

# TRES Tidyverse Tutorial

Raphael, Pratik, Theo and Richèl

2020-07-14



# Contents

2	<b>Outline</b>	<b>5</b>
3	About . . . . .	5
4	Schedule . . . . .	5
5	Possible extras . . . . .	6
6	Join . . . . .	6
7	<b>1 Reading files and string manipulation</b>	<b>7</b>
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> . . . . .	20
12	<b>2 Reshaping data tables in the tidyverse, and other things</b>	<b>23</b>
13	2.1 The new data frame: <code>tibble</code> . . . . .	24
14	2.2 The concept of tidy data . . . . .	26
15	2.3 Reshaping with <code>tidyr</code> . . . . .	28
16	2.4 Extra: factors and the <code>forcats</code> package . . . . .	36
17	2.5 External resources . . . . .	41
18	<b>3 Data manipulation with <code>dplyr</code></b>	<b>43</b>
19	3.1 Introduction . . . . .	43
20	3.2 Working with existing variables . . . . .	44
21	3.3 Working with observations . . . . .	52
22	3.4 Making new variables . . . . .	59
23	3.5 Working with multiple tables . . . . .	67
24	<b>4 Working with lists and iteration</b>	<b>73</b>
25	4.1 List columns with <code>tidyr</code> . . . . .	73
26	4.2 Iteration with <code>map</code> . . . . .	76
27	4.3 More <code>map</code> variants . . . . .	81
28	4.4 Combining map variants and tidyverse functions . . . . .	82
29	4.5 A return to map variants . . . . .	84
30	4.6 Other functions for working with lists . . . . .	86
31	4.7 Lists of <code>ggplots</code> with <code>patchwork</code> . . . . .	89

32	<b>5</b>	<b>ggplot2 and the grammar of graphics</b>	<b>91</b>
33	5.1	Introduction . . . . .	92
34	5.2	But first, the data . . . . .	93
35	5.3	Geom layers . . . . .	95
36	5.4	Coordinate-system . . . . .	106
37	5.5	Facetting . . . . .	107
38	5.6	The right format for the dataset . . . . .	110
39	5.7	Plotting as part of a pipeline . . . . .	111
40	5.8	Customization . . . . .	112
41	5.9	Combining plots . . . . .	121
42	5.10	Saving a plot . . . . .	123
43	5.11	High throughput plotting workflow . . . . .	123
44	5.12	Want more? . . . . .	126
45	5.13	References . . . . .	127
46	<b>6</b>	<b>Regular expressions and testthat</b>	<b>129</b>
47	6.1	Introduction . . . . .	130
48	6.2	Testing . . . . .	133
49	6.3	Detect a full match . . . . .	134
50	6.4	Extract a pattern . . . . .	141
51	6.5	Other functions . . . . .	146
52	6.6	Bigger picture . . . . .	147
53	6.7	Regex usage beyond R is common . . . . .	147
54	6.8	Resources . . . . .	148
55	<b>7</b>	<b>Programming in the <i>tidyverse</i></b>	<b>149</b>
56	7.1	An explanation of the problem . . . . .	150
57	7.2	Flexible selection is easy . . . . .	152
58	7.3	A first attempt at a flexible function . . . . .	153
59	7.4	Flexible filtering in a function . . . . .	154
60	7.5	Flexible grouping in a function . . . . .	156
61	7.6	Flexible summarising in a function . . . . .	159
62	7.7	Further resources . . . . .	164

# 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

## 77 Possible extras

- 78 • Reproducibility and package-making (with e.g. usethis)
- 79
- 80 • Embedding C++ code with Rcpp

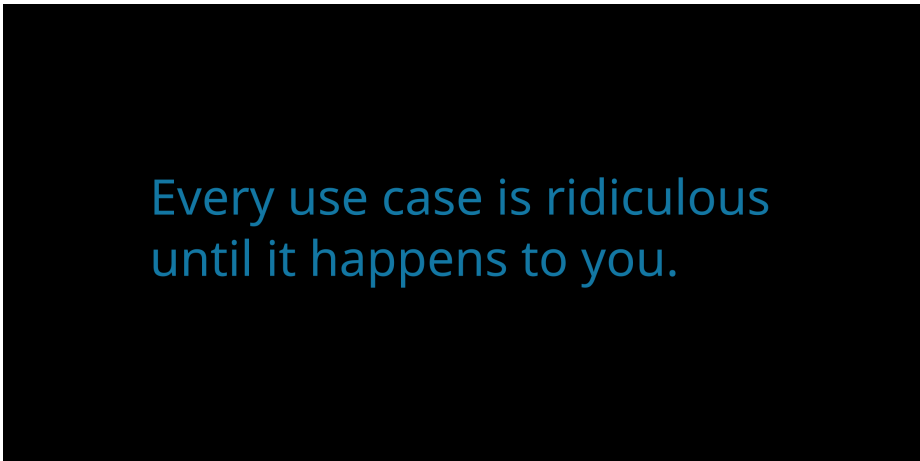
## 81 Join

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

83 \*Tentative dates.

## Chapter 1

# Reading files and string manipulation



Every use case is ridiculous  
until it happens to you.

Load the packages for the day.

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

### 1.1 Data import and export with readr

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

functions to help with reading 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

```

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

## Excel files

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

## 1.2 String manipulation with **stringr**

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

**stringr** functions begin with **str\_**.

### 1.2.1 Putting strings together

Concatenate two strings with **str\_c**, and duplicate strings with **str\_dup**. Flatten a list or 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"

```

151 `str_flatten` is especially useful when displaying the type of an object that  
 152 returns a list 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"

```

### 153 1.2.2 Detecting strings

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

```

158 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

```

159 **str\_locate** locates the search pattern in a string, and returns the start and  
 160 end as a two 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
```

161 Detect whether a string starts or ends with a pattern. Also vectorised. Both  
 162 have a **negate** argument, which returns the negative, i.e., returns FALSE if the  
 163 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  
 165 character vector based on a **str\_detect** predicate. In the example, all elements  
 166 containing “banana” are subset.

167 **str\_which** has the same logic except that it returns the vector position and not  
 168 the elements.

```

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

# returns an integer vector
str_which(c("banana",
            "bananageddon is coming",
            "applegeddon 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"

```

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

188 The return type is a list, with one element per input string. Each element is a  
 189 character matrix, where each row is one possible match, and each column after  
 190 the first (the full 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}[\\-\\/](\\w+)"
#> [[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}[\\-\\/](\\w+)"
#> [[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"

```

### 191 1.2.4 Simpler pattern extraction

192 The full functionality of `str_match_*` can be boiled down to the most com-  
 193 mon use case, extracting one or more full matches of the search pattern using  
 194 `str_extract` and `str_extract_all` respectively.

195 `str_extract` returns a character vector with the same length as the input string  
 196 vector, while `str_extract_all` returns a list, with a character vector whose  
 197 elements are the 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"
```

### 198 1.2.5 Breaking strings apart

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

202 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"
```

203 This can be useful in recovering simulation parameters from a filename, but may  
 204 require 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"  ""

```

205 `str_split_fixed` split the string into as many pieces as specified, and can be  
 206 especially 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"

```

## 207 1.2.6 Replacing string elements

208 `str_replace` is intended to replace the search pattern, and can be co-opted  
 209 into the task of recovering simulation parameters or other data from regularly  
 210 named files. `str_replace_all` works the same way but replaces all matches of  
 211 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 "

```

212 `str_remove` is a wrapper around `str_replace` where the replacement is set to  
 213 `""`. This is not covered here.

214 Having replaced unwanted characters in the filename with spaces, `str_trim`  
 215 offers a way 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"

```

### 216 1.2.7 Subsetting within strings

217 When strings are highly regular, useful data can be extracted from a string using  
 218 `str_sub`. In the date-time example, the year is always represented by the first  
 219 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"

```

220 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"

```

221 Finally, it's also possible to replace characters within a string based on the  
 222 position. This 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"
```

### 223 1.2.8 Padding and truncating strings

224 Strings included in filenames or plots are often of unequal lengths, especially  
 225 when they represent numbers. `str_pad` can pad strings with suitable characters  
 226 to maintain equal 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"
```

227 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."
```

### 228 1.2.9 Stringr aspects not covered here

229 Some `stringr` functions are not covered here. These include:

- 230 • `str_wrap` (of dubious use),
- 231 • `str_interp`, `str_glue*` (better to use `glue`; see below),
- 232 • `str_sort`, `str_order` (used in sorting a character vector),
- 233 • `str_to_case*` (case conversion), and
- 234 • `str_view*` (a graphical view of search pattern matches).
- 235 • `word`, `boundary` etc. The use of `word` is covered below.

236 `stringi`, of which `stringr` is a wrapper, offers a lot more flexibility and control.

## 237 1.3 String interpolation with glue

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

240 `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

```

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

242 The related `glue_data` is even more useful in printing from a dataframe. In  
 243 this example, 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

```

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

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

## 1.4 Strings in ggplot

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

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

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

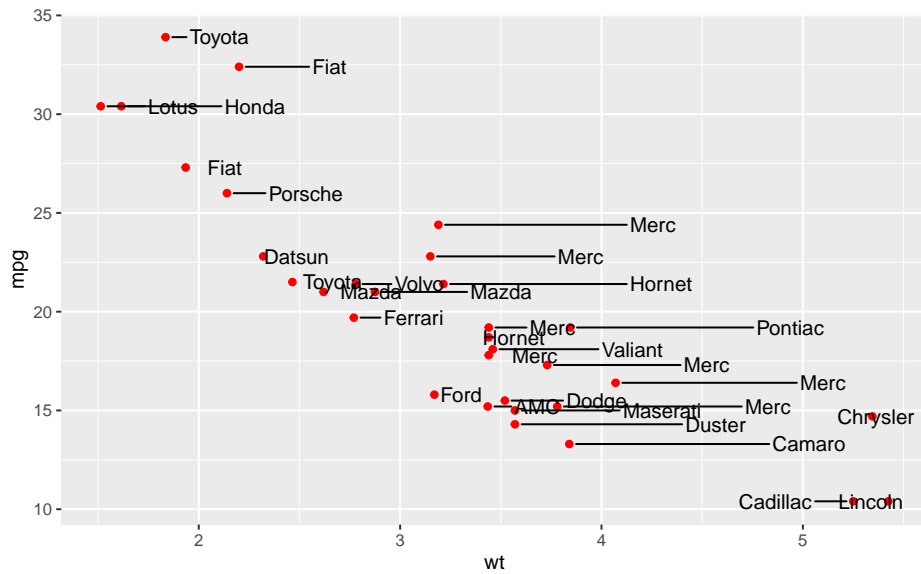
Here, the arguments to `geom_text_repel` are taken both from the `mtcars` data (position), as well as from the car brands extracted using the `stringr::word` (labels), which tries to separate strings based on a regular pattern.

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)
```



264

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

294 As a rule of thumb, try to convert your tables to tibbles whenever you can,  
 295 especially when the original table is *not* a data frame. For example, the prin-  
 296 cipal component analysis function `prcomp` outputs a `matrix` of coordinates in  
 297 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>
```

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

301 So, in the tidyverse we are going to work with tibbles, got it. But what does  
 302 “tidy” mean 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  Isabela body_weight  10.8
#> 2     1  Isabela beak_length   4.94
#> 3     1  Isabela beak_width    1.94
#> 4     2  Isabela body_weight  15.4
#> 5     2  Isabela beak_length   5.02
#> 6     2  Isabela beak_width    2.00
#> # ... with 294 more rows

```

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

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

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

Another common example of wide-or-long dilemma is when dealing with *contingency tables*. This would be our case, for example, if we asked how many observations we have for 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
#> Isabela           50          50          50
#> Santa Cruz         50          50          50

```

A variety of statistical tests can be used on contingency tables such as Fisher's

338 exact test, the chi-square test or the binomial test. Contingency tables are in  
 339 the wide format by construction, but they too can be pivoted to the long format,  
 340 and the tidyverse manipulation tools will expect you to do so. Actually, `tibble`  
 341 knows that very well and does it by default if you convert your `table` into a  
 342 `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.

