nest() %>%
  mutate(fig = map(data, plot_this))
newdata
#> # A tibble: 6 x 4
#> # Groups:   strain, assay [6]
#>   strain assay data          fig
#>   <chr>   <chr> <list>      <list>
#> 1 strain 1 test 1 <tibble [72 x 5]> <gg>
#> 2 control test 1 <tibble [48 x 5]> <gg>

```

```
#> 3 strain 2 test 1 <tibble [48 x 5]> <gg>
#> 4 strain 2 test 2 <tibble [46 x 5]> <gg>
#> 5 strain 1 test 2 <tibble [48 x 5]> <gg>
#> 6 control test 2 <tibble [48 x 5]> <gg>
```

1342 where the new list-column `fig` is a list of `ggplot` objects, that we can check individually:

```
newdata$fig[[1]]
```



1343

1344 Looks purrrfect.

1345 If you ask yourself why going through this hassle with only two assays and three strains,
1346 just think about a case where you would have hundreds of e.g. simulations, sequences,
1347 field sites or study species.

1348 Let us go a bit further. Now we want to combine plots for each `strain` into one figure per
1349 assay. We also want to give the resulting combined plot a figure file name, and save all
1350 the figures. There we go:

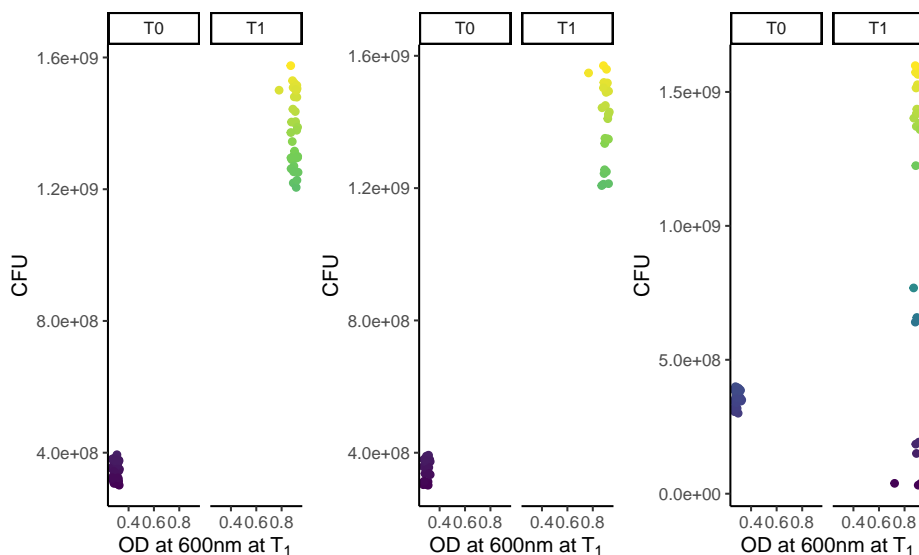
```
newdata <- newdata %>%
  select(-data) %>% # just to clean up a bit
  group_by(assay) %>%
  nest() %>%
  mutate(combifig = map(data, ~ wrap_plots(.x$fig)))
newdata
#> # A tibble: 2 x 3
#> # Groups:   assay [2]
#>   assay data      combifig
#>   <chr> <list>      <list>
#> 1 test 1 <tibble [3 x 2]> <patchwrk>
#> 2 test 2 <tibble [3 x 2]> <patchwrk>
```

1351 Note that we use the *formula*-way of passing functions to `map` (using `~`), which is
1352 more succinct than the *lambda* way (using an anonymous function `function(x)`
1353 `wrap_plots(x)`), and where `.x` is interpreted as an element of the list we iterate through
1354 (here the list-column `data`). Please refer to the `purrr` documentation for more details.

1355 As we can see, we have created a new list-column `combifig`, filled with patchwork ob-

1356 jects, i.e. combined plots:

```
newdata$combifig[[1]]
```



1357

1358 We could of course further customize the assembly of plots, but we refer the reader to the
1359 patchwork documentation for this.

1360 Last step, preparing file names and saving the figures, using old friends from the tidy-
1361 verse:

```
library(glue)
newdata %>%
  mutate(figure = glue("data/figure_{str_replace(assay, ' ', '_')}.png")) %>%
  mutate(saved = walk2(figure, combifig, ggsave))
#> # A tibble: 2 x 5
#> # Groups:   assay [2]
#>   assay data          combifig figure          saved
#>   <chr> <list>         <list>   <glue>         <glue>
#> 1 test 1 <tibble [3 x 2]> <patchwrk> data/figure_test_1.p~ data/figure_test_1.p~
#> 2 test 2 <tibble [3 x 2]> <patchwrk> data/figure_test_2.p~ data/figure_test_2.p~
```

1362 5.12 Want more?

1363 ggplot2 is undoubtedly one of the largest chunks of the tidyverse. Here we tried to pro-
1364 vide a global understanding of how it works, but we could not dig into all possible func-
1365 tions it has (this would take us days). Hopefully now you are armed with the necessary
1366 knowledge to be able to find the missing pieces you need.

1367 Some things, however, are missing from ggplot2. Fortunately, there are many of exten-
1368 sions building on ggplot2 that respect the same grammar. Some of them implement

new geoms (e.g. such as `ggridges` for ridge-density plots, `ggadar` for radial plots, or `gg halves` for mixes of geoms), others combine plots together (examples cited above), offer more complex themes (e.g. `ggnewscale` for multiple scales of the same type to coexist, or `ggdark` for a dark background), deal with complicated objects that are not trivial to fit in data frames (e.g. `ggtree` for tree-like objects or `ggraph` for networks), or provide shortcuts to quickly produce publication-ready figures for common plot layouts and their corresponding statistical analyses (e.g. `ggpubr`, `ggrapid` or `GGally`). There are even packages for animated graphics (`gganimate`), interactive plot building (`esquisse`) or 3D surface plotting (`rayshader`). See the links below!

5.13 References

- The `ggplot2` website where you can find links to other resources
- The `ggplot2` cheatsheet
- The dedicated chapter in R for Data Science
- A non-exhaustive list of extensions at this link
- The R graph gallery for inspiration
- Hadley's article explaining the grammar of graphics
- The `patchwork` documentation
- The `ggtree` and `ggraph` packages

1387 **Chapter 6**

1388 **Regular expressions and**
1389 **testthat**

6.1 Introduction



'Regular expressions' from <https://xkcd.com/208>

6.1.1 Goal

In this chapter, you will learn:

- How to express your ideas as a regular expression
- Verify that you indeed did so

6.1.2 Why is this important?

Knowing the basics of regular expressions, prevents you having to hand-craft functions to detect patterns in any text.

Being able to verify your own assumptions allows you to speed up any development of any

code. It is estimated that 50-90% of all the time, we are debugging our code. Being good at testing, is the way to become faster.

6.1.3 What are regular expressions?

A regular expression 'is a sequence of characters that define a search pattern'. Such a pattern may be a zip code, a date, or any other text of which you can say: 'this is not just text, it is a [something]'.

For example, take a Dutch zip code: 9747 AG. Dutch zip code have four digits, a space and then two uppercase alphabet characters.

A regex for this is `[:digit:]{4} [:upper:]{2}`.

6.1.4 Applications

DNA data:

```
>KU215420.1|Felinecoronavirus|Feliscatus|Belgium|2013|Envelope
ATGATGTTTCCTAGGGCATTACTATCATAGATGACCATGGTATGGTTGTTAGTGTCTTC
>KP143511.1|Felinecoronavirus|Feliscatus|UnitedKingdom|2013|Envelope
ATGATGTTTCCTAGGGCATTACTATCATAGACGACCATGGTATGGTTGTTAGTGTCTTC
```

Protein data:

```
>sp|P0DTC2|SPIKE_SARS2 Spike glycoprotein OS=Severe acute respiratory syndrome coronavirus 2 OX=2697049 C
MFVFLVLLPLVSSQCVNLTTRTQLPPAYTNSFTRGVYYPDKVFRSSVLHSTQDLFLPFFS
>sp|P0DTC5|VME1_SARS2 Membrane protein OS=Severe acute respiratory syndrome coronavirus 2 OX=2697049 PE=3
MADSNGTITVEELKKLLEQWNLVIGFLFTWICLLQFAYANRRNFLYIIKLIFLWLLWPV
```

Most messy Excel sheets :-)

1	Project Name	Started	Date Due	Client	% Complete	Status
2	A Scandal in Bohemia	02/12/2002		B King	95%	Monitoring
3	The Red-Headed League	03/11/2014		Jabez Wilson	10%	Considering
4	Boscombe Valley	04/10/2012	01/05/2015	Alice Turner	15%	Considering
5	Blue Carbuncle	07/07/2010		Helen Stoner	25%	Consulting
6	Speckled Band	05/03/2013	01/02/2015	Victor Hatherley	75%	Monitoring
7	The Sign of Four	Yes		Mary Morstan	10%	Gathering
8	Charles Augustus	01/03/2011		CA Milverton	45%	Investigating
9	Solitary Cyclist	Mar-13		Violet Smith	5%	Cnsidering

6.1.5 Using regexes in R



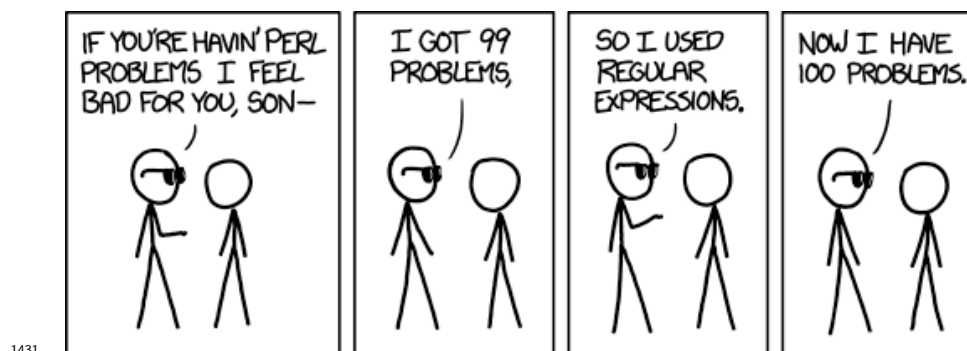
1426 The ‘stringr’ logo. ‘stringr’ is part of the Tidyverse

1427 Multiple R functions to work with regular expressions:

- 1428 • `stringr::str_`
- 1429 • `egrep`, `grepl`, `gsub`