## 343 2.3 Reshaping with `tidyr`

344 The `tidyr` package implements tools to easily switch between layouts and also  
 345 perform a few other reshaping operations. Old school R users will be famil-  
 346 iar with the `reshape` and `reshape2` packages, of which `tidyr` is the tidyverse  
 347 equivalent. Beware that `tidyr` is about playing with the general *layout* of the  
 348 dataset, while *operations* and *transformations* of the data are within the scope  
 349 of the `dplyr` and `purrr` packages. All these packages work hand-in-hand really  
 350 well, and analysis pipelines usually involve all of them. But today, we focus  
 351 on the first member of this holy trinity, which is often the first one you'll need  
 352 because you will want to reshape your data before doing other things. So, please  
 353 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

```

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

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

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

### 2.3.2 Handling missing values

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
```

371 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
```

372 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
```

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

375 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
```

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

But note that the `day`, `month` and `year` columns are now of class `character` and not `integer` anymore. This is because they result from the splitting of `date`, which itself was a `character` column.

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
```

### 2.3.4 Expanding tables using combinations

Instead of getting rid of rows with NAs, we may want to add rows with NAs, for example, 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
```

398 We could generate a tibble with all combinations of island, morphometric and  
399 year using `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
```

400 If we already have a tibble to work from that contains the variables to combine,  
401 we can 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
```

402 As you can see, we get all the combinations of the variables of interest, even  
403 those that are missing. But sometimes you might be interested in variables  
404 that are *nested* within each other and not *crossed*. For example, say we have  
405 measured birds at different locations 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
```

Of course, if sites A and B are on Isabela, they cannot be on Santa Cruz, where we have sites C and D instead. It would not make sense to **expand** assuming that **island** and **site** are 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
```

But now the missing data for Santa Cruz in 2020 are not accounted for because **expand** thinks the **year** is also nested within island. To get back the missing combination, we use **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
```

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

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

418 which nicely keeps the rest of the columns in the tibble and just adds the missing  
419 combinations.

### 420 2.3.5 Nesting

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

### 425 2.3.6 What else can be tidied up?

#### 426 2.3.6.1 Model output with broom

427 Check out the `broom` package and its `tidy` function to tidy up messy linear  
428 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 6.1e-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
```

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

### 432 2.3.6.2 Graphs with **tidygraph**

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

### 443 2.3.6.3 Trees with **tidytree**

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

## 451 2.4 Extra: factors and the **forcats** package

```
library(forcats)
```

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

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

456 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
```

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

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

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

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

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

469 But this is a tidyverse tutorial, so I would like to introduce here the package  
 470 **forcats**, which offers tools to manipulate factors. First of all, most tools from  
 471 **stringr** *will work* on factors. The **forcats** functions expand the string manip-  
 472 ulation toolbox with factor-specific utilities. Similar in philosophy to **stringr**  
 473 where functions started with **str\_**, in **forcats** most functions start with **fct\_**.

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

### 478 2.4.1 Change the order of the levels

479 One example use-case where you would want to change the order of the levels  
 480 of a factor is when plotting. Your categorical variable, for example, may not be  
 481 plotted in the order you want. If we plot the distribution of each variable across  
 482 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!
```

```

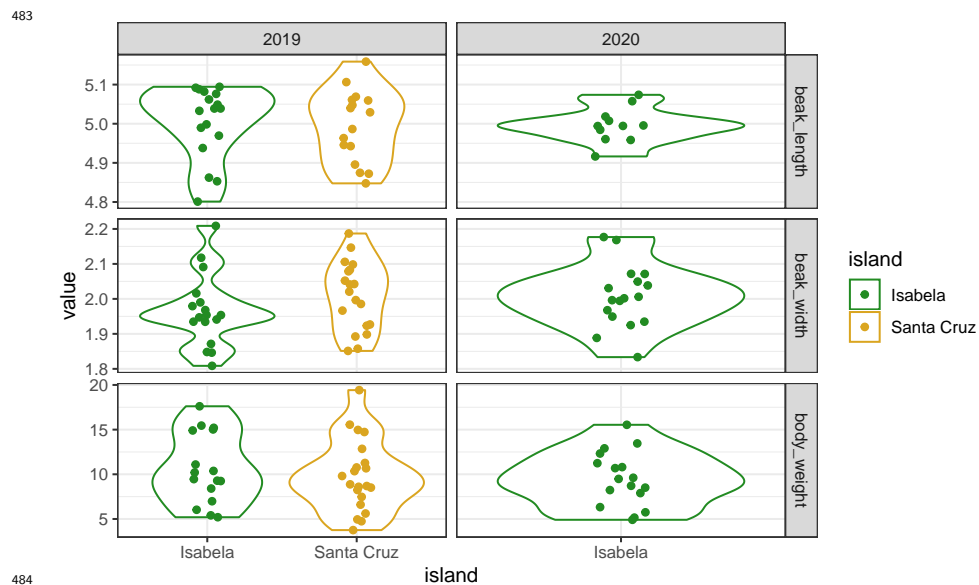
library(ggplot2)
ggplot(data, aes(x = island, y = value, color = island)) +
  geom_violin() +
  geom_jitter(width = 0.1) +
  facet_grid(variable ~ year, scales = "free") +
  theme_bw() +
  scale_color_manual(values = c("forestgreen", "goldenrod"))
}

```

```

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

```



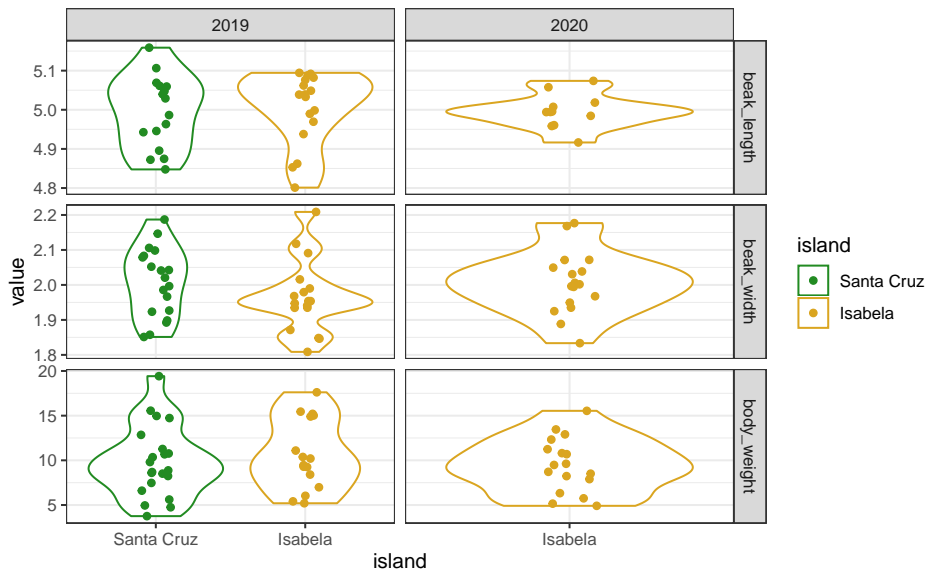
485 Here, the islands (horizontal axis) and the variables (the facets) are displayed  
 486 in alphabetical order. When making a figure you may want to customize these  
 487 orders in such a way that your message is optimally conveyed by your figure,  
 488 and this may involve playing with the order of levels.

489 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!

```



491

492 Beware that reordering a factor *does not change* the order of the items within  
 493 the vector, only the order of the *levels*. So, it does not introduce any mismatch  
 494 between the `island` column and the other columns! It only matters when the  
 495 levels are called, for example, in 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
```

496 Alternatively, use `fct_inorder` to set the order of the levels to the order in  
 497 which they 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"
```

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

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

499 Other variants exist to do more complex reordering, all present in the forcats  
 500 cheatsheet, for example: \* `fct_infreq` to re-order according to the frequency

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

## 507 2.4.2 Change the levels themselves

508 Changing the levels of a factor will change the labels in the actual vector. It  
 509 is similar to performing a string substitution in `stringr`. One can change the  
 510 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
```

511 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
```

512 Again, we do not provide an exhaustive list of `forcats` functions here but the  
 513 most usual ones, to give a glimpse of many things that one can do with factors.  
 514 So, if you are dealing with factors, remember that `forcats` may have handy tools  
 515 for you. Among others: \* `fct_anon` to “anonymize”, i.e. replace the levels by  
 516 random integers \* `fct_lump` to collapse levels together based on their frequency  
 517 (e.g. the two most frequent levels together)

## 518 2.4.3 Dropping levels

519 If you use factors in your tibble and get rid of one level, for any reason, the  
 520 factor will usually remember the old levels, which may cause some problems  
 521 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"
```



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

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

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

527 Note that this is equivalent to doing:

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

#### 528 2.4.4 Other things

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

#### 531 2.4.5 Take home message for forcats

532 Use this package to manipulate your factors. Do you need factors? Or are  
533 character vectors enough? That is your call, and may depend on the kind of  
534 analyses you want to do and what they require. We saw here that for plotting,  
535 having factors can allow you to do quite some tweaking of the display. If you  
536 encounter a situation where the order of encoding of your character vector starts  
537 to matter, then maybe converting into a factor would make your life easier. And  
538 if you do so, remember that lots of tools to perform all kinds of manipulation  
539 are available to you with both `stringr` and `forcats`.

### 540 2.5 External resources

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

- 542 • The `readr/tibble/tidyr` and `forcats` cheatsheets.
- 543 • This link on the concept of tidy data
- 544 • The `tibble`, `tidyr` and `forcats` websites
- 545 • The `broom`, `tidymodels`, `tidygraph` and `tidytree` 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>

```

565 `readr` automatically loads the data in a `tibble`, as we have seen in chapter 1  
 566 and 2. Calling the `tibble` gives a nice preview of what it contains. We have data  
 567 for 5,831 mammal species, and the variables contain information on taxonomy,  
 568 (broad) habitat, mass, IUCN status, and diet.

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

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

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

## 578 3.2 Working with existing variables

### 579 3.2.1 Renaming variables with `rename()`

580 The variable names in the phylacine dataset are descriptive, but quite unpracti-  
 581 cal. Typing `Binomial.1.2.` is cumbersome and subject to typos (in fact, I just  
 582 made one). `binomial` would be much simpler to use.

583 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>
```

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

589 You can refer to variables names either with strings or directly as objects,  
 590 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"
)
```

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

```
593 #> # A tibble: 5,831 x 24
594 #>   binomial order family genus species terrestrial marine freshwater aerial
595 #>   <chr>      <chr> <chr> <chr> <chr>          <dbl> <dbl>          <dbl> <dbl>
596 #> 1 Abditom~ Rode~ Murid~ Abdi~ latide~          1 0              0 0
597 #> 2 Abeomel~ Rode~ Murid~ Abeo~ sevia          1 0              0 0
598 #> 3 Abraway~ Rode~ Crice~ Abra~ ruschii          1 0              0 0
```

```

599 #> 4 Abrocom~ Rode~ Abroc~ Abro~ bennet~          1      0      0      0
600 #> 5 Abrocom~ Rode~ Abroc~ Abro~ bolivi~          1      0      0      0
601 #> 6 Abrocom~ Rode~ Abroc~ Abro~ budini          1      0      0      0
602 #> # ... with 5,825 more rows, and 15 more variables: life_habit_method <chr>,
603 #> #   life_habit_source <chr>, mass_g <dbl>, mass_method <chr>,
604 #> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
605 #> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
606 #> #   diet_plant <dbl>, diet_vertibrate <dbl>, diet_invertebrate <dbl>,
607 #> #   diet_method <chr>, diet_source <chr>

```

### 3.2.2 The pipe operator %>%

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

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

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

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

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

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

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

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

636 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_ruschi"
```

```
#> [4] "Abrocoma_bennettii" "Abrocoma_boliviensis" "Abrocoma_budini"
```

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

```
phylacine %>% pull(binomial) %>% head()
```

```
#> [1] "Abditomys_latidens" "Abeomelomys_sevia" "Abrawayaomys_ruschi"
```

```
#> [4] "Abrocoma_bennettii" "Abrocoma_boliviensis" "Abrocoma_budini"
```

### 3.2.3 Select variables with `select()`

To extract a set of variables (i.e. columns), use the conveniently-named `select()`. The basic syntax is the same as `rename()`: pass your data as the first argument, then call the 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_ruschi
```

```
#> 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 ruschi       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

```

643 You can select by variable numbers. This is not recommended, as prone to  
644 errors, especially 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

```

645 `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>
```

646 `select()` and `rename()` are pretty similar, and in fact, `select()` can also  
647 rename variables 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
```

648 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
```

### 649 3.2.4 Select variables with helpers

650 The Rstudio team just released `dplyr` 1.0.0, and along with it, some nice  
 651 helper functions to ease the selection of a set of variables. I give three examples  
 652 here, and encourage you to look at the documentation (`?select()`) to find out  
 653 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

```

### 654 3.2.5 Rearranging variable order with relocate()

655 The order of variables seldom matters in `dplyr`, but due to popular demand,  
 656 `dplyr` now has a dedicated verb to rearrange the order of variables. The syntax  
 657 is identical to `rename()`, `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_vertibrate <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_vertibrate
#>   <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>

```

## 658 3.3 Working with observations

### 659 3.3.1 Ordering rows by value - arrange()

660 `arrange()` sorts rows in the data by **ascending** value for a given variable. Use  
 661 the wrapper `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 190000000
#> 2 Balaena_mysticetus   100000000
#> 3 Balaenoptera_physalus  70000000
#> 4 Caperea_marginata    32000000
#> 5 Megaptera_novaeangliae 30000000
#> 6 Eschrichtius_robustus  28500000
#> # ... 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

```

### 3.3.2 Subset rows by position - `slice()`

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 Abraway~ Rode~ Crice~ Abra~ ruschii          1      0              0      0
#> 2 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(-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~ Balae~ 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>
```

665 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 Rhinol~ 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>
```

### 3.3.3 Subsetting rows by value with filter()

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

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

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

```
# megafauna
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 megafauna
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
```

673 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 :(
```

674 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>
```

675 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~ uspall~          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>
```

676 Tip: `dplyr` introduces the useful function `between()` that does exactly what  
 677 the name 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
```

678 Note that you can pipe operations inside function arguments as in the last line  
 679 above (arguments are expressions, after all!).

## 680 3.4 Making new variables

### 681 3.4.1 Create new variables with `mutate()`

682 Very often in data analysis, you will want to create new variables, or edit existing  
683 ones. 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_ruschii    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
```

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

688 `mutate()` adds a variable to the table, and keeps all other variables. Sometimes  
 689 you may want to just keep the new variable, and drop the other ones. That's the  
 690 job of `mutate()`'s twin sibling, `transmute()`. For example, I want to combine  
 691 `diet_invertebrate` and `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
```

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

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertibrate,
    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
```

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

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertibrate,
    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

```

697 Sometimes, you may need to perform an operation based on the row number (I  
 698 don't have a good example in mind). `tibble` has a built-in function to do just  
 699 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_ruschii
#> 4 4      Abrocoma_bennettii
#> 5 5      Abrocoma_boliviensis
#> 6 6      Abrocoma_budini
#> # ... with 5,825 more rows

```

### 700 3.4.2 Summarise observations with `summarise()`

701 `mutate()` applies operations to all observations in a table. By contrast,  
 702 `summarise()` applies operations to *groups* of observations, and returns, er,  
 703 summaries. The default grouping 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.
```

704 Above you can see that bats account for a large portion of mammal species  
 705 diversity (`nb_aerial`). How much exactly? Just as with `mutate()`, you can  
 706 perform operations 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
```

707 One fifth!

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

709 Here's a simple trick with logical (TRUE / FALSE) variables. Their sum is the  
 710 count of observations that evaluate to TRUE (because TRUE is taken as 1 and  
 711 FALSE as 0) and their mean is the proportion of TRUE observations. This can be  
 712 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
```

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

717 `mutate()` and `transmute()` can compute summaries as well, but they will return  
 718 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
```

### 719 3.4.3 Grouping observations by variables

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

724 `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
```

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

728 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>
```

At first glance, nothing has changed, apart from an extra line of information in the output that tells me the observations have been grouped. But now here's what happen if I run 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
```

I get one value for each group.

Observations can be grouped by multiple variables, which will output a summary for every 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

```

736 Whenever you call `summarise()`, the last level of grouping is dropped. Note  
 737 how in the output table above, observations are still grouped by order, and no  
 738 longer by IUCN status. 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

```

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

742 For example, I would like to know how the species in each order are distributed  
 743 between the different levels of threat in the IUCN classification. To get these  
 744 proportions, I need to first get the count of each number of species in a level of  
 745 threat inside an order, and divide 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

```

746 10.2% of Carnivores are Endangered ("EN").

#### 747 3.4.4 Grouped data and other dplyr verbs

748 Grouping does not only affect the behaviour of `summarise`, but under circum-  
749 stances, 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 190000000
# 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

```

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

## 751 3.5 Working with multiple tables

### 752 3.5.1 Binding tables

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

756 `bind_rows()` has an option to pass a variable specifying which dataset each  
 757 observation 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

```

### 3.5.2 Combining variables of two tables with mutating joins

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

As an example, let's split the phylacine dataset in two smaller datasets, one containing 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_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
#> 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

```

764 The two datasets each contain 5 species, the first three are shared, and the two  
 765 last differ 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"

```

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

```

inner_join(diet, life_habit, by = "binomial")
#> # A tibble: 3 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
#> # ... with 2 more variables: freshwater <dbl>, aerial <dbl>

```

771 `inner_join` combined the variables, and dropped the observations that weren't  
 772 matched between the two tables. There are three other variations of mutating  
 773 joins, differing in 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_invertebrate
#>   <chr>              <dbl>         <dbl>         <dbl>
#> 1 Abrocoma_bennettii      100           0           0
#> 2 Abrocoma_boliviensis    100           0           0

```

### 774 3.5.3 Filtering matching observations between two tables 775 wiht filtering joins

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

```

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_ruschi 88 1 11
anti_join(diet, life_habit, by = "binomial")
#> # A tibble: 2 x 4
#>   binomial diet_plant diet_vertibrate diet_invertebrate
#>   <chr>      <dbl>      <dbl>      <dbl>
#> 1 Abrocoma_bennettii 100 0 0
#> 2 Abrocoma_boliviensis 100 0 0
```





## Chapter 4

# Working with lists and iteration

Every use case is ridiculous  
until it happens to you.

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

### 4.1 List columns with `tidyr`

#### 4.1.1 Nesting data

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

788 NB: Nesting data is done using `tidyr::nest`, which is different from the simi-  
 789 larly named `tidyr::nesting`.

790 The example below shows how *Phylacine* data can be converted into a nested  
 791 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"
```

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

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

### 797 4.1.2 Unnesting data

798 A nested tibble can be converted back into the original, or into a processed form,  
 799 using `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

```

800 The `unnest_longer` and `unnest_wider` variants of `unnest` are maturing func-  
 801 tions, that is, not in their final form. They allow interesting variations on  
 802 unnesting — these are shown here but advised against. Unnest the data first,  
 803 and then convert it to the form needed.

### 804 4.1.3 Working with list columns

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

808 Two examples are shown below when getting the class and number of rows of  
 809 the nested 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"

```

## 810 Functionals

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

## 814 4.2 Iteration with map

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

### 817 Why use purrr

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

### 822 4.2.1 Basic use of map

823 `map` works very similarly to `lapply`, where `.x` is object on whose elements to  
 824 apply the 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
```

825 `map` works on any list-like object, which includes vectors, and always returns a  
 826 list. `map` takes two arguments, the object on which to operate, and the function  
 827 to apply to each 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
```

## 828 4.2.2 map variants returning vectors

829 Though `map` always returns a list, it has variants named `map_*` where the suffix  
 830 indicates the return type. `map_chr`, `map_dbl`, `map_int`, and `map_lgl` return  
 831 character, double (numeric), 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

```

### 832 4.2.3 map variants returning data frames

833 map\_df returns data frames, and by default binds dataframes by rows, while  
 834 map\_dfr does this explicitly, and map\_dfc does returns a dataframe bound by  
 835 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>

```

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

838 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>
```

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

#### 841 4.2.4 Working with list columns using map

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

846 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
```

#### 847 4.2.5 Selective mapping using map variants

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

851 In this example, the nested tibble is given a new column using `dplyr::mutate`,  
 852 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>
```

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

## 856 4.3 More map variants

857 `map` also has variants along the axis of how many elements are operated upon.  
 858 `map2` operates on two vectors or list-like elements, and returns a single list as  
 859 output, while `pmap` operates on a list of list-like elements. The output has as  
 860 many elements as the input lists, which must be of the same length.

### 861 4.3.1 Mapping over two inputs with `map2`

862 `map2` has the same variants as `map`, allowing for different return types. Here  
 863 `map2_int` 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
```

864 `map2` doesn't have `_at` and `_if` variants.

865 One use case for `map2` is to deal with both a list element and its index, as shown  
 866 in the example. This may be necessary when the list index is removed in a  
 867 `split` or `nest`. This can also be done with `imap`, where the index is referred to  
 868 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

```

### 869 4.3.2 Mapping over multiple inputs with pmap

870 pmap instead operates on a list of multiple list-like objects, and also comes with  
 871 the same return type variants as map. The example shows both aspects of pmap  
 872 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"

```

## 873 4.4 Combining map variants and tidyverse func- 874 tions

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

879 In the pipeline:

- 880 1. The tibble becomes a nested dataframe by order (using `tidyr::nest`),
- 881 2. If there are enough data points ( $> 100$ ), a linear model of mass  $\sim$  plant  
 882 diet is fit (using `purrr::map_if`, and `stats::lm`),
- 883 3. The model coefficients are extracted if the model was fit (using  
 884 `purrr::map` & `dplyr::case_when`),
- 885 4. The model coefficients are converted to data for plotting (using  
 886 `purrr::map`, `tibble::tibble`, & `tidyr::pivot_wider`),

- 887 5. The raw data is plotted along with the model fit, taking the title from the  
 888 nested data 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}'))
})

```

## 889 4.5 A return to map variants

890 Lists are often nested, that is, a list element may itself be a list. It is possible  
 891 to map a function over elements at a specific depth.

892 In the example, `phylacine` is split by `order`, and then by IUCN status, creating  
 893 a two-level 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

```

894 **4.5.1 Iteration without a return**

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

899 `walk` is the function for this task. It has only the variants `walk2`, `iwalk`, and  
 900 `pwalk`, whose logic is similar to `map2`, `imap`, and `pmap`. In the example, the  
 901 function applied to each list 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
```

### 902 4.5.2 Modify rather than map

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

906 In the example, simply adding 2 to each vector element produces an error,  
 907 because the output is a `numeric`, or `double`. `modify` helps ensure some type  
 908 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"
```

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

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

### 911 A note on `invoke`

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

## 916 4.6 Other functions for working with lists

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

### 919 4.6.1 Filtering lists

920 Lists can be filtered on any predicate using `keep`, while the special case `compact`  
 921 is applied when the empty elements of a list are to be filtered out. `discard` is  
 922 the opposite of `keep`, and keeps only elements not satisfying a condition. Again,  
 923 the predicate is specified 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
```

924 `head_while` is bit of an odd case, which returns all elements of a list-like object  
 925 in sequence 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
```

### 926 4.6.2 Summarising lists

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

### 930 4.6.3 Reduction and accumulation

931 `reduce` helps combine elements along a list using a specific function. Consider  
 932 the example 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
```

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

936 The way `reduce` works in the example below is to take the first element and find  
 937 its intersection with the second, and to take the result and find its intersection  
 938 with the third and 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)
```

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

#### 943 4.6.4 Miscellaneous operation

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



948 `flatten` has a similar behaviour, and converts a list of vectors or list of lists to a  
 949 single 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"
```

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

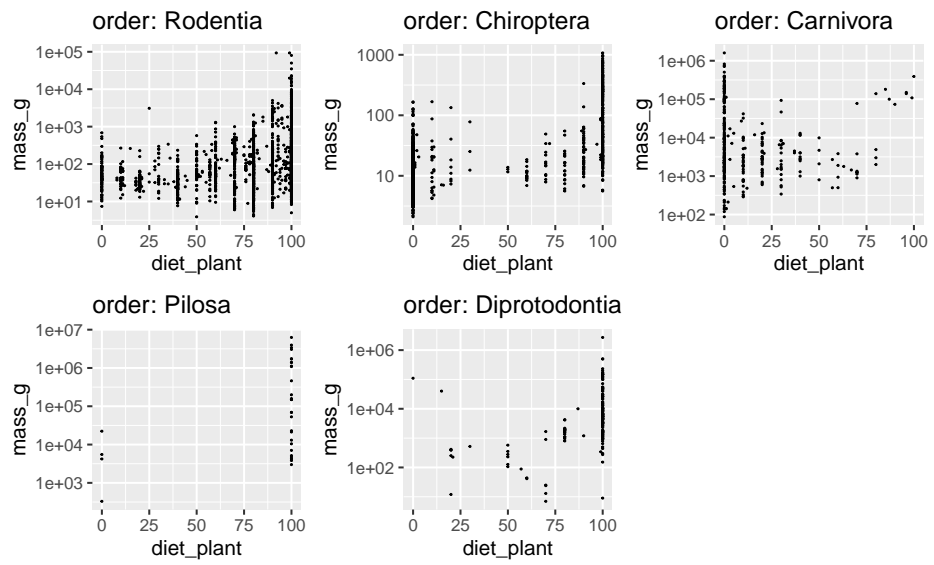
```
this_list = split(data, data$order) %>%
  map(function(df) {split(df, df$iucn_status)})