```
library(stringr)
```

1430 6.1.6 Dangers of regexes



1432 ‘Perl problems’, from <https://xkcd.com/1171/>

1433 Regexes have different dialects, such as POSIX and perl. Within R, there are the base R
1434 dialect and the Tidyverse dialect.

1435 We’ll have to test!

1436 6.2 Testing



1437

1438 From George Dinwiddie's blog, [http://blog.gdinwiddie.com/2012/12/26/tdd-](http://blog.gdinwiddie.com/2012/12/26/tdd-hat/)
1439 [hat/](http://blog.gdinwiddie.com/2012/12/26/tdd-hat/)

1440 6.2.1 Why test?

- 1441 • To be sure your code is correct
- 1442 • Spend less time fixing bugs
- 1443 • Unit of communication
- 1444 • Clean software interface

1445 6.2.2 Our first test

1446 The `testthat` package is the Tidyverse package to write tests.

```
library(testthat)
```

1447 All test functions start with `expect_`, for example:

```
expect_true(1 + 1 == 2)
expect_false("cat" == "dog")
expect_equal(1 + 1, 2)
```

1448 If a test fails:

```
expect_equal(1 + 1, 3)
#> Error: 1 + 1 not equal to 3.
#> 1/1 mismatches
#> [1] 2 - 3 == -1
```

1449 6.3 Detect a full match

1450 Here, we will detect simple patterns using `str_which`.

1451 Tip: run `?str_which` for its documentation.

Detect Matches



TRUE
TRUE
FALSE
TRUE

str_detect(string, **pattern**) Detect the presence of a pattern match in a string.
`str_detect(fruit, "a")`



1
2
4

str_which(string, **pattern**) Find the indexes of strings that contain a pattern match.
`str_which(fruit, "a")`



0
3
1
2

str_count(string, **pattern**) Count the number of matches in a string.
`str_count(fruit, "a")`



start and
2 4
4 7
NA NA
3 4

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**. `str_locate(fruit, "a")`

1452

1453 From 'Work with Strings Cheatsheet', [https://rstudio.com/resources/](https://rstudio.com/resources/cheatsheets)
1454 cheatsheets

1455 6.3.1 str_which demo

```
fruit <- c("apple", "banana", "pinapple")
expect_equal(str_which(fruit, "banana"), 2)
expect_equal(str_which(fruit, "apple"), c(1, 3))
expect_equal(str_which(fruit, "submarine"), integer(0))
```


1456 6.3.2 Example exercise: has_a_one

1457 Write a function called `has_a_one` that detects if a character vector contains at least one
1458 one.

1459 To be precise: 'a one' is a string that starts with a 1, then ends directly.

1460 These tests must pass:

```
1461 expect_true(has_a_one("1"))
1462 expect_true(has_a_one(c("X", "1")))
1463 expect_true(has_a_one(c("1", "1")))
1464 expect_false(has_a_one("X"))
1465 expect_false(has_a_one("11"))
1466 expect_false(has_a_one("1 1"))
1467 expect_false(has_a_one(integer(0)))
1468 expect_false(has_a_one(NULL))
1469 expect_false(has_a_one(NA))
```

1470 Use the anchors as shown on the cheatsheet to specify that the complete string, from
1471 begin to the end, must consist out of characters

regex

^a

a\$

matches

start of string

end of string

1472

1473 From 'Work with Strings Cheatsheet', [https://rstudio.com/resources/](https://rstudio.com/resources/cheatsheets)
1474 cheatsheets

1475 Here is a stub of the function, but feel free to use your own function body:

```
has_a_one <- function(text) {
  length(stringr::str_which(text, "your regex here")) >= 1
}
```

1476 6.3.2.1 Answer has_a_one

```
has_a_one <- function(text) {
  length(stringr::str_which(text, "^1$")) >= 1
}
```

1477 Note that you may have had a different regex. No worries: if all tests pass, you did a great
1478 job!

```
expect_true(has_a_one("1"))
expect_true(has_a_one(c("X", "1")))
```

```

expect_true(has_a_one(c("1", "1")))
expect_false(has_a_one("X"))
expect_false(has_a_one("11"))
expect_false(has_a_one("1 1"))
expect_false(has_a_one(integer(0)))
expect_false(has_a_one(NULL))
expect_false(has_a_one(NA))

```

1479 Also, using another `stringr` function, such as `str_count`, `str_subset` or `str_match`
 1480 are all valid as well. It just made the code longer. Also here: if all tests pass, you did a great
 1481 job!

1482 6.3.3 Exercise: has_a_digit

1483 Write a function called `has_a_digit` that detects if a character vector contains at least
 1484 one digit. To be precise, ‘a digit’ is a string that starts with a (decimal) digit, the ends
 1485 directly.

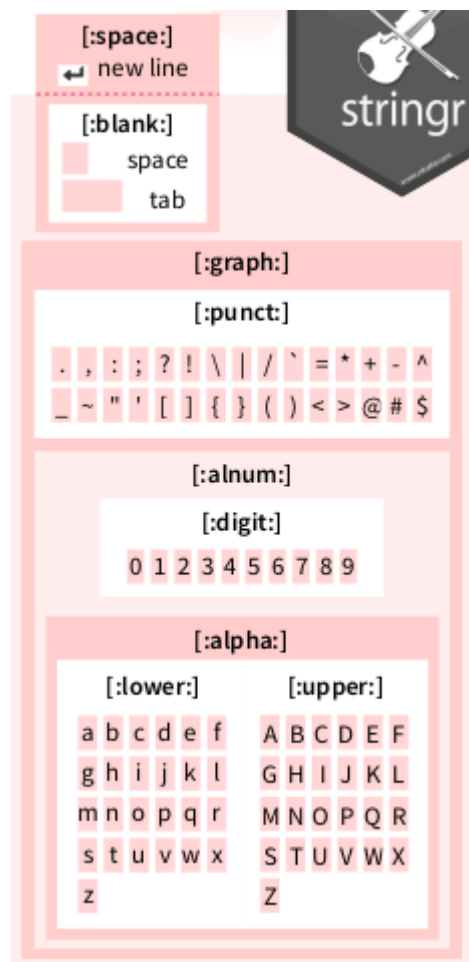
1486 These tests must pass:

```

1487 expect_true(has_a_digit("0"))
1488 expect_true(has_a_digit("1"))
1489 expect_true(has_a_digit(c("1", "2")))
1490 expect_true(has_a_digit(c("X", "1")))
1491 expect_false(has_a_digit(""))
1492 expect_false(has_a_digit("12"))
1493 expect_false(has_a_digit("1 2"))
1494 expect_false(has_a_digit("X"))
1495 expect_false(has_a_digit(character(0)))
1496 expect_false(has_a_digit(NULL))
1497 expect_false(has_a_digit(NA))

```

1498 Use the regex pattern as shown on the cheatsheet to specify a digit:



1499

1500 From 'Work with Strings Cheatsheet', [https://rstudio.com/resources/](https://rstudio.com/resources/cheatsheets)
 1501 cheatsheets

1502 Here is a stub of the function, but feel free to use your own function body:

```
has_a_digit <- function(text) {
  length(stringr::str_which(text, "your regex here")) >= 1
}
```

1503 6.3.3.1 Answer: has_a_digit

```
has_a_digit <- function(text) {
  length(stringr::str_which(text, "^[:digit:]+$")) >= 1
}

expect_true(has_a_digit("0"))
expect_true(has_a_digit("1"))
```

```
expect_true(has_a_digit(c("1", "2")))
expect_true(has_a_digit(c("X", "1")))
expect_false(has_a_digit(""))
expect_false(has_a_digit("12"))
expect_false(has_a_digit("1 2"))
expect_false(has_a_digit("X"))
expect_false(has_a_digit(character(0)))
expect_false(has_a_digit(NULL))
expect_false(has_a_digit(NA))
```

1504 6.3.4 Exercise: has_a_word

1505 Write a function called `has_a_word` that detects if a string is a word. To be precise (and
1506 to simplify), 'a word' starts with one or more lowercase characters, then ends directly.

1507 These tests must pass:

```
1508 expect_true(has_a_word("a"))
1509 expect_true(has_a_word("an"))
1510 expect_true(has_a_word("apple"))
1511 expect_true(has_a_word(c("an", "apple")))
1512 expect_true(has_a_word(c("", "apple")))
1513 expect_false(has_a_word("."))
1514 expect_false(has_a_word("X"))
1515 expect_false(has_a_word("hI"))
1516 expect_false(has_a_word("an apple"))
1517 expect_false(has_a_word(character(0)))
1518 expect_false(has_a_word(NULL))
1519 expect_false(has_a_word(NA))
```

1520 Use the quantifiers as shown on the cheatsheet to specify that one needs one or more
1521 characters:

regexp	matches
<code>a?</code>	zero or one
<code>a*</code>	zero or more
<code>a+</code>	one or more
<code>a{n}</code>	exactly n
<code>a{n, }</code>	n or more
<code>a{n, m}</code>	between n and m

1522

1523 From 'Work with Strings Cheatsheet', [https://rstudio.com/resources/](https://rstudio.com/resources/cheatsheets)
 1524 cheatsheets

1525 Here is a stub of the function, but feel free to use your own function body:

```
has_a_word <- function(text) {
  length(stringr::str_which(text, "your regex here")) >= 1
}
```

1526 6.3.4.1 Answer: has_a_word

```
has_a_word <- function(text) {
  length(stringr::str_which(text, "^[[:lower:]]+$")) >= 1
}
```

```
expect_true(has_a_word("a"))
expect_true(has_a_word("an"))
expect_true(has_a_word("apple"))
expect_true(has_a_word(c("an", "apple")))
expect_true(has_a_word(c("", "apple")))
expect_false(has_a_word("."))
expect_false(has_a_word("X"))
expect_false(has_a_word("hI"))
expect_false(has_a_word("an apple"))
expect_false(has_a_word(character(0)))
expect_false(has_a_word(NULL))
expect_false(has_a_word(NA))
```

1527 6.3.5 Exercise: has_dna_seq (alternates)

1528 Write a function called `has_dna_seq` that detects if a character vector contains one
 1529 or more DNA sequences. To be precise, ‘a DNA sequence’ starts with one or more
 1530 nucleotides (an ‘A’, ‘C’, ‘G’ or ‘T’), then ends directly.

1531 These tests must pass:

```
1532 expect_true(has_dna_seq("A"))
1533 expect_true(has_dna_seq(c("A", "CGT")))
1534 expect_true(has_dna_seq(c("", "CGT")))
1535 expect_false(has_dna_seq("Ax"))
1536 expect_false(has_dna_seq("A C"))
1537 expect_false(has_dna_seq(character(0)))
1538 expect_false(has_dna_seq(NULL))
1539 expect_false(has_dna_seq(NA))
```

1540 Use the alternates as shown on the cheatsheet to specify that each character must be one
 1541 of the four nucleotides:

regex	matches
<code>ab d</code>	or
<code>[abe]</code>	one of
<code>[^abe]</code>	anything but
<code>[a-c]</code>	range

1543 From ‘Work with Strings Cheatsheet’, [https://rstudio.com/resources/](https://rstudio.com/resources/cheatsheets)
 1544 [cheatsheets](https://rstudio.com/resources/cheatsheets)

1545 Here is a stub of the function, but feel free to use your own function body:

```
has_dna_seq <- function(text) {
  length(stringr::str_which(text, "your regex here")) >= 1
}
```

1546 6.3.5.1 Answer: has_dna_seq

```
has_dna_seq <- function(text) {
  length(stringr::str_which(text, "[ACGT]+$")) >= 1
}

expect_true(has_dna_seq("A"))
expect_true(has_dna_seq(c("A", "CGT")))
expect_true(has_dna_seq(c("", "CGT")))
expect_false(has_dna_seq("Ax"))
expect_false(has_dna_seq("A C"))
expect_false(has_dna_seq(character(0)))
```

```
expect_false(has_dna_seq(NULL))
expect_false(has_dna_seq(NA))
```

1547 6.4 Extract a pattern for one submatch

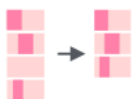
1548 Here, we will extract a pattern using `str_match`.

1549 Tip: run `?str_match` for its documentation.

Subset Strings



str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector.
str_sub(fruit, 1, 3); str_sub(fruit, -2)



str_subset(string, **pattern**) Return only the strings that contain a pattern match.
str_subset(fruit, "b")



str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*



str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each () group in pattern. Also **str_match_all**.
str_match(sentences, "(a|the) ([^]+)")

1550

1551 From 'Work with Strings Cheatsheet', [https://rstudio.com/resources/](https://rstudio.com/resources/cheatsheets)
 1552 [cheatsheets](https://rstudio.com/resources/cheatsheets)

1553 6.4.1 str_match

1554 `str_match` allows to extract multiple matches at once:

```
text <- c(
  "2020/03/14",
  "not a date",
  "2020/07/22"
)
str_match(
  text, "([[:digit:]]{4})/([[:digit:]]{2})/([[:digit:]]{2})"
)
#>      [,1]      [,2]      [,3]      [,4]
#> [1,] "2020/03/14" "2020" "03" "14"
```

```
#> [2,] NA NA NA NA
#> [3,] "2020/07/22" "2020" "07" "22"
```

1555 6.4.2 Context

1556 Here we will work on a DNA sequence:

```
library(readr)
text <- read_lines("data/virus.fas")
head(text, n = 10)
#> [1] ">KX722530.1|Felinecoronavirus|Feliscatus|Denmark|2015|Envelope"
#> [2] "ATGATGTTTCCTAGGGCTTTTACTATCATAGATGACCATGGTATGGTTGTAAGCGTCTTC"
#> [3] "TTCTGGCTCCTGTTGATAATTATATTGATATTGTTTCAATAGCATTGCTAAATGTTATT"
#> [4] "AAGTTATGCATGGTTTGTGCAATCTGGGTAAGACTATTATAGTACTACCTGCACGCCAT"
#> [5] "GCATATGATGCCTACAAGACTTTTATGCAAAATAAGGCATATAATCCCGACGAAGCACTT"
#> [6] "TTGGTTTGA"
#> [7] ">FJ938053.1|Felinecoronavirus|cat|NetherlandsUtrecht|2007|Envelope"
#> [8] "ATGATGTTTCCTAGGGCATTACTATCATAGATGACCATGGTATGGTTGTCAGCGTCTTC"
#> [9] "TTTTGGCTCCTGTTGATAATTATATTGATATTGTTTCAATAGCATTGCTAAATGTTATT"
#> [10] "AAGTTATGCATGGTATGTTGCAATTTGGGTAAGACTATTATAGTATTACCTGCACGCCAT"
```

1557 The data encoded in this text:

```
1558 >[DNA sequence number]|[virus name]|[host species name]|[country of host]|[year]|[protein name]
1559 [DNA sequence]
```

1560 We know that there are 30 sequences and 180 lines in that file:

```
n_sequences <- 30
n_lines <- 180
```

1561 6.4.3 Extract a character vector from a submatch

1562 Using a pattern that is specific for the DNA sequence descriptors, we get matched strings
1563 and NAs:

```
matches <- str_match(text, ">.*")
expect_is(matches, "matrix")
expect_equal(nrow(matches), n_lines)
expect_equal(ncol(matches), 1)
head(matches, n = 8)
#> [1,]
#> [1,] ">KX722530.1|Felinecoronavirus|Feliscatus|Denmark|2015|Envelope"
#> [2,] NA
#> [3,] NA
#> [4,] NA
#> [5,] NA
#> [6,] NA
```



```
#> [7,] ">FJ938053.1|Felinecoronavirus|cat|NetherlandsUtrecht|2007|Envelope"
#> [8,] NA
```

1564 Using round brackets, the matrix gives one extra column per sub-match. Here, we select
1565 for all info after the >:

```
matches <- str_match(text, ">(.*?)")
expect_is(matches, "matrix")
expect_equal(nrow(matches), n_lines)
expect_equal(ncol(matches), 2)
head(matches, n = 8)
#>      [,1]
#> [1,] ">KX722530.1|Felinecoronavirus|Feliscatus|Denmark|2015|Envelope"
#> [2,] NA
#> [3,] NA
#> [4,] NA
#> [5,] NA
#> [6,] NA
#> [7,] ">FJ938053.1|Felinecoronavirus|cat|NetherlandsUtrecht|2007|Envelope"
#> [8,] NA
#>      [,2]
#> [1,] "KX722530.1|Felinecoronavirus|Feliscatus|Denmark|2015|Envelope"
#> [2,] NA
#> [3,] NA
#> [4,] NA
#> [5,] NA
#> [6,] NA
#> [7,] "FJ938053.1|Felinecoronavirus|cat|NetherlandsUtrecht|2007|Envelope"
#> [8,] NA
```

1566 After select the second column, we get rid of the NAs using na.omit and converting to a
1567 character vector:

```
matches <- as.character(na.omit(matches[, 2]))
expect_is(matches, "character")
expect_equal(length(matches), n_sequences)
head(matches)
#> [1] "KX722530.1|Felinecoronavirus|Feliscatus|Denmark|2015|Envelope"
#> [2] "FJ938053.1|Felinecoronavirus|cat|NetherlandsUtrecht|2007|Envelope"
#> [3] "GU553362.1|Felinecoronavirus|feline|Netherlands|2007|Envelope"
#> [4] "KP143512.1|Felinecoronavirus|Feliscatus|UnitedKingdom|2013|Envelope"
#> [5] "KU215424.1|Felinecoronavirus|Feliscatus|Belgium|2013|Envelope"
#> [6] "HQ392470.1|Felinecoronavirus|feline|NetherlandsUtrecht|2007|Envelope"
```

1568 All of this in one go:

```
matches <- as.character(
  na.omit(
    str_match(text, ">(.*?)")[, 2]
```

```

    )
  )
  expect_equal(length(matches), n_sequences)
  head(matches)
#> [1] "KX722530.1|Felinecoronavirus|Feliscatus|Denmark|2015|Envelope"
#> [2] "FJ938053.1|Felinecoronavirus|cat|NetherlandsUtrecht|2007|Envelope"
#> [3] "GU553362.1|Felinecoronavirus|feline|Netherlands|2007|Envelope"
#> [4] "KP143512.1|Felinecoronavirus|Feliscatus|UnitedKingdom|2013|Envelope"
#> [5] "KU215424.1|Felinecoronavirus|Feliscatus|Belgium|2013|Envelope"
#> [6] "HQ392470.1|Felinecoronavirus|feline|NetherlandsUtrecht|2007|Envelope"

```

1569 6.4.4 Example exercise: extract_dna_seq_numbers (1 submatch)

1570 Extract the DNA sequence numbers.

1571 These tests must pass:

```

1572 dna_seq_numbers <- extract_dna_seq_numbers(text)
1573 expect_equal(n_sequences, length(dna_seq_numbers))
1574 expect_equal("KX722530.1", dna_seq_numbers[1])
1575 expect_equal("KP143511.1", dna_seq_numbers[30])

```

1576 Here is a stub of the function, but feel free to use your own function body:

```

extract_dna_seq_numbers <- function(text) {
  as.character(
    na.omit(
      str_match(text, "your regex here")[, 2]
    )
  )
}

```

1577 Note that the [, 2] denotes the second column. It can be another column as well

1578 Hint:

- 1579 • it is the text between > and |Felinecoronavirus.
- 1580 • Use \\| in your regex to indicate you want the pipe character (as a|b is the regex
- 1581 for 'a or b')

1582 6.4.5 Answer: extract_dna_seq_numbers

```

extract_dna_seq_numbers <- function(text) {
  as.character(
    na.omit(
      str_match(text, ">(.*?)\\|Felinecoronavirus.*")[, 2]
    )
  )
}

```

```

dna_seq_numbers <- extract_dna_seq_numbers(text)
expect_equal(n_sequences, length(dna_seq_numbers))
expect_equal("KX722530.1", dna_seq_numbers[1])
expect_equal("KP143511.1", dna_seq_numbers[30])

```

1583 The regex ">(.*?)\\|.*" would not work, because the asterisk is *greedy*.

1584 6.5 Extract a pattern for multiple submatches

1585 6.5.1 Context

1586 Here we will work on a proteome:

```

text <- read_lines("data/UP000464024.fasta")
head(text, n = 7)
#> [1] ">sp|P0DTC7|NS7A_SARS2 Protein 7a OS=Severe acute respiratory syndrome coronavirus 2 OX=269704
#> [2] "MKIILFLALITLATCELYHYQECVRGTTVLLKEPCSSGTYEGNSPFHPLADNKFALTCFS"
#> [3] "TQFAFACPDGVKHHVYQLRARSVSPKLFIRQEEVQELYSPIFLIVAAIVFITLCFTLKRKT"
#> [4] "E"
#> [5] ">sp|P0DTD1|R1AB_SARS2 Replicase polyprotein 1ab OS=Severe acute respiratory syndrome coronavi
#> [6] "MESLVPGFNEKTHVQLSLPVLQVRDVLVRGFGDSVEEVLSEARQHLKDGTCGLVEVEKGV"
#> [7] "LPQLEQPYVFIKRSDARTAPHGHVMVELVAELEGIQYGRSGETLGVLVPHVGEIPVAYRK"

```

1587 The data encoded in this text:

```

1588 >sp|[Sequence ID]|[Protein ID] [Protein description] OS=[Virus name] OX=[OX] GN=[GN] PE=[PE] SV=[SV]
1589 [Peptide sequence]

```

1590 We will only look at [Sequence ID] and [Protein ID].

1591 We know that:

```
n_proteins <- 13
```

1592 6.5.2 Exercise: extract_prot_and_seq_ids

1593 Extract all proteins' ID and sequence ID, in a tibble.

```
library(tibble)
```

1594 These tests must pass:

```

1595 t <- extract_prot_and_seq_ids(text)
1596 expect_true(is_tibble(t))
1597 expect_equal(n_proteins, nrow(t))
1598 expect_equal(2, ncol(t))
1599 expect_equal(colnames(t), c("seq_id", "prot_id"))
1600 expect_equal(t$seq_id[1], "P0DTC7")
1601 expect_equal(t$prot_id[1], "NS7A_SARS2")
1602 expect_equal(t$seq_id[13], "P0DTC5")

```

1603 `expect_equal(t$prot_id[13], "VME1_SARS2")`

1604 Here is a stub of the function, but feel free to use your own function body:

```
extract_prot_and_seq_ids <- function(text) {
  matrix <- na.omit(
    str_match(
      text,
      "your regex here"
    ), c(2, 3)
  )
  colnames(matrix) <- c("seq_id", "prot_id")
  tibble::as_tibble(matrix)
}
```

1605 6.5.3 Answer: extract_prot_and_seq_ids

```
extract_prot_and_seq_ids <- function(text) {
  matrix <- na.omit(
    str_match(text,
      ">sp\\|([[:graph:]]+)\\|([[:graph:]]+) ")[, c(2, 3)]
  )
  colnames(matrix) <- c("seq_id", "prot_id")
  tibble::as_tibble(matrix)
}
```

```
t <- extract_prot_and_seq_ids(text)
expect_true(is_tibble(t))
expect_equal(n_proteins, nrow(t))
expect_equal(2, ncol(t))
expect_equal(colnames(t), c("seq_id", "prot_id"))
expect_equal(t$seq_id[1], "P0DTC7")
expect_equal(t$prot_id[1], "NS7A_SARS2")
expect_equal(t$seq_id[13], "P0DTC5")
expect_equal(t$prot_id[13], "VME1_SARS2")
```

1606 **6.6 Mutate**

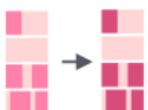
Mutate Strings



str_sub() <- value. Replace substrings by identifying the substrings with str_sub() and assigning into the results.
`str_sub(fruit, 1, 3) <- "str"`



str_replace()(string, **pattern**, replacement)
 Replace the first matched pattern in each string. `str_replace(fruit, "a", "-")`



str_replace_all()(string, **pattern**, replacement) Replace all matched patterns in each string. `str_replace_all(fruit, "a", "-")`

1607

1608 From 'Work with Strings Cheatsheet', [https://rstudio.com/resources/](https://rstudio.com/resources/cheatsheets)
 1609 cheatsheets

```
s <- "UnitedKingdom"
t <- str_replace(
  s,
  "([:upper:][:lower:]+)([:upper:][:lower:]+)",
  "\\1 \\2"
)
expect_equal("United Kingdom", t)
```

1610 **6.7 Test for match**

1611 You may want to test if a function's output matches a pattern:

```
#' Get the version, for example '1.0'
get_version <- function() {
  sample(c("1.0", "1.1"), size = 1)
}
```

1612 Using testthat::expect_match gives an unexpected result:

```
expect_match(get_version(), "1\\.[:digit:]")
#> Error: get_version\(\) does not match "1\\.[:digit:]".
#> Actual value: "1\\.0"
```

1613 Take a look at ?testthat::expect_match:

1614 Details

1615 `expect_match()` is a wrapper around `grepl()`. See its documentation for more
 1616 detail about the individual arguments.

1617 Use the base R regex dialect:

```
expect_match(get_version(), "1\\.\\.[:digit:]")
```

1618 6.8 Bigger picture

1619 6.8.1 Develop in packages

- 1620 • Also when ‘just’ doing data analysis
- 1621 • Cleanly read files
- 1622 • Test you regexes

1623 6.8.2 Regex usage outside R

1624 There are plenty of tools that allow to use regular expressions:

- 1625 • `grep`, `egrep`
- 1626 • `sed`
- 1627 • `dir/ls`

1628 6.8.3 Warning



1630 ‘Regex Golf’, from <https://xkcd.com/1313/>

1631 Don’t overthink your regexes! If all tests pass, you did a good job

1632 6.9 Resources

- 1633 • RStudio cheatsheets, including the ‘Work with Strings Cheatsheet’

1634 Chapter 7

1635 Programming in the *tidyverse*



Every use case is ridiculous
until it happens to you.

1636

1637 Load the packages for the day.

```
library(tidyverse)
library(rlang)
```

1638 A function to look at errors.

```
try_this <- function(ex) {
  tryCatch(
    expr = {
      ex
    },
    error = function(e) {
      print(glue::glue(as.character(e), "\n"))
    }
  )
}
```

```
)
}
```

1639 7.1 An explanation of the problem

1640 7.1.1 What the issue is

1641 Get some data from *Phylacine*, and attempt to select or filter.

```
# read in phylacine data
data = read_csv("data/phylacine_traits.csv")
```

```
# regular filtering
small_mammals = data %>%
  filter(Mass.g < 1000)

# filtering on a string
small_mammals_too = data %>%
  filter("Mass.g" < 1000)
```

1642 Examine `small_mammals` and `small_mammals_too` to check whether they are as
1643 expected.

```
# count rows
map_int(list(sm_1 = small_mammals, sm2 = small_mammals_too),
        nrow)
#> sm_1 sm2
#> 4381 0
```

1644 The difference in the number of rows is because `dplyr::filter` could not understand
1645 the string `"Mass.g"` as a variable in the dataframe.

1646 This is because the tidyverse, through its `tidyselect` package, makes a distinction
1647 between `"Mass.g"`, and `Mass.g`.

1648 A better explanation of (some of) the theory behind this can be found here: Programming
1649 with `dplyr`.

1650 The same issue arises with functions such as `dplyr::summarise` and `dplyr::group_by`.

```
# summarise using an unquoted variable
summarise(data,
           mean_mass = mean(Mass.g))
#> # A tibble: 1 x 1
#>   mean_mass
#>   <dbl>
#> 1 156882.
```

```
# this will print a warning
summarise(data,
```



```

      mean_mass = mean("Mass.g"))
#> Warning in mean.default("Mass.g"): argument is not numeric or logical: returning
#> NA
#> # A tibble: 1 x 1
#>   mean_mass
#>   <dbl>
#> 1      NA

```

1651 7.1.2 Why the issue is a problem

1652 Consider an analysis pipeline as follows.

1653 data %>% select variables %>% summarise by groups

```

data %>%
  select(Mass.g, Diet.Plant, Order.1.2) %>%
  group_by(Order.1.2) %>%
  summarise_all(.funs = mean) %>%
  head()
#> # A tibble: 6 x 3
#>   Order.1.2      Mass.g Diet.Plant
#>   <chr>      <dbl>    <dbl>
#> 1 Afrosoricida    306.      0.947
#> 2 Carnivora    47905.     14.1
#> 3 Cetartiodactyla 1854811.    76.2
#> 4 Chiroptera     49.1     27.3
#> 5 Cingulata    235529.    43.0
#> 6 Dasyuromorphia   748.      1.09

```

1654 Now consider that this analysis pipeline is repeated many times in your document. Con-

1655 sider also that a well intentioned person has renamed the dataframe columns.

```

data <- data %>%
  `colnames<-` (str_replace_all(colnames(data), "\\.", "_") %>%
    str_to_lower %>%
    str_remove("_1_2"))

```

1656 The group-summarise code above will no longer work.

```

try_this(ex =

  data %>%
    select(Mass.g, Diet.Plant, Order.1.2) %>%
    group_by(Order.1.2) %>%
    summarise_all(.funs = mean) %>%
    head()

)
#> Error: Can't subset columns that don't exist.
#> x Column `Mass.g` doesn't exist.

```

1657 This illustrates the problem in part: when the columns to be operated upon are *unknown*
 1658 *to the programmer*, much of basic tidyverse code cannot be generalised to be used with
 1659 any dataframe.

1660 7.1.3 Passing variables as strings is (also) an issue

1661 The variables to be operated on could be given as strings, perhaps as the argument to a
 1662 function, or as a global variable. This way, a single global vector could contain the group-
 1663 ing variables for all further summarise procedures.

1664 This runs into the problem identified earlier.

```
# choose some variables
vars_to_select = c("Mass.g", "Diet.Plant")
vars_to_group = c("Order.1.2")

# attempt to select and summarise on group
# the tidyverse will not be pleased
try_this(ex =

  data %>%
    select(vars_to_select) %>% # this works with a warning
    group_by(vars_to_group) %>%
    summarise(mean_mass = mean(Mass.g),
              mean_plant = mean(Diet.Plant))
)
#> Error: Can't subset columns that don't exist.
#> x Columns `Mass.g` and `Diet.Plant` don't exist.
```

1665 In the case of a standard filter %>% group %>% summarise pipeline, the function's
 1666 operations are evident. It must filter a dataframe based on a/some column(s), and then
 1667 summarise by groups. The filter to be applied, the variables to group by, and the variables
 1668 to be summarised should be passed as function arguments — just how this is to be done
 1669 is not immediately obvious.

1670 7.2 Flexible selection is easy

1671 Selection often precedes data operations, but is not part of the pipeline dealt with further.

1672 This is because `dplyr::select` appears to work on both quoted and unquoted variables,
 1673 but in general some useful select helpers such as `dplyr::all_of` should be used.
 1674 These straightforward helper functions significantly expand `select`'s flexibility and
 1675 ease of use, and are not covered here. See the `select` help for more information.

1676 7.3 A first attempt at a flexible function

1677 The attempt below to write such a function, which gives the mean and confidence inter-
1678 vals of groups is likely to fail.

```
# define a ci function
ci <- function(x, ci = 95) {
  qnorm(1 - (1 - ci / 100)/2) * sd(x, na.rm = TRUE) / sqrt(length(x))
}

custom_summary <- function(data, filters, grouping_vars, summary_vars) {

  data %>%
    filter(filters) %>%
    group_by(grouping_vars) %>%
    summarise(mean = mean(summary_vars),
              ci = ci(summary_vars))

}
```

1679 7.3.1 Failure of the first attempt

```
# this is going to fail, so look at the error message
try_this(ex = custom_summary(data,
  filters = list(mass_g > 1000),
  grouping_vars = list(order, family),
  summary_vars = list(diet_plant))
)

#> Error: Problem with `filter()` input `..1`.
#> x object 'mass_g' not found
#> i Input `..1` is `filters`.
```

1680 This function initially failed because `filter` could not find `mass_g` in the dataframe. This
1681 is because `mass_g` is treated as an independent R object, while the function should in-
1682 stead treat it as a variable in a dataframe.

1683 The difference between so-called data and environment variables is explained better at
1684 the `rlang` and `tidyeval` websites and tutorials linked at the end of this chapter. It is this
1685 difference that prevents `filter` from correctly interpreting `mass_g`.

1686 7.3.2 Passing arguments as strings doesn't help

1687 The example below tries to get `filter` to work. What could be tried? One option is to
1688 attempt passing the filtering process as a string argument, i.e., `"mass_g > 1000"`.

```
# it doesn't matter whether filters is a vector or list
try_this(ex = custom_summary(data,
  filters = c("mass_g > 1000"),
  grouping_vars = list(order, family),
```

```

summary_vars = list(diet_plant))
)
#> Error: Problem with `filter()` input `..1`.
#> x Input `..1` must be a logical vector, not a character.
#> i Input `..1` is `filters`.

```

1689 While this doesn't work, it is on the right track, which is that the `filters` argument needs
 1690 some extra work beyond changing the type.

1691 7.3.3 None of the other arguments will be successful

1692 `filter` was the first failure, after which it stopped further evaluation, but none of the
 1693 steps of the custom function would have worked, for the same reason `filter` would not have
 1694 worked: all the arguments need some work before they can be passed to their respective
 1695 functions.

1696 7.4 Flexible filtering in a function

1697 The first thing to try is to change how `filter` uses the argument passed to it. Here, the
 1698 argument `filters` is passed as a character vector, and is set by default to filter out mam-
 1699 mals with masses below 1 kg.

1700 The argument could be passed as a list, but the `rlang::parse_exprs` function works on
 1701 vectors, not lists. The conversion between them is trivial for single level lists with atomic
 1702 types (`purrr::as_vector`).

1703 A brief detour: Expressions in R

1704 A full explanation of R works under the hood would take a very long time. A working
 1705 knowledge of how this working can be exploited is usually sufficient to use most of R's
 1706 functionality.

1707 R expressions are one such. They represent a promise of R code, but without being evalu-
 1708 ated. Any string can be parsed (interpreted) as an R expression.

1709 What does `rlang::parse_exprs` do? It interprets a string as an R command. This ex-
 1710 pression can then be evaluated later. Consider the following, where `a` is assigned the nu-
 1711 meric value 3.

```

# a is assigned
a = 3

# parsed but not evaluated
rlang::parse_expr("a + 3")
#> a + 3

# evaluated

```

```
rlang::parse_expr("a + 3") %>% eval
#> [1] 6
```

1712 Here, `a + 3` was converted to an expression in the second command, and only evaluated
1713 in the third.

1714 **Unquoting with !!!**

1715 R expressions underlie R code. Their evaluation can be forced inside another function us-
1716 ing the special operators `!!` and `!!!`, for single and multiple R expressions respectively.

1717 **7.4.1 Flexible filtering using expressions**

1718 Consider the case where mammals below 1 kg body mass are to be excluded. The `dplyr`
1719 code would look like this:

```
1720 filter(data, mass_g > 1000)
```

1721 This fixes both the variable to be filtered by, as well as the cut-off value. This can be made
1722 flexible for a custom function that allows any kind of filtering.

```
custom_summary = function(data,
                           filters = c("mass_g > 1000")) {

  # THIS IS THE IMPORTANT BIT
  filters = rlang::parse_exprs(filters)

  data %>%
    filter(!!!filters)
}
```

1723 Try this function with single and multiple filters.

```
# mammals above a kilo
custom_summary(data,
               filters = c("mass_g > 1000")) %>%

  select(binomial, mass_g) %>%
  head()
#> # A tibble: 6 x 2
#>   binomial      mass_g
#>   <chr>      <dbl>
#> 1 Acerodon_jubatus    1075
#> 2 Acinonyx_jubatus   46700
#> 3 Acratocnus_odontrigonus 22990
#> 4 Acratocnus_ye      21310
#> 5 Addax_nasomaculatus  70000.
#> 6 Aepyceros_melampus   52500.
```

```

# mammals between 250 and 500 g and which are mostly carnivorous
custom_summary(data,
  filters = c("between(mass_g, 250, 500)",
    "diet_plant < 10")) %>%

  select(binomial, mass_g, diet_plant) %>%

  head()
#> # A tibble: 6 x 3
#>   binomial      mass_g diet_plant
#>   <chr>      <dbl>     <dbl>
#> 1 Chrysospalax_trevelyani 426.         0
#> 2 Cyclopes_didactylus    330.         0
#> 3 Desmana_moschata      383.         0
#> 4 Dologale_dybowskii    350.         0
#> 5 Hydromys_chrysogaster  480.         0
#> 6 Hyosciurus_heinrichi   296.         0

```

1724 The function `filter` correctly processes the string passed to filter the data.

1725 7.5 Flexible grouping in a function

1726 Just as the exact filtering approach can be controlled from a single string vector in the
 1727 example above, the grouping variables can also be stored and passed as arguments using
 1728 the `...` (dots) argument. Dots are a convenient way of referring to all unnamed argu-
 1729 ments of a function. Here, they are used to accept the grouping variables.

1730 7.5.1 Using `...` and ‘forwarding’

```

custom_summary = function(data,
  filters = c("mass_g > 1000"),
  ...) {
  # deal with groups
  grouping_vars = rlang::enquos(...)

  data %>%
    filter(!!!rlang::parse_exprs(filters)) %>%

    # this is the important bit
    group_by(!!!grouping_vars)
}

```

1731 Try the function again, and check the grouping variables.