# 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])
```

## 953 4.7 Lists of ggplots with patchwork

954 The patchwork library helps compose `ggplots`, which will be properly intro-  
 955 duced in the next session. `patchwork` usually works on lists of `ggplots`, which  
 956 can come from a standalone list, or from a list column in a nested dataframe.  
 957 The example below shows the latter, with the `data` data frame from earlier.

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



## 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>
```

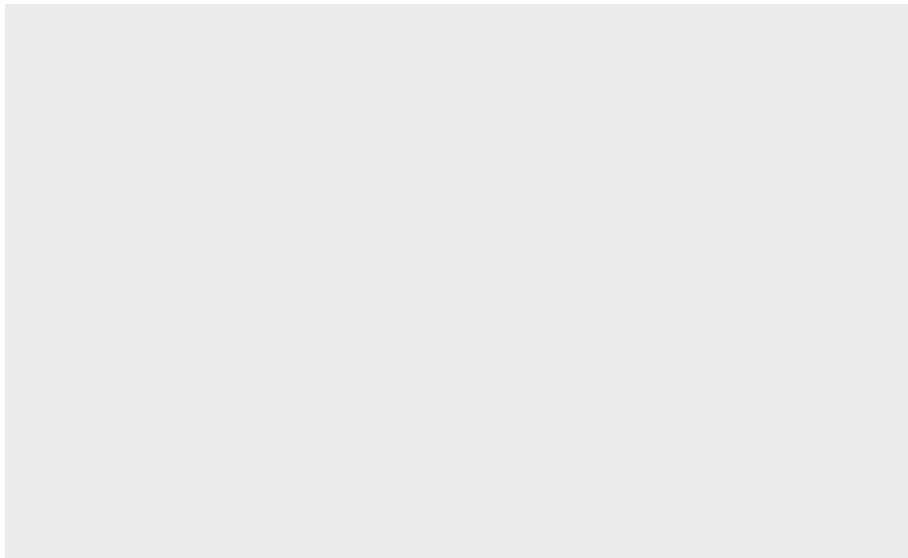
This pseudocode snippet illustrates a fundamental aspect of `ggplot2`, which is that plots are built by *successive* commands, each corresponding to a layer, assembled together using the `+` operator. This might seem less practical than having a whole plot made in a single call to the `plot` function,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

1028 created mutator strains of bacteria (that is, bacteria that mutate at a much  
 1029 higher rate than usual) and tracked their growth through time and at different  
 1030 concentrations of an agent supposed to activate the full “mutation potential” of  
 1031 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
```

1032 The different strains of bacteria were grown in two different **assays**, whose  
 1033 details are irrelevant for the purpose of this tutorial. **cfu** is the number of  
 1034 colony forming units while OD600 is the optical density at 600nm wavelength;  
 1035 both are estimates of bacterial population density. **ratio** represents the ratio  
 1036 in mutants between two time points, T0 and T1 (encoded in **time**).

1037 In this table, the unit of observation is the time point (T0 and T1 are on different  
 1038 rows), therefore the values of **ratio**, which are attributed to each T0-T1 pair,  
 1039 are duplicated to yield one value per time point. To make our life easier with  
 1040 later plotting and to stay within the *tidy* spirit of the tidyverse (where one table  
 1041 should have one unit of observation), we use the tools we have already learnt to  
 1042 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
```

## 5.3 Geom layers

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

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.

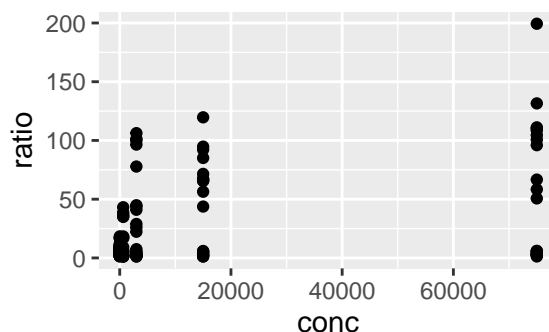
Some of the main aesthetics to know, besides geom-specific coordinates (e.g. `x`, `y`), include: `color`, `fill` (color used to fill surfaces), `group` (used e.g. to plot multiple lines with similar aspect on the same plot), `alpha` (transparency), `size`, `linetype`, `shape`, and `label` (for showing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

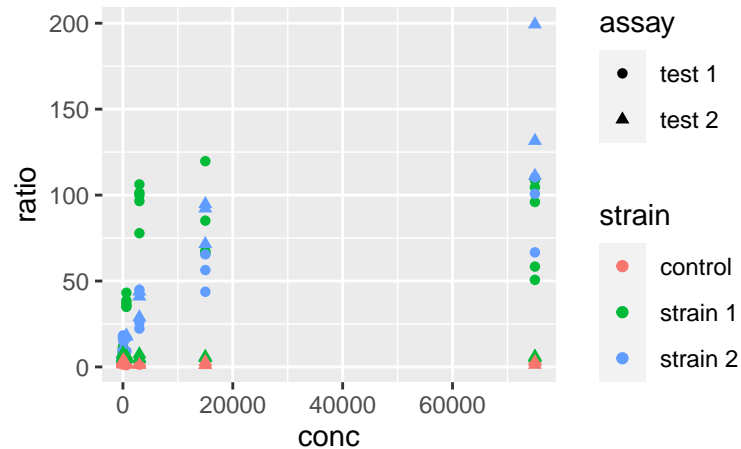
```
ggplot(data2) +  
  geom_point(mapping = aes(x = conc, y = ratio))
```