```

custom_summary(data,
  filters = c("mass_g > 1000"),
  order, family) %>%

```

```

group_vars()
#> [1] "order" "family"

```

1732 7.5.2 Passing grouping variables as strings

1733 In the previous example, the grouping variables were passed as unquoted variables, then
 1734 enquo-ted and parsed, after which they were applied. An alternative way of passing argu-
 1735 ments to a function is as a string vector, i.e, `grouping_vars = c("var_a", "var_b")`.

1736 This can be done by interpreting the string vector as R symbols using `rlang::syms`. It
 1737 could also be done by treating them as a full expression using the previously covered
 1738 `rlang::parse_exprs`. However, both methods must use an unquoting-splice (!!!), i.e.,
 1739 force the evaluation of a list of R expressions.

1740 7.5.3 Using `rlang::syms`

```

custom_summary = function(data,
                           filters = c("mass_g > 1000"),
                           grouping_vars) {
  # deal with groups
  grouping_vars = rlang::syms(grouping_vars)

  data %>%
    filter(!!!rlang::parse_exprs(filters)) %>%

    # this is the important bit
    group_by(!!!grouping_vars)
}

custom_summary(data,
               filters = c("mass_g > 1000"),
               grouping_vars = c("order", "family")
               ) %>%

  summarise(mean_mass = mean(mass_g)) %>%
  head()
#> # A tibble: 6 x 3
#> # Groups:   order [2]
#>   order      family    mean_mass
#>   <chr>     <chr>         <dbl>
#> 1 Afrosoricida Tenrecidae    13220
#> 2 Carnivora    Ailuridae      4900
#> 3 Carnivora    Canidae     10502.
#> 4 Carnivora    Eupleridae    5853.
#> 5 Carnivora    Felidae     52801.
#> 6 Carnivora    Herpestidae   2334.

```

1741 7.5.4 Using `rlang::parse_exprs`

```

custom_summary = function(data,
                           filters = c("mass_g > 1000"),
                           grouping_vars) {
  # deal with groups
  grouping_vars = rlang::parse_exprs(grouping_vars)

  data %>%
    filter(!!!rlang::parse_exprs(filters)) %>%

    # this is the important bit
    group_by(!!!grouping_vars)
}

custom_summary(data,
               filters = c("mass_g > 1000"),
               grouping_vars = c("family", "iucn_status")
               ) %>%

  summarise(mean_mass = mean(mass_g)) %>%
  head()
#> # A tibble: 6 x 3
#> # Groups:   family [5]
#>   family      iucn_status mean_mass
#>   <chr>      <chr>      <dbl>
#> 1 Ailuridae    EN           4900
#> 2 Anomaluridae DD           1770
#> 3 Antilocapridae EP          40503.
#> 4 Antilocapridae LC          46083.
#> 5 Aotidae      LC           1060
#> 6 Aplodontiidae LC           1004

```

1742 7.6 Flexible summarising in a function

1743 Summarising using string expressions has been around in the tidyverse for a very long
 1744 time, and `summarise_at` is a function most users are familiar with, along with its variants
 1745 `summarise_if`, `summarise_all`

1746 7.6.1 Using `dplyr::summarise_at`

1747 Simply pass a string vector to the `.vars` argument of `summarise_at`, while passing a list,
 1748 named or otherwise, of functions to the `.funs` argument.

```

custom_summary = function(data,
                           filters = c("mass_g > 1000"),
                           grouping_vars,

```



```

summary_vars,
summary_funs) {

  # deal with groups
  grouping_vars = rlang::parse_exprs(grouping_vars)

  data %>%
    filter(!!!parse_exprs(filters)) %>%
    group_by(!!!grouping_vars) %>%

    # important bit
    summarise_at(.vars = summary_vars,
                 .funs = summary_funs)
}

custom_summary(data,
               grouping_vars = c("order", "family"),
               summary_vars = "mass_g",
               summary_funs = list(this_is_a_mean = mean, sd))

#> # A tibble: 113 x 4
#> # Groups:   order [24]
#>   order      family    this_is_a_mean    fn1
#>   <chr>      <chr>          <dbl>   <dbl>
#> 1 Afrosoricida Tenrecidae      13220      NA
#> 2 Carnivora    Ailuridae       4900      NA
#> 3 Carnivora    Canidae      10502.  11618.
#> 4 Carnivora    Eupleridae     5853.   6234.
#> 5 Carnivora    Felidae      52801.  88201.
#> 6 Carnivora    Herpestidae   2334.   937.
#> # ... with 107 more rows

```

1749 7.6.2 Using the across argument for summary variables

1750 dplyr 1.0.0 had summarise_* superseded by the across argument to summarise.
 1751 This works somewhat differently. The example below shows how the mean of a trait of
 1752 mammal groups can be found.

1753 This example makes use of embracing using {{ }}, where the double curly braces indi-
 1754 cate a promise, i.e., an expectation that such a variable will exist in the function environ-
 1755 ment.

```

custom_summary = function(data,
                          filters = c("mass_g > 1000"),
                          grouping_vars,
                          summary_vars) {

  # deal with groups
  grouping_vars = parse_exprs(grouping_vars)

```

```

data %>%
  filter(!!!parse_exprs(filters)) %>%
  group_by(!!!grouping_vars) %>%

  # important bit
  summarise(across({{ summary_vars }},
    ~ mean(.)))
}

custom_summary(data,
  grouping_vars = c("order", "family"),
  summary_vars = c(mass_g, diet_plant)) %>%

  head()
#> # A tibble: 6 x 4
#> # Groups:   order [2]
#>   order      family      mass_g diet_plant
#>   <chr>      <chr>      <dbl>    <dbl>
#> 1 Afrosoricida Tenrecidae 13220      4
#> 2 Carnivora    Ailuridae   4900     80
#> 3 Carnivora    Canidae   10502.    15.0
#> 4 Carnivora    Eupleridae  5853.     2.67
#> 5 Carnivora    Felidae   52801.    0.348
#> 6 Carnivora    Herpestidae 2334.     9.86

```

1756 `across` also accepts multiple functions just as `summarise_did`. This works as follows.

```

# mean and sd
data %>%
  group_by(order, family) %>%
  summarise(across(c(mass_g, diet_plant),
    list(~ mean(.),
      ~ sd(.))
  )
  ) %>%

  head()
#> # A tibble: 6 x 6
#> # Groups:   order [2]
#>   order      family      mass_g_1 mass_g_2 diet_plant_1 diet_plant_2
#>   <chr>      <chr>      <dbl>    <dbl>    <dbl>    <dbl>
#> 1 Afrosoricida Chrysochloridae   60.7     86.6      0      0
#> 2 Afrosoricida Tenrecidae    449.    2197.     1.5     6.83
#> 3 Carnivora    Ailuridae    4900      NA      80     NA
#> 4 Carnivora    Canidae   10268.  11568.    16.0    18.0
#> 5 Carnivora    Eupleridae   3777.   5364.     4.6     6.72
#> 6 Carnivora    Felidae   52801.  88201.    0.348    2.36

```

1757 **7.6.3 Summarise multiple variables using . . .**

1758 Here, the unquoted and unnamed variables passed to the function are captured by . . .
 1759 and enquos-ed, i.e, their evaluation is delayed. Then the variables are forcibly evaluated
 1760 within the mean function, and this expression is captured using expr. Since there are
 1761 multiple variables to summarise, these expressions are stored as a list.

```

custom_summary = function(data,
                           grouping_vars,
                           filters,
                           ...) {
  # deal with groups
  grouping_vars = rlang::parse_exprs(grouping_vars)

  # deal with summary variables
  summary_vars = rlang::enquos(...)

  # apply the summary function to the variables
  summary_vars <- purrr::map(summary_vars, function(var) {
    rlang::expr(mean(!!var, na.rm = TRUE))
  })

  data %>%
    filter(!!!rlang::parse_exprs(filters)) %>%
    group_by(!!!grouping_vars) %>%

    # important bit
    summarise(!!!summary_vars)
}

custom_summary(data,
               grouping_vars = c("order", "family"),
               filters = "mass_g > 10",
               mass_g, diet_plant) %>%

  head()
#> # A tibble: 6 x 4
#> # Groups:   order [2]
#>   order      family      `mean(mass_g, na.rm = T` `mean(diet_plant, na.rm = ~
#>   <chr>      <chr>      <dbl>      <dbl>
#> 1 Afrosorici~ Chrysochlori~      60.7      0
#> 2 Afrosorici~ Tenrecidae      597.      2
#> 3 Carnivora   Ailuridae     4900     80
#> 4 Carnivora   Canidae     10268.    16.0
#> 5 Carnivora   Eupleridae    3777.     4.6
#> 6 Carnivora   Felidae     52801.    0.348

```

1762 **expr and enqu**

1763 `expr` and `enquo` are essentially the same, defusing/quoting (delaying evaluation) of R
 1764 code. `expr` works on expressions supplied by the primary user, while `enquo` works on
 1765 arguments passed to a function. When in doubt, ask whether the expression to be quoted
 1766 has entered the function environment as an argument. If yes, use `enquo`, and if not `expr`.
 1767 The plural forms `enquos` and `exprs` exist for multiple arguments.

1768 **7.6.3.1 Correct the names of summary variables**

1769 The example above returns summary variables that are not assigned a name. The `enquos`
 1770 function can assign the name from the variable names, so `mean(mass_g)` is returned as
 1771 `mass_g`. Since it is useful to add a tag to make clear what the summary variable is (mean,
 1772 variance etc.) an extra `glue` step is added to assign informative names to the summary
 1773 variables.

```

custom_summary = function(data,
                           grouping_vars,
                           filters,
                           ...) {
  # deal with groups
  grouping_vars = rlang::parse_exprs(grouping_vars)

  # deal with summary variables
  summary_vars = rlang::enquos(..., .named = TRUE)

  # apply the summary function to the variables
  summary_vars <- purrr::map(summary_vars, function(var) {
    rlang::expr(mean(!!var, na.rm = TRUE))
  })

  # add a prefix to the summary variables
  names(summary_vars) <- glue::glue('mean_{names(summary_vars)}')

  data %>%
    filter(!!!rlang::parse_exprs(filters)) %>%
    group_by(!!!grouping_vars) %>%

    # important bit
    summarise(!!!summary_vars)
}

custom_summary(data,
               grouping_vars = c("order", "family"),
               filters = "mass_g > 10",
               mass_g, diet_plant) %>%

```

```

head()
#> # A tibble: 6 x 4
#> # Groups:   order [2]
#>   order      family      mean_mass_g mean_diet_plant
#>   <chr>      <chr>      <dbl>      <dbl>
#> 1 Afrosoricida Chrysochloridae    60.7         0
#> 2 Afrosoricida Tenrecidae      597.         2
#> 3 Carnivora    Ailuridae      4900        80
#> 4 Carnivora    Canidae     10268.       16.0
#> 5 Carnivora    Eupleridae    3777.         4.6
#> 6 Carnivora    Felidae     52801.       0.348

```

1774 7.6.4 Summarise with multiple functions

1775 The final step is to pass multiple summary functions to the summary variables. Unlike
 1776 the earlier example using `summarise(across(vars, funs))`, the goal here is to apply
 1777 one function to each variable.

1778 This is done by passing the functions and the variables on which they should operate as
 1779 strings, and using string interpolation via `glue` to construct a coherent R expression. This
 1780 expression is then named and evaluated.

```

custom_summary = function(data,
                           grouping_vars,
                           filters,
                           functions,
                           summary_vars) {
  # deal with groups
  grouping_vars = parse_exprs(grouping_vars)

  # deal with summary variables
  # summary_vars = # enquo(..., .named = TRUE)

  # apply the summary function to the variables
  summary_exprs <- parse_exprs(glue::glue('{functions}({summary_vars}, na.rm = TRUE)'))

  # add a prefix to the summary variables
  names(summary_exprs) <- glue::glue('{functions}_{summary_vars}')

  data %>%
    filter(!!!parse_exprs(filters)) %>%
    group_by(!!!grouping_vars) %>%

    # important bit
    summarise(!!!summary_exprs)
}

```

```

custom_summary(data,
  grouping_vars = c("order", "family"),
  filters = "mass_g > 10",
  functions = c("mean", "var"),
  summary_vars = c("mass_g", "diet_plant")) %>%

  head()
#> # A tibble: 6 x 4
#> # Groups:   order [2]
#>   order      family    mean_mass_g var_diet_plant
#>   <chr>      <chr>      <dbl>      <dbl>
#> 1 Afrosoricida Chrysochloridae    60.7         0
#> 2 Afrosoricida Tenrecidae      597.        61.8
#> 3 Carnivora    Ailuridae      4900         NA
#> 4 Carnivora    Canidae     10268.        325.
#> 5 Carnivora    Eupleridae     3777.         45.2
#> 6 Carnivora    Felidae     52801.         5.57

```

7.7 Further resources

1781

1782

1783

1784

1785

- dplyr: <https://dplyr.tidyverse.org/index.html>
- Tidy evaluation: Superseded and archived, but still useful <https://tidyeval.tidyverse.org/>
- rlang: <https://rlang.r-lib.org/>

1786 Chapter 8

1787 Developing R packages

1788 Raphael Scherrer (thanks to Pedro Neves for guiding me through these steps)



Every use case is ridiculous
until it happens to you.

1789

1790 By now you know what R packages are and you have been using many of them, some of
1791 them part of the tidyverse and others not. R packages are modules, or coherent libraries of
1792 functions, designed at specific sets of tasks. Packages, or libraries, are common to many
1793 programming languages, the philosophy behind them being: pick only the tools you need
1794 for your task, without having to download all the possible toolboxes. Currently CRAN (the
1795 Comprehensive R Archive Network) is host to more than 16,000 packages ([link](#)), and that
1796 is not counting R packages hosted by other platforms such as GitHub, Bioconductor or
1797 rOpenSci. This is what makes R such a powerful and popular language. Why there are
1798 so many packages is because *anyone* can write their own package and make it available to
1799 others, so the growth of the R universe is very much community-driven. Here we will show
1800 you how to write your own package. Most of the content of this tutorial follows Hadley
1801 Wickham's exhaustive book on R packages.

1802 8.1 Why writing packages?

1803 You may very well have written analysis pipelines in R for various projects and never felt
1804 the need to make packages for them. So why bother? you may ask. The main reasons are:

- 1805 • *deployment*: packages make it easier for people to use your code
- 1806 • *reproducibility*: packages can be a convenient way to make your study fully repro-
1807 ducible
- 1808 • *consistency*: there is a common set of rules on how packages should be organized,
1809 which forces you to make your code understandable to everyone
- 1810 • *security*: the common conventions around package syntax make it possible for
1811 third-party tools to check your code for bugs or style, which also means you can
1812 trust packages hosted at some platforms when you know they run these tests, for
1813 example

1814 8.2 Hands-on workflow

1815 8.2.1 Primer: what is an RStudio project?

1816 An RStudio project is a virtual context associated with a specific working directory on your
1817 computer. A project is the recommended unit of work for a given analysis. This is because
1818 it keeps track of the R workspace and history for that analysis, together with the working
1819 directory (meaning you never have to use `setwd` anymore). A project has the extension
1820 `.Rproj`. See this page for more information. As we shall see, developing a package requires
1821 creating a project for it.

1822 8.2.2 Create a project for your package

1823 In RStudio, click on File, then New Project. There, you have the option to create a new
1824 package. This will create all the files that are needed, in particular a DESCRIPTION, a
1825 NAMESPACE, a `.Rbuildignore`, and a `man/` and `R/` folders. Use the `.Rproj` file to develop
1826 the package (launching it will open RStudio and place you in the right directory). It is
1827 possible to create an R package by assembling all those files together by yourself, but
1828 RStudio really makes it painless.

1829 8.2.3 Link to GitHub?

1830 At this stage you may want to host your package on an online version control platform
1831 such as GitHub. One way to do this is the following. Assuming that git is already installed
1832 on your machine and linked to your GitHub account, you need to:

- 1833 1. Create a project for your package locally (the step above)
- 1834 2. Create an empty repository on GitHub for your package
- 1835 3. Initialize git in the local copy by running `git init` from within
- 1836 4. Stage and commit (`git add .` and `git commit -m "some commit message"`)
- 1837 5. Link the local copy to the remote one with `git remote add origin`
1838 `https://github.com/username/reponame`

1839 6. Push using `git push -u origin master`

1840 You should be all set. Useful links include this page, this one and also the instructions
1841 given by GitHub upon creation of an empty online repository.

1842 8.2.4 Write your functions

1843 A package is nothing much more than a convenient collection of functions that one may
1844 want to use repeatedly. Here we assume that you are comfortable with writing R functions.
1845 Prefer saving each function as its own R script (.R) and save them in the dedicated R/
1846 folder. Here is an example function that repeats multiple elements, multiple times and
1847 returns a vector of those.

```
mrep <- function(x, n) {  
  
  assertthat::are_equal(length(x), length(n)) # security check  
  purrr::reduce(purrr::map2(x, n, ~ rep(.x, .y)), c)  
  
}
```

1848 We can use this function, for example, to repeat the number 1 once, number 2 twice and
1849 number 3 three times:

```
mrep(seq(3), seq(3))  
#> [1] 1 2 2 3 3 3
```

1850 *Note* that when calling functions from other packages (here `purrr` and `assertthat`) we
1851 do not use `library` or `require`, as this would make all the functions of these packages
1852 available. Instead we use the namespace of the respective package, separated from the
1853 function name with a `::`. Although a package that uses `library` will typically build just
1854 fine, it is considered bad practice and will not pass CRAN's requirements, which are im-
1855 plemented in the R CMD CHECK command (more on this later).

1856 8.2.5 Tests

1857 Do you want to go test-driven? Then write your tests first, and follow those guidelines.
1858 Although tests are out of the scope of this tutorial, they are a vital part of package devel-
1859 opment, so we highly recommend this read as your next step to go further.

1860 8.2.6 Document your functions

1861 The documentation of a function is what shows up when you type `?function-name` for
1862 example (e.g. `?purrr::reduce`). When writing your package, you must provide a doc-
1863 umentation for each of your functions so your user knows what the function does, what
1864 arguments it takes, what it returns and has examples of the function being used. Each
1865 function documentation goes in its own .Rd file, stored in the `man/` folder.

1866 `roxygen2` is an R package that makes documentation very easy. It allows you to write
1867 the documentation as a header of a function's R script, and save this header into its own

1868 .Rd file in man/. All the lines that go into the documentation must start with the special
 1869 comment characters `#'`. If we take our previous example:

```
#' Repeat multiple things multiple times
#'
#' A function to repeat multiple things multiple times.
#'
#' @param x A vector of things
#' @param n A vector of numbers times each thing must be repeated
#'
#' @details The function can take a vector of any atomic type
#'
#' @return A vector of the same type as `x`
#'
#' @examples
#'
#' mrep(seq(3), seq(3))
#'
#' @export

mrep <- function(x, n) {

  assertthat::are_equal(length(x), length(n)) # security check
  purrr::reduce(purrr::map2(x, n, ~ rep(.x, .y)), c)