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

1071 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))
```

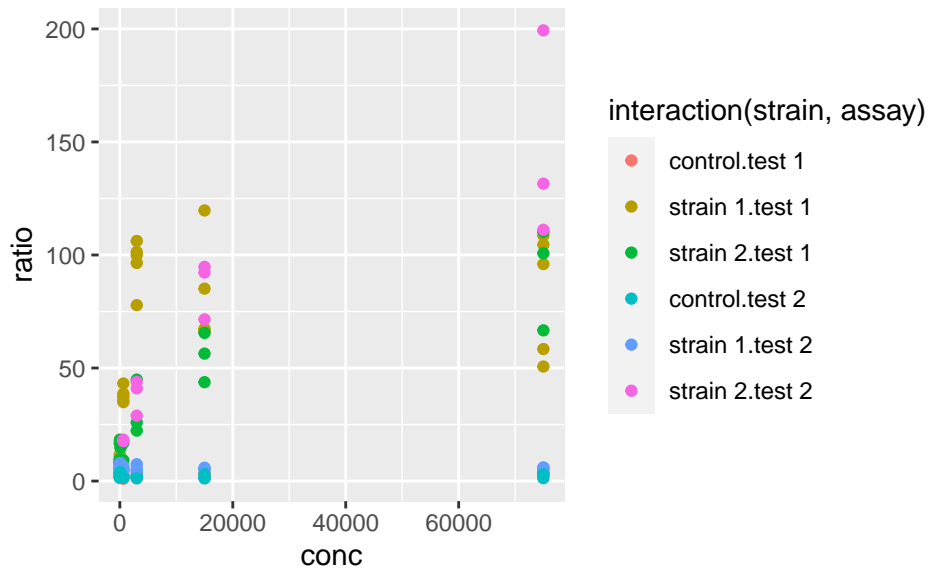


1072

1073 Do you want to map several variables to a single aesthetic? Then **interaction**  
1074 from base R can be used within a **ggplot**:

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

1075



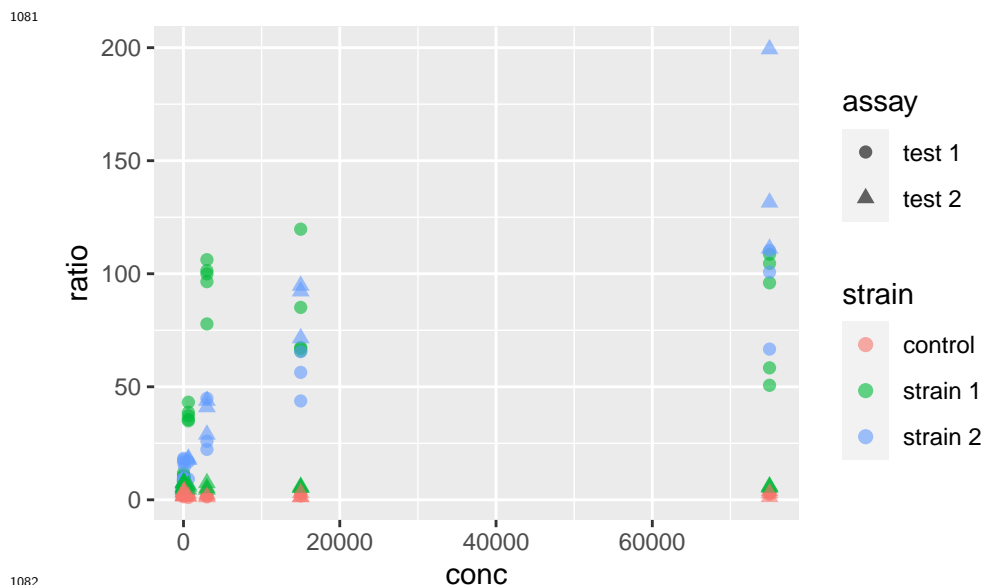
1076



### 1077 5.3.2 Fixed aesthetics

1078 Fixed graphical parameters (i.e. that are not mapped to a variable) should be  
 1079 added as arguments of the geom *outside* the `aes` command. For example, to  
 1080 make *all* points a little bigger and more transparent, we can use

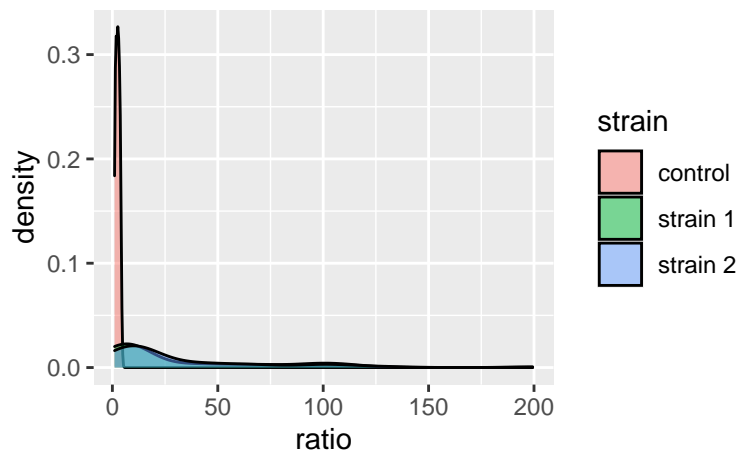
```
ggplot(data2) +  
  geom_point(  
    mapping = aes(x = conc, y = ratio, color = strain, shape = assay),  
    size = 2, alpha = 0.6  
  )
```



### 1083 5.3.3 Statistical transformation

1084 Statistical transformations, or `stat` functions, can be applied to the data within  
 1085 a `geom` call. Actually, statistical transformations are *always* applied within a  
 1086 `geom` call, but most of the time the identity function is used. To illustrate,  
 1087 consider the following plot showing a distribution of `ratio` for different strains:

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

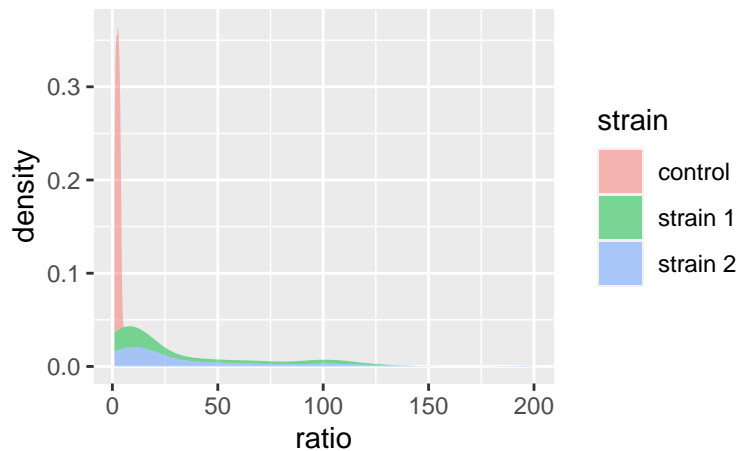


1088

1089 Here, the `density` axis is not part of the original dataset `data2`; it was computed  
 1090 from the data, for each value of `ratio`, by using a density-estimation algorithm.  
 1091 This shows that `stat_density` (and not `stat_identity`) is the default `stat`  
 1092 used in `geom_density`. Every `geom` comes with its default `stat`.

1093 Similarly, `stat` functions can be used in place of `geom` because every `stat` has  
 1094 a default `geom` associated to it. So, we can call:

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

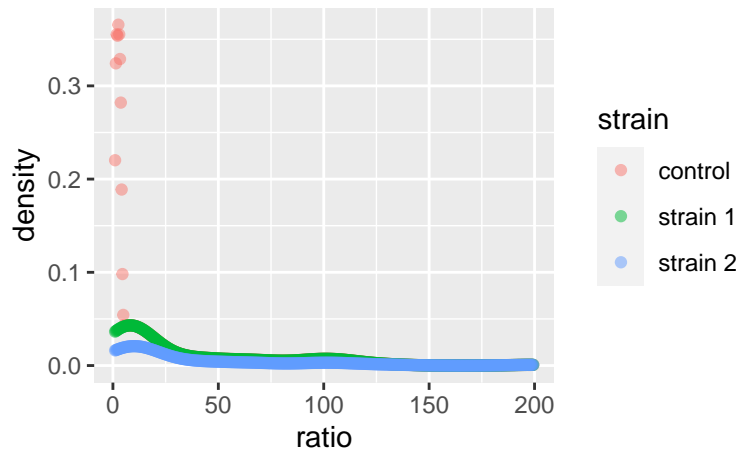


1095

1096 which has `geom_density` as default `geom`.

1097 It is possible to override the default `stat` using the `stat` argument of `geom`,  
 1098 and conversely, it is possible to change the default `geom` associated with a given  
 1099 `stat`. For example, say we want to plot our densities as points. Then,

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



1100

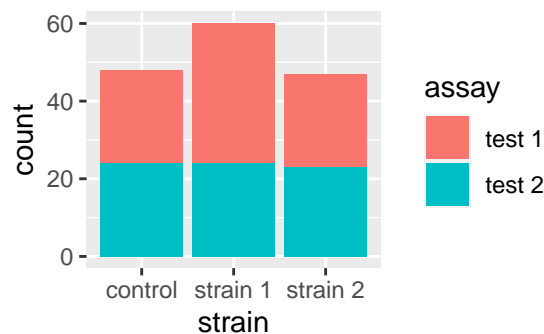
1101 does the job (note that we replaced `fill` with `color` because our points do not  
 1102 have a surface to fill).

1103 Note that default `geom-stat` combinations are usually well thought of (density  
 1104 plots are a good example). Therefore, it is often not necessary to play with  
 1105 stats. It may matter in some specific cases, e.g. when using `geom_bar`, but we  
 1106 do not cover that here (you can check out the dedicated chapter in R for Data  
 1107 Science for an example).

#### 1108 5.3.4 Position

1109 The `position` argument of geoms allows to adjust the positioning of the geom's  
 1110 elements. It has a few variants, but the possibilities depend on the geom used.  
 1111 We illustrate those available to `geom_bar`. By default, `geom_bar` uses  
 1112 the `stat_count` statistical transformation, meaning that it will show us the  
 1113 number of observations into each category of a factor, e.g. `strain`, splitted into  
 1114 categories of another factor, e.g. `assay`:

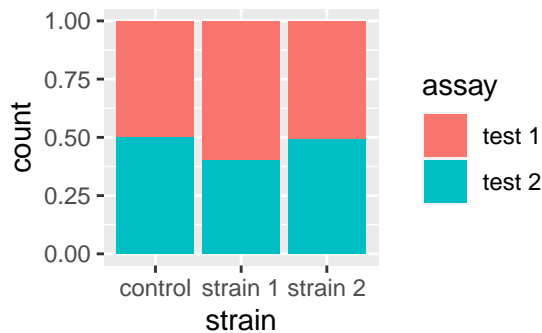
```
ggplot(data2) +  
  geom_bar(aes(x = strain, fill = assay))
```



1115

1116 If we wanted to visualize proportions instead of numbers, we could use the `fill`  
 1117 value of the `position` argument:

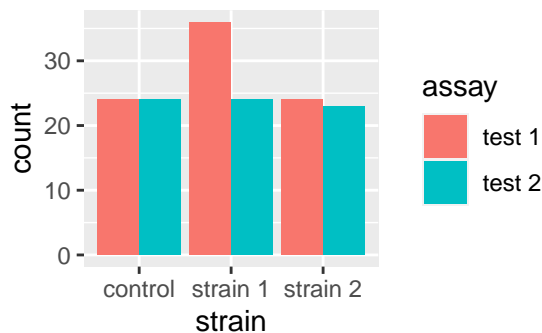
```
ggplot(data2) +  
  geom_bar(aes(x = strain, fill = assay), position = "fill")
```



1118

1119 Alternatively we could use the `dodge` option to show the different categories  
 1120 side-by-side:

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



1121

1122 Those are only two examples of what can be done. Just remember that `position`  
 1123 exists and look into the documentation of your geom of interest to see what  
 1124 position adjustments are available! (Check out `geom_jitter` as a nice wrapper  
 1125 around `geom_point` with a `jitter` position adjustment, perfect to overlay with  
 1126 boxplots or violin plots.)

### 1127 5.3.5 Other geoms

1128 The most common geoms you may encounter are:

- 1129 • `geom_point` for scatter plots and `geom_jitter` for the dodged equivalent
- 1130 • `geom_bar` for a barplot
- 1131 • `geom_text` for a scatter plot of labels
- 1132 • `geom_histogram` and `geom_density`, self-explanatory

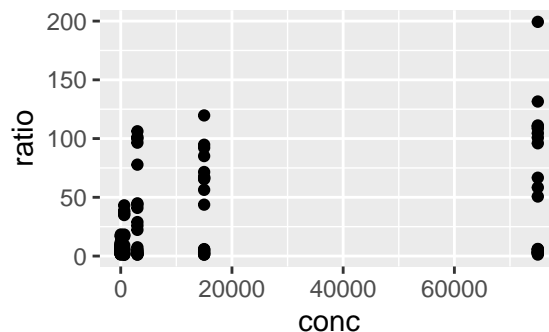
- `geom_boxplot` and `geom_violin`
- `geom_line`, `geom_path` (a line never goes backwards along the x-axis, while a path can) and `geom_smooth` (local regression smoothing)
- `geom_segment`, `geom_hline`, `geom_vline` and `geom_abline` that may come handy as annotations
- `geom_tile` for heatmaps

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

### 5.3.6 Extra on aesthetics

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

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

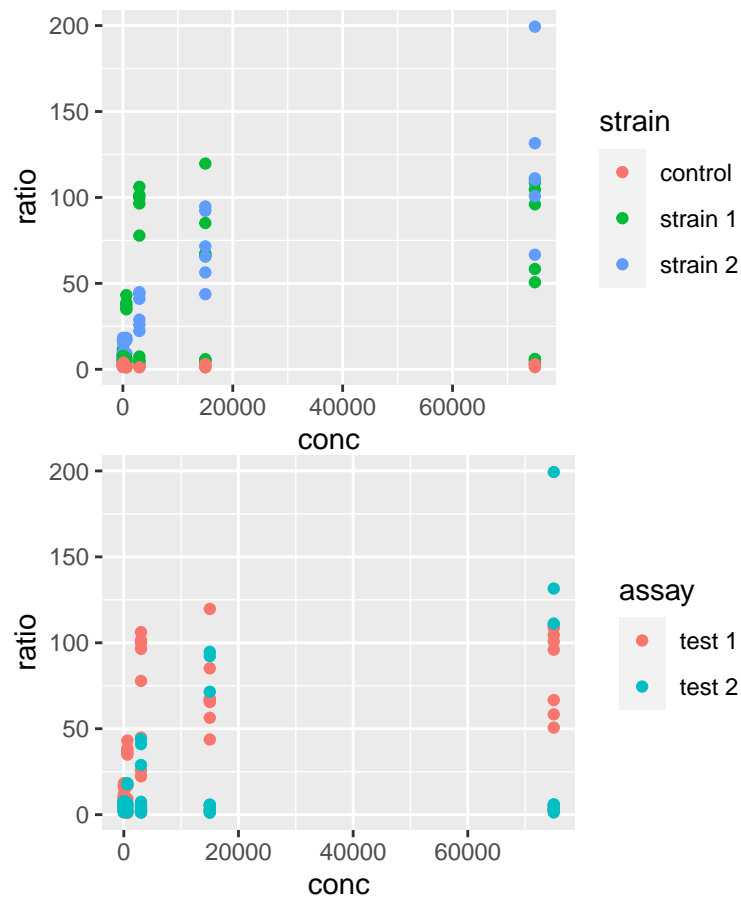


We may want to color-code the points based on `strain` or `assay`, or both, thus requiring two plots building on this single one. An important property of `ggplot` objects is that they can be assigned to variables, e.g.

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

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

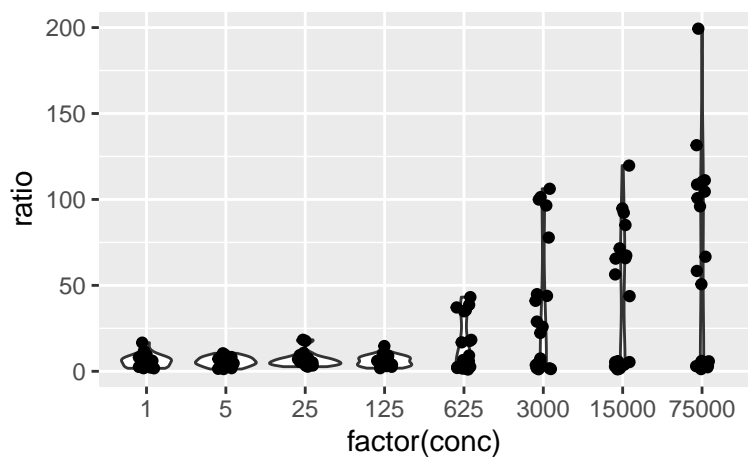
```
p + aes(color = strain)  
p + aes(color = assay)
```



### 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
```

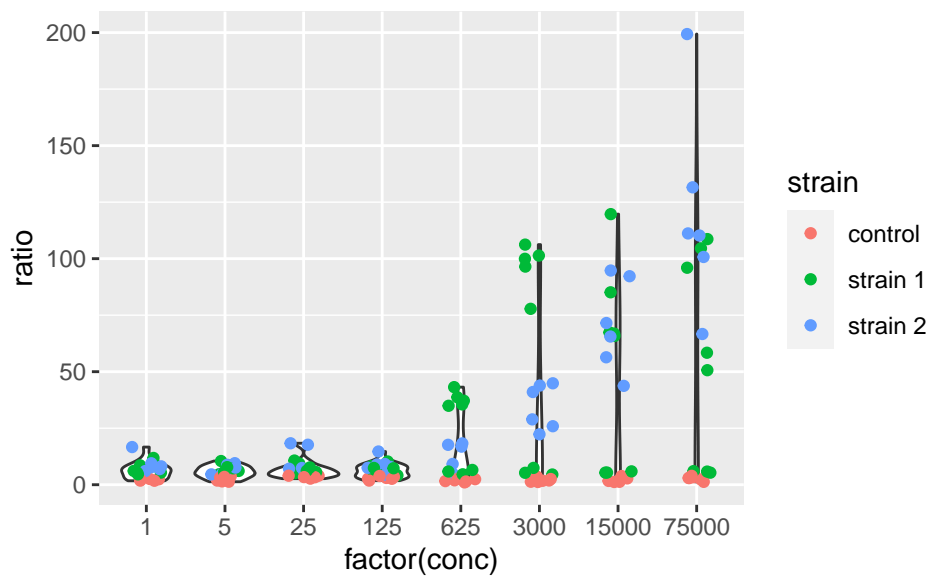


1167

1168 This shows a nice example of multiple geoms combined in a single plot. If,  
 1169 however, the aesthetics used in some geoms are geom-specific, better pass them  
 1170 to their respective `geom`. For example, if you want to color only the points but  
 1171 not the violins, use:

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

1172

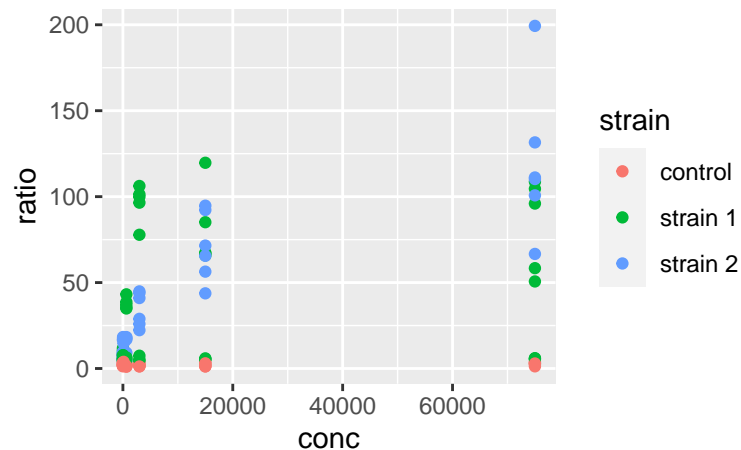


1173

### 5.3.8 Multiple geoms with different datasets

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

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



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

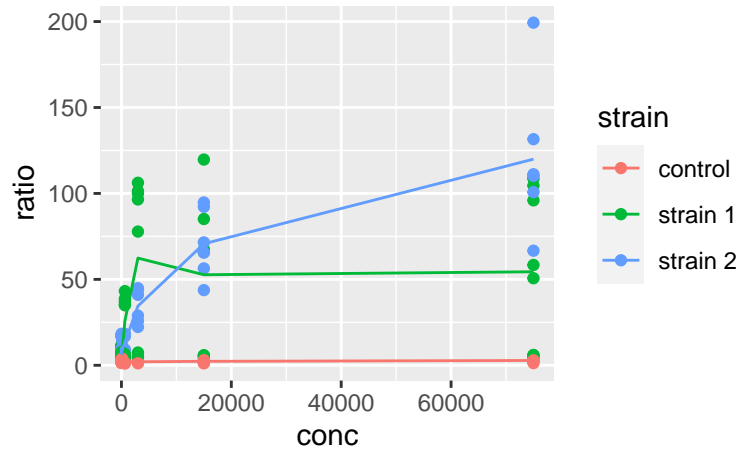
```
data3 <- data2 %>%
  group_by(conc, strain) %>%
  summarize(ratio = mean(ratio))
data3
#> # A tibble: 24 x 3
#> # Groups:   conc [8]
#>   conc strain  ratio
#>   <dbl> <chr>   <dbl>
#> 1     1 control    2.21
#> 2     1 strain 1    7.09
#> 3     1 strain 2    9.16
#> 4     5 control    2.50
#> 5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

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



1186 dataset:

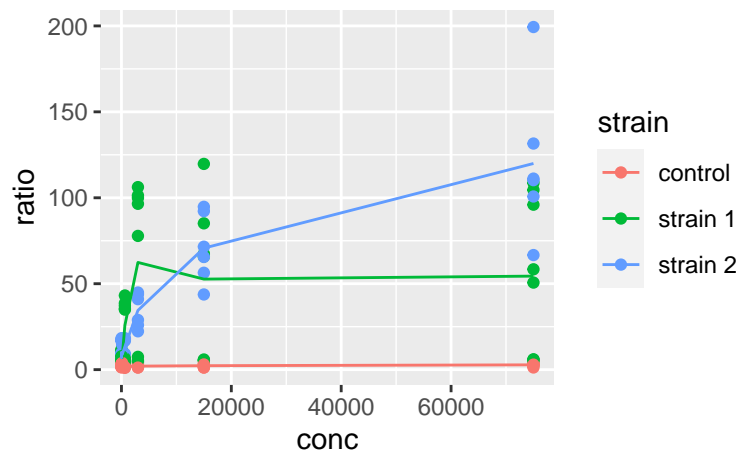
```
ggplot() +
  geom_point(data = data2, mapping = aes(x = conc, y = ratio, color = strain)) +
  geom_line(data = data3, mapping = aes(x = conc, y = ratio, color = strain))
```



1187

1188 Here, we could save some typing by writing:

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



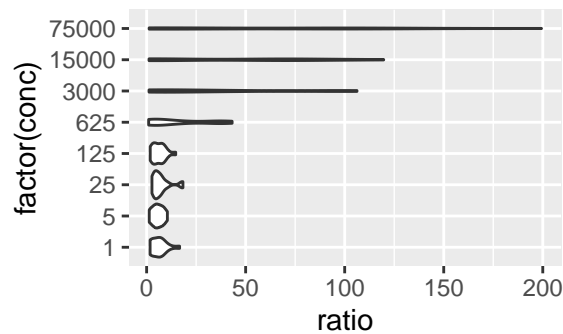
1189

1190 where `geom_line` inherits the same aesthetic mapping as `geom_point`. But then,  
 1191 you have to make sure that `data3` contains all the aesthetics that the `ggplot`  
 1192 call expects to see in each of its `geoms` (here `x`, `y` and `color`).

## 1193 5.4 Coordinate-system

1194 The default way that the plotting window is organized is an orthogonal space  
 1195 with a horizontal x-axis and a vertical y-axis. Use the `coord` commands to  
 1196 deviate from this. For example, `coord_flip` will flip the axes:

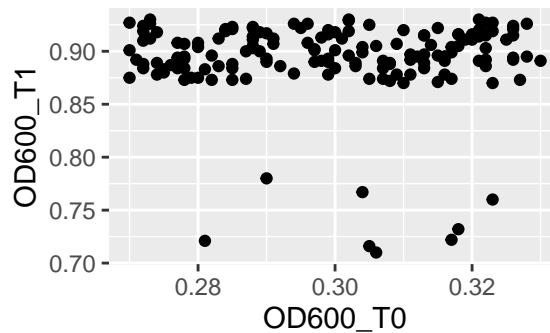
```
ggplot(data2, aes(x = factor(conc), y = ratio)) +  
  geom_violin() +  
  coord_flip()
```



1197

1198 while `coord_fixed` will fix the aspect ratio between the axes, thus showing  
 1199 them on the same scale. For example, the following plot of the optical density  
 1200 between two time points,

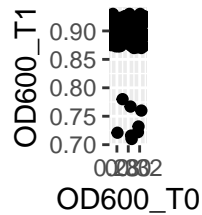
```
ggplot(data2, aes(x = OD600_T0, y = OD600_T1)) +  
  geom_point()
```



1201

1202 becomes:

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



when both axes are shown on the same scale.

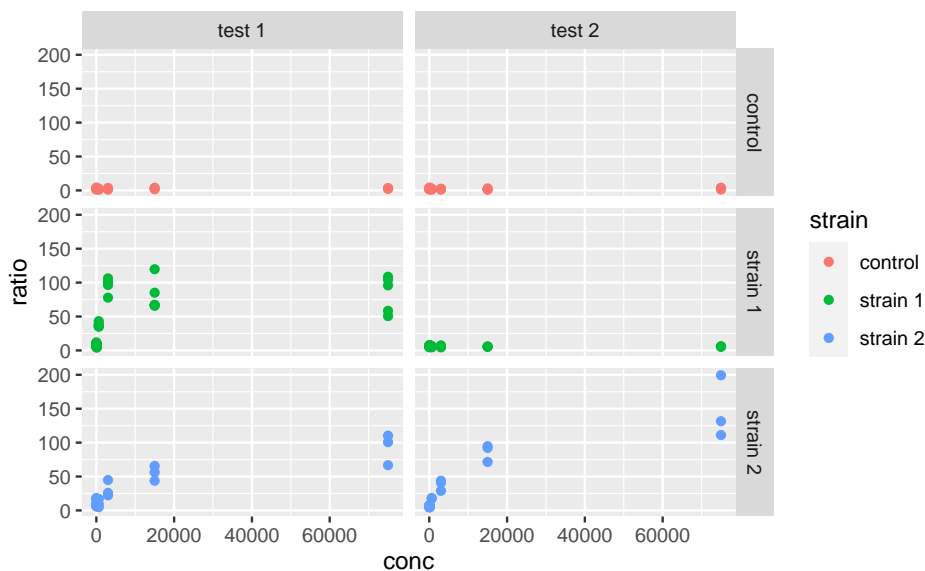
Other coordinate systems exist, depending on the need, including `coord_polar` for radial plots or `coord_quickmap`, tailored at latitude-longitude plotting.

## 5.5 Facetting

One of the most powerful features of `ggplot2` is its easy way of splitting a plot into multiple subplots, or *facets*.

There are two functions for facetting: `facet_grid` and `facet_wrap`. `facet_grid` will 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)
```

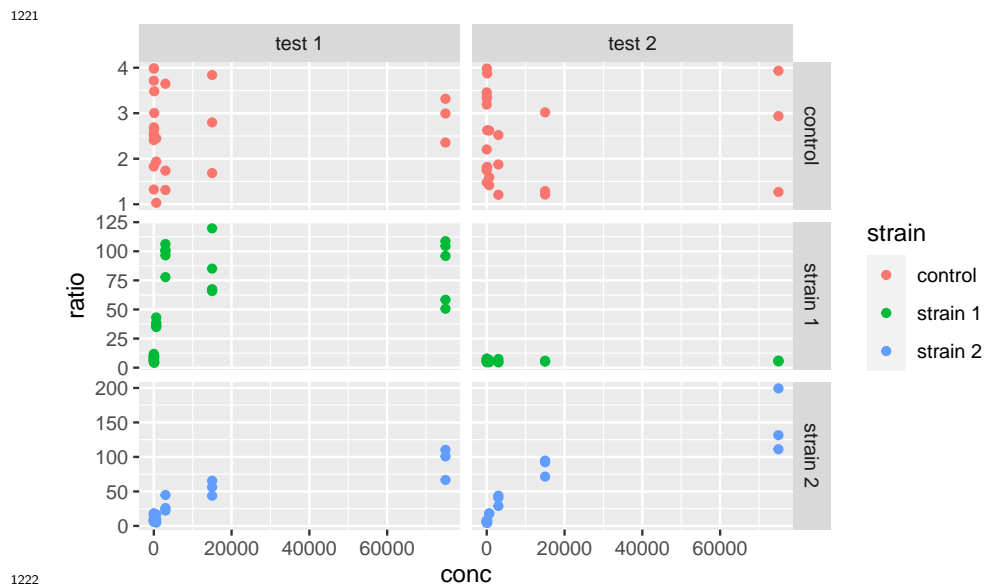


Here the tilde (`~`) symbolizes a *formula*, a type of expression in R with a left and right-hand side, which here are interpreted as variables to use for rows and

columns, respectively. If using only one variable for faceting, use `.` or nothing on the other side of the tilde.