}
```

1870 Here, everything starting with `#'` will be interpreted by `roxygen2` as part of the documen-
 1871 tation. Different fields can be supplied:

- 1872 • The first line is the title of the documentation page
- 1873 • The second line is the description
- 1874 • `@param` goes for each of the parameters, with their description
- 1875 • `@details` if you want to be more specific on what happens backstage
- 1876 • `@return` tells the user what the function returns
- 1877 • `@examples` shows some use-cases
- 1878 • `@export` indicates that this function can be called explicitly by the user (as opposed
- 1879 to an internal function of the package that is only meant to be used by other func-
- 1880 tions of the package)

1881 Other fields such as `@note` can be specified, but these are the main ones. A package with
 1882 incomplete documentation will build fine, but again this will not pass R CMD CHECK for
 1883 CRAN's requirements, which require you, for example, to always have examples for ex-
 1884 ported functions.

1885 To effectively produce the documentation, run `roxygen2::roxygenize()` (or `roxy-`
 1886 `genise`) from within the working directory of the package. `roxygen2` may not update
 1887 the `NAMESPACE` file if it has not been created by `roxygen2` in the first place, so you may

1888 have to erase `NAMESPACE` before running `roxygenize()` (then it will automatically
1889 create a new `NAMESPACE`). We do not describe here what the `NAMESPACE` is, as it is
1890 a bit too advanced for this tutorial, just remember that you may have to erase it before
1891 documenting if you see a warning.

1892 8.2.7 Build the package

1893 Once some functions are added and their documentation is ready, the package should
1894 be able to build. Use the Install and Restart button under the Build tab in RStudio for
1895 that. Your package is now installed and loaded. Alternatively you can build your pack-
1896 age from the command line by running `R CMD INSTALL`. If your package is on GitHub
1897 (or another remote server), you can also build it with `devtools`, for example with `dev-`
1898 `tools::install_github("username/reponame")`.

1899 8.3 Write a vignette

1900 A vignette is a more user-oriented overview of your package. In contrast to the individual
1901 documentation of each function, the vignette takes the user for a tour of the package to
1902 show use-cases of the functions *in context*.

1903 A vignette is written in Rmarkdown. The Rmarkdown language is out of the scope of this
1904 tutorial, but is a great way to combine textual information (it inherits from markdown)
1905 with embedded chunks of R code and their output (this tutorial is written in Rmarkdown).
1906 See this link, or this cheatsheet, or inspire yourself from the source code of this tutorial
1907 to get more familiar with Rmarkdown.

1908 We use the `usethis` package to set up everything we need to get our vignette ready. Run-
1909 ning `usethis::use_vignette` will create a `vignettes/` folder with the vignette `.Rmd`
1910 file in it, that you can then edit.

1911 The vignette can be rendered in multiple output formats,, such as an HTML web page or
1912 a LaTeX-looking PDF. RStudio does this through the Knit button, which calls the `knitr`
1913 package in the background. By default, upon creation of the vignette only the HTML out-
1914 put is supported. To change the possible outputs (e.g. allow both HTML and PDF), change
1915 the output part of the header of the `.Rmd` file with:

```
1916 output:  
1917   pdf_document: default  
1918   html_document:  
1919     keep_md: yes
```

1920 Now the drop-down menu of the Knit button will offer the possibility to render the vignette
1921 as PDF as well as HTML.

1922 The Knit button renders a vignette, but does not save it. You could of course save it man-
1923 ually, but `devtools` offers the `build_vignettes` function to automatize this task. Run-
1924 ning it will create two new folders, `doc/` and `Meta/`. The former contains the rendered
1925 vignette, in the first format specified in the output header (so PDF in the above example)

while the latter contains some data used to render that vignette. It is best to not touch those, and stick to editing the vignette file located in the `vignettes/` folder. One exception: one can render a vignette manually with the Knit button and save the rendered output into the `doc/` folder.

Do you want to host the vignette on a web page dedicated to your package, also with an overview of all the functions as well as their documentation? Then `pkgdown` is your friend, but this is out of the scope of this tutorial (yes, the web page for the `pkgdown` package is built with `pkgdown`).

8.4 Update the description

In the top folder of your package is a `DESCRIPTION` file. This contains some important information. Make sure that you update the Title, Author, Maintainer, Description and License fields. The Imports field requires you to supply the names of the dependencies of your package: what packages *need* to be installed for your functions to work? In our example, `mrep` calls functions from `assertthat` and `purrr`, so our Imports field will look something like:

```
Imports:
  assertthat,
  purrr
```

These dependencies will be downloaded and installed automatically upon installation of your package. You can specify version requirements for the packages you load (see Hadley's book). The Suggests field is for packages that are not required but recommended (e.g. `knitr` to build the vignette locally).

Dependencies will be downloaded from CRAN by default. In order to add packages from other platforms, you may have to add some keywords to your `DESCRIPTION` file. For example, the `ggtree` package is hosted by Bioconductor. You can add it with the other packages in Imports, but you need to add "biocViews:" before Imports, e.g.

```
biocViews:
Imports:
  assertthat,
  purrr,
  ggtree
```

A special case of dependencies is operators from other packages, such as the famous pipe (`%>%`) from `magrittr`, because you cannot just write `magrittr::%>%` in your functions. Again, `usethis` is our friend here, and you can run `usethis::use_pipe()` to make the pipe operator fully available to your functions without having to use `library`. (This command will update the `NAMESPACE`.)

As a minor note, you can also use the `DESCRIPTION` file to give extra options to the build of your documentation. For example, to allow `roxygen2` to understand the markdown syntax when rendering the help pages of your functions, use

1965 Roxygen: list(markdown = TRUE)

1966 8.5 Check the package

1967 8.5.1 Good practices

1968 As mentioned before, CRAN has specific requirements that are implemented in the R CMD
1969 CHECK command. Running this command, or clicking on Check within the Build tab, will
1970 run a series of quality controls on your code, and will indicate what does not meet the
1971 requirements. A package is CRAN-compatible if no errors and no warnings are issued
1972 (notes are fine).

1973 Generally, CHECK will make sure all the things we talked about above are done. It will
1974 look at the functions, the documentation, run your examples (and your tests if you have
1975 some) make sure that the vignette renders, and that all dependencies are accessible. If
1976 anything is wrong, it will tell you what.

1977 One thing to keep in mind is that CHECK will run your examples (in the documentation
1978 files), *unless* these are surrounded with `\dontrun{` and `}`. This can be used for examples
1979 that, e.g., would require some specific data that you do not make available with the pack-
1980 age, or just because the example takes too long or is too computation-heavy.

1981 CHECK also dislikes files and folders that are not absolutely necessary to the package.
1982 It will complain if, say, you have a `scripts/` folder with extra draft scripts you used to
1983 develop and try your functions, or a `data/` folder containing some example data. You can
1984 add the names of these folders to `.Rbuildignore` to tell CHECK to ignore those when
1985 checking your package (`.Rbuildignore` works in many respects just like a `.gitignore`
1986 file).

1987 8.5.2 Better practices

1988 If all the above are met, CHECK should be happy and in theory your package should be
1989 CRAN-compatible. Some platforms, such as rOpenSci, have stricter standards, however,
1990 and those requirements come from a good place. We will highlight two things here.

1991 First, rOpenSci will require 100% code coverage in your package. This means that during
1992 the execution of the CHECK command, every single line of code must be run. This is often
1993 impossible to achieve without having tests, and thus strongly encourages test-driven de-
1994 velopment. The `testthat` package can be used to write tests that check for the outcomes
1995 of your functions under different circumstances, or scenarios. See the section on `regex`
1996 for examples. In a package, `test` will be stored in a `tests/` folder, which can be set-up by
1997 our old friend `usethis`, by running `usethis::use_testthat()`. Having tests is always
1998 good!

1999 Second, rOpenSci will also check your coding *style*. In R, it is possible to write the same
2000 code in different ways, for example:

```
library(tidyverse)
```

```

x <- mrep(seq(3), seq(3))
y <- rep(1, 6)
tibble(
  V1 = x,
  V2 = y
)
#> # A tibble: 6 x 2
#>       V1     V2
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     2     1
#> 4     3     1
#> 5     3     1
#> 6     3     1

```

2001 versus

```

x = mrep(seq(3), seq(3))
y = rep(1, 6)
tibble(V1 =x,
       V2 =y
)
#> # A tibble: 6 x 2
#>       V1     V2
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     2     1
#> 4     3     1
#> 5     3     1
#> 6     3     1

```

2002 Both styles will run, and CHECK will not complain. However, `lintr` will. `lintr` is a style
 2003 checker that makes sure that you follow the *tidyverse* recommended style. This style in-
 2004 cludes things such as: no use of `=` as an assignment operator (only use `<-`), always put a
 2005 space after an equal sign or a comma among others. `lintr` will be run on all of your R
 2006 code if you submit your package to rOpenSci. The reason behind using a style checker is
 2007 similar to the basic philosophy of the *tidyverse*: make things follow a *convention*, so that
 2008 pieces of code speak the same language (so to speak, pun intended) and integrate nicely
 2009 with each other.

2010 8.5.3 Even better practices

2011 Git and GitHub (or other version control platforms) are your friends when it comes to de-
 2012 veloping packages or software in general. You may want to check out how to use them.
 2013 One strength of these platforms is that they allow you to give access to third-party plat-

forms to your package, that can be used to quality-control your code. These are known as *continuous integration* tools, Travis CI and AppVeyor being two famous examples. By activating these tools on your repository (hosted, say, on GitHub), these platforms can access your package and remotely run all kinds of things for you: run R CMD CHECK, make sure that the code coverage is 100%, or run `lintr` for you, every time you upload an edited version of your code. This gives you an extra safety net to make sure that your package (or at least the version hosted online and available to people) is always working, and it may even give you a hint if, for example, one dependency of your package breaks (due to errors independent of you). If you want to know more, you can for example check the R package `babette`, which makes use of these tools and is hosted at `rOpenSci`.

8.6 References

- Hadley's book on developing R packages