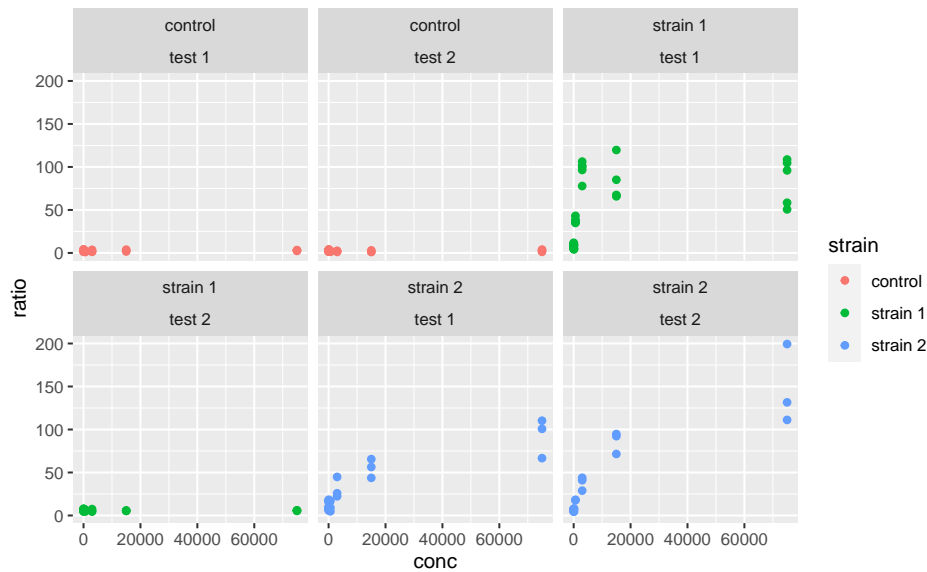
Note that facets are plotted on the same scale. We can use the `scales` argument to allow free scales, for example:

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



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

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



1227

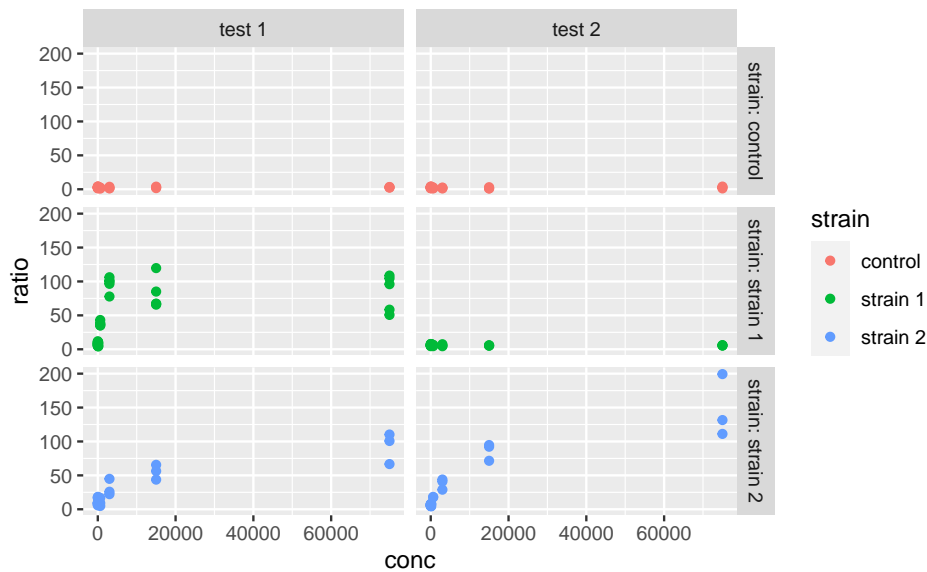
1228 where the position of the variables relative to the `~` becomes irrelevant.

1229 Note that a faceted ggplot is still *one* ggplot, not a combination of ggplots,  
 1230 which we will cover later.

1231 Custom-labelling the strips of the facets is done with the `labeller` argument.  
 1232 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))
```

1233



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:

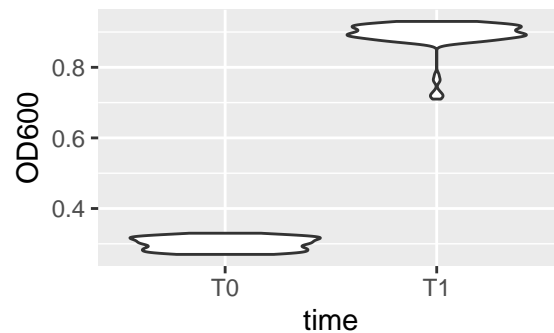
```
devtools::install_github("rscherrer/ggsim")
```

## 5.6 The right format for the dataset

One question that may come to your mind is: what is the right format of a dataset for use in `ggplot`, especially since it is part of the tidyverse? The answer is: it depends, and this is where the intergration with other tidyverse tools makes our life easier. If, for example, we want to use a variable for facetting or as an aesthetics, it is important to have this variable as a single column.

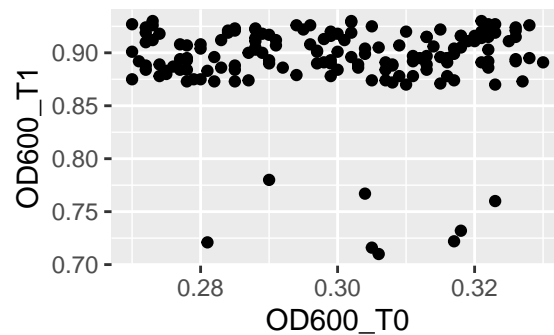
For example, in the original `data` dataset, we could have compared the optical density between the two time point:

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



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

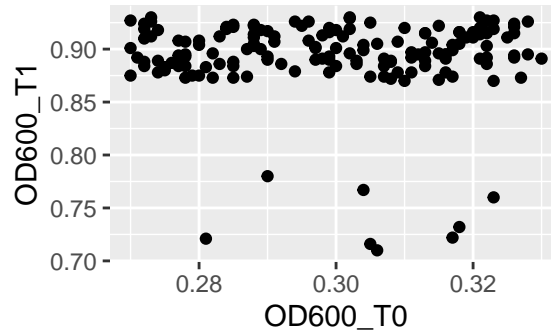
```
ggplot(data2, aes(x = OD600_T0, y = OD600_T1)) +  
  geom_point()
```



## 5.7 Plotting as part of a pipeline

What we just saw means that sometimes reformatting of a dataset is needed (e.g. using `pivot_longer` or `pivot_wider` from `tidyr`) to get this one plot done that requires reshaping. 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()
```



1274

1275 Notice the use of the pipe `%>%` to pass the resulting data frame on to the `ggplot`  
 1276 command. Because `ggplot` is called with a pipe, its first argument is already  
 1277 passed (it is the data frame coming through the pipe), so we only need to pass  
 1278 the second argument, i.e. the aesthetics mapping, to the `ggplot` function.

## 1279 5.8 Customization

1280 Now that we saw everything there is to know about structuring a `ggplot`, it is  
 1281 time to learn how to polish it (the easiest and most rewarding part!).

### 1282 5.8.1 Scales

1283 Every aesthetics can be scaled. This includes specifying what values an aesthet-  
 1284 ics can take (e.g. what colors to pick, or what range of transparencies to use),  
 1285 possible break points along the legend, or legend titles and labels, among others.  
 1286 Use the `scale_*` family of functions for that. There are many such functions,  
 1287 because many aesthetics can be modified, but the logic behind their naming is  
 1288 always the same:

```
1289 scale_<AESTHETIC>_<TYPE>
```

1290 where `<AESTHETIC>` is replaced by the aesthetic you want to scale (e.g. `color`,  
 1291 `size`, `alpha`) and `<TYPE>` is the type of variable that is mapped to this aesthetic  
 1292 (common types are `continuous`, `discrete` and `manual`). Some scaling functions  
 1293 do not take a `<TYPE>` but just an `<AESTHETIC>` in their name, e.g. `scale_alpha`.

1294 In our example, if we color-code points according to their `strain`, which is a cat-  
 1295 egorical variable, we can use `scale_color_manual` (aka `scale_colour_manual`)  
 1296 to manually pick the colors we want:

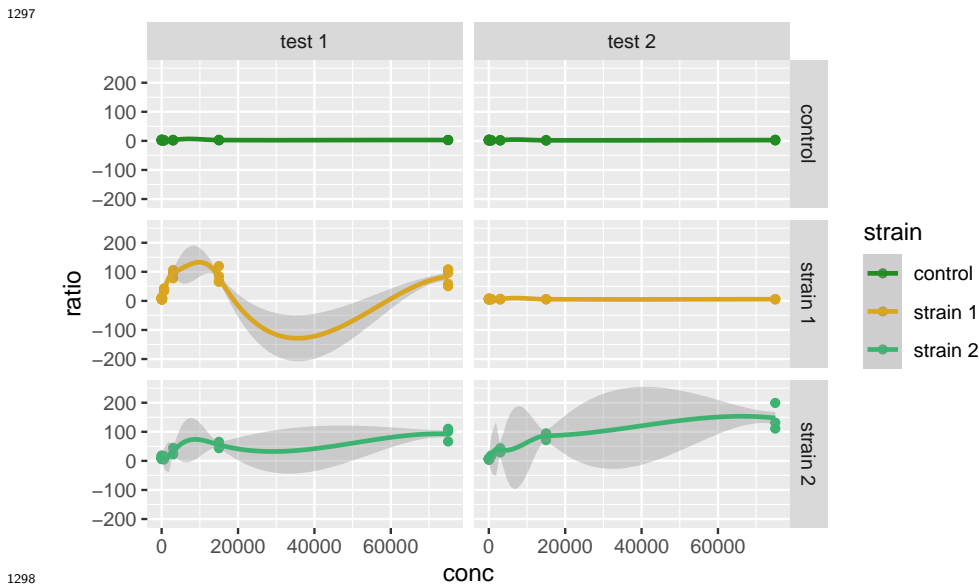
```
ggplot(data2, aes(x = conc, y = ratio, color = strain)) +
  geom_point() +
```



```

geom_smooth() + # just to spice up our use of geoms
facet_grid(strain ~ assay) +
scale_color_manual(values = c("forestgreen", "goldenrod", "mediumseagreen"))

```



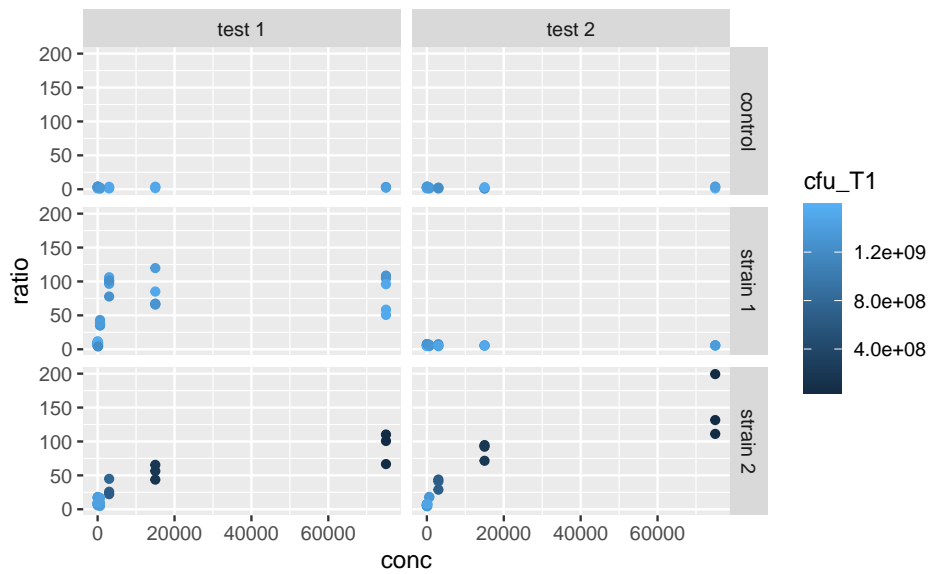
1299 Alternatively, we could color-code the points based on their number of  
 1300 CFU at time point T1, `cfu_T1`, which is a continuous variable, using  
 1301 `scale_color_continuous`. Without scaling:

```

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

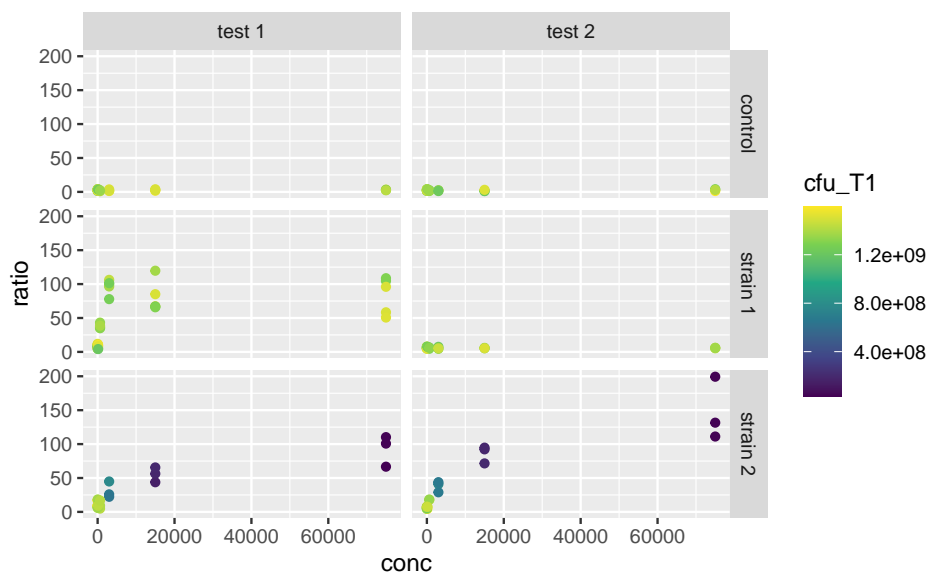
```

1302



With scaling:

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

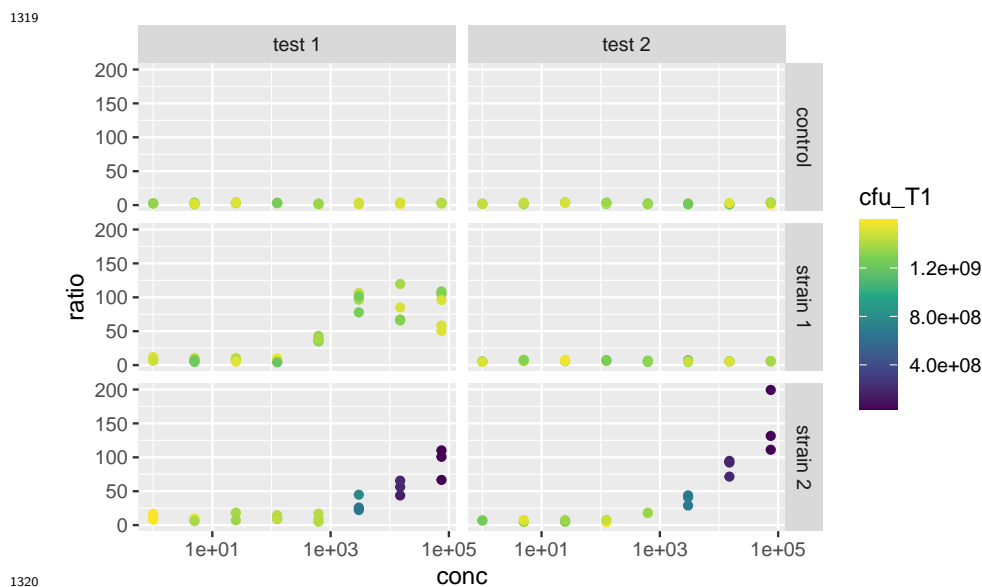


The arguments that are taken by the `scale_` function really depend on the use case, e.g. `scale_color_manual` expects discrete values,

1309 `scale_color_continuous` expects a `type` of built-in continuous color gradient,  
 1310 and `scale_color_gradient` expects a `low` and `high` color boundaries (and also  
 1311 a mid-gradient color in the case of `scale_color_gradient2`). But the logic  
 1312 shown here is similar across many aesthetics, e.g. `scale_alpha_continuous`  
 1313 and `scale_size_continuous` work in similar ways, both taking a `range`  
 1314 argument. So, lots of scaling functions to play with, of which we do not provide  
 1315 an exhaustive list here.

1316 Mandatory aesthetics, such as `x` and `y`, also have their scaling functions. If  
 1317 `x` or `y` is continuous, one can e.g. use `scale_x_log10` to show this axis on a  
 1318 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()
```



1321 More on re-scaling legend titles and labels further down.

## 1322 5.8.2 Labels

1323 The functions `ggtitle`, `xlab`, `ylab` and `labs` allow you to customize the labels  
 1324 shown for each aesthetics (remember that the x- and y-axes are aesthetics too),  
 1325 and for the main title of the plot. On to a full-fledge example:

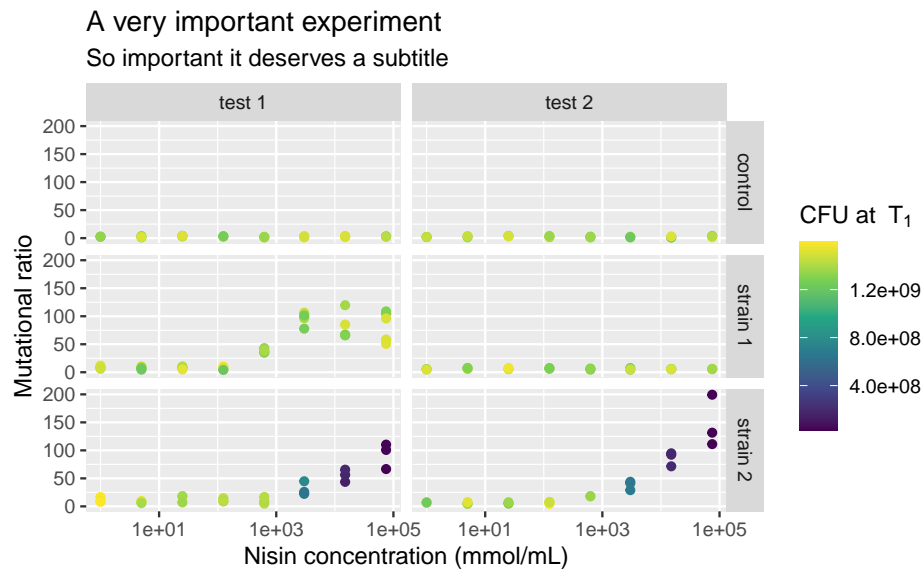
```
p <- ggplot(data2, aes(x = conc, y = ratio, color = cfu_T1)) +
  geom_point() +
```

```

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"
)

```

1326



1327

1328 Note that `xlab` and `ylab` are wrappers around `labs`, meaning that we could  
 1329 have provided `labs` with `x = ...` and `y = ...` in addition to `color = ...`,  
 1330 its arguments just need to take the names of the aesthetics. If you want no  
 1331 labels, use e.g. `xlab(NULL)` or `ylab(NULL)`.

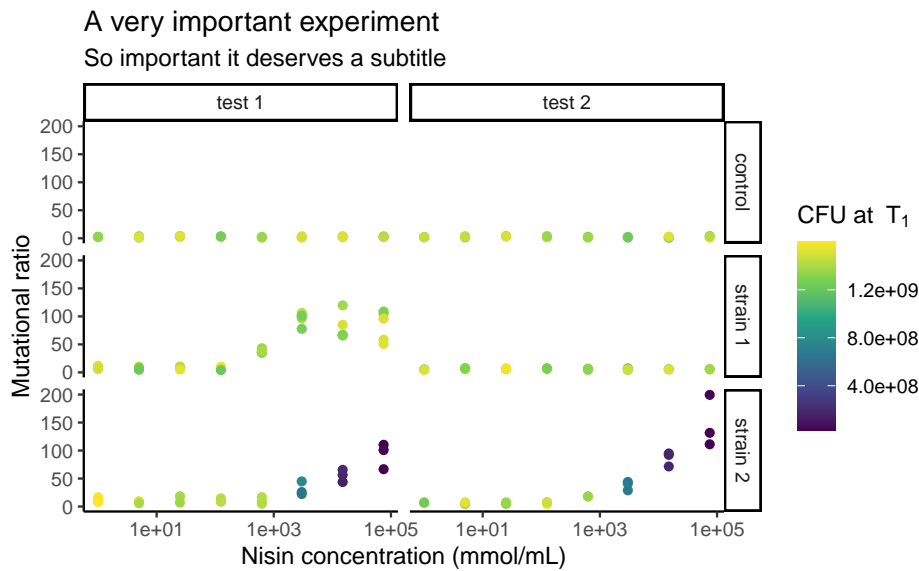
1332 Also notice the use of `parse` to display mathematical notations using the  
 1333 `plotmath` syntax. This is not part of the tidyverse though, so it is a story for  
 1334 another day, feel free to look it up (type `?bquote`)!

### 1335 5.8.3 Themes

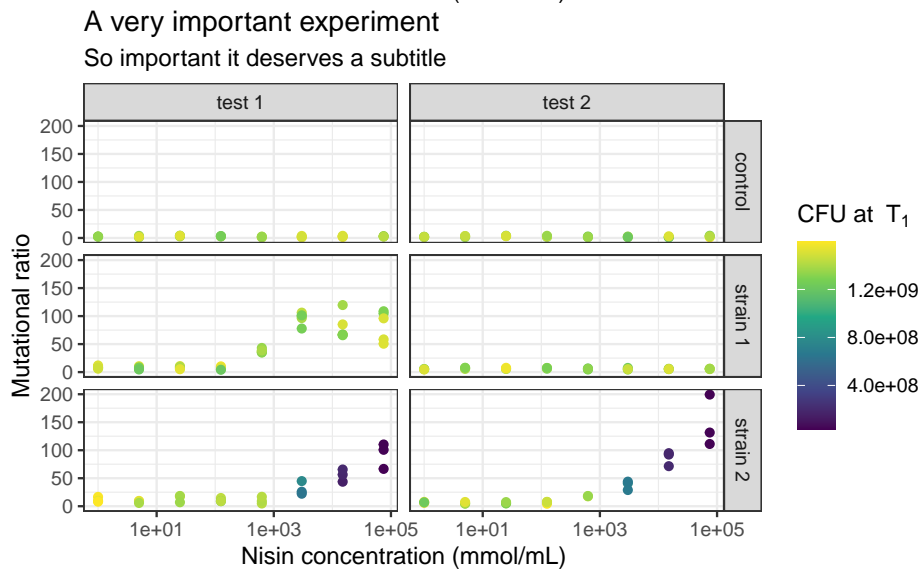
1336 You may be already frustrated that all plots have this same grey default `ggplot2`  
 1337 background. Of course, it is possible to change this too by playing with the  
 1338 `theme` functions. There are other built-in themes than the default grey one,  
 1339 such as `theme_bw` or `theme_classic`:

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

1340



1341



1342

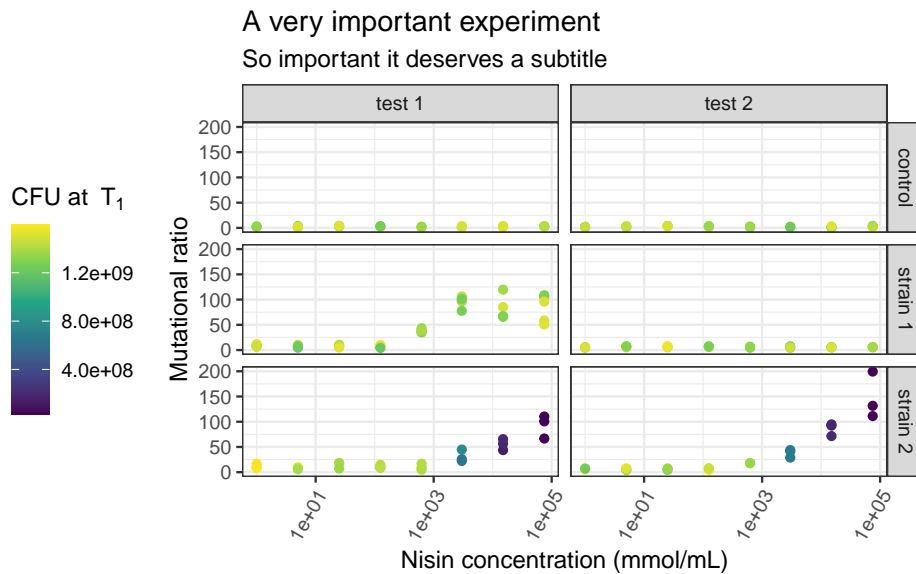
1343 The individual elements of the theme, e.g. the background grid or the color of  
 1344 the panel, can be customized using the arguments in the `theme` function. The  
 1345 `theme` function can also be used to modify stuff related to the legend or the axes  
 1346 of the plots. For example:

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

```

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

```



1347

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!

#### 5.8.4 Legend

1357

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:

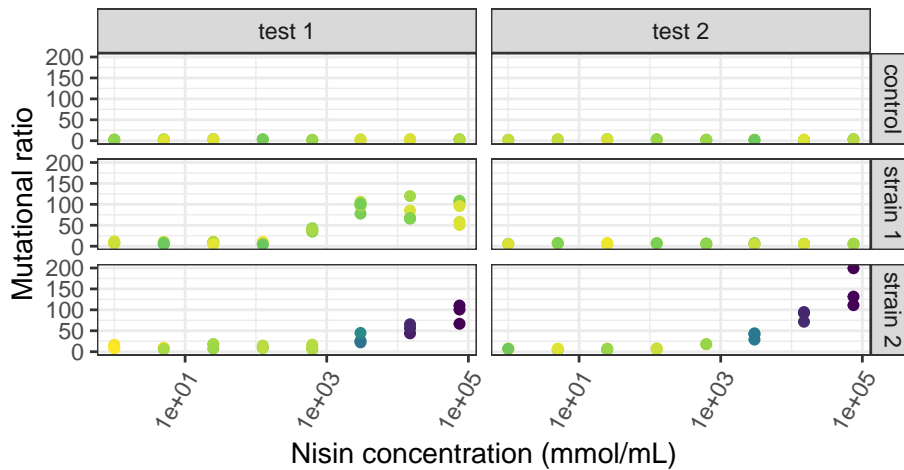
1361

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

1362

A very important experiment

So important it deserves a subtitle



1363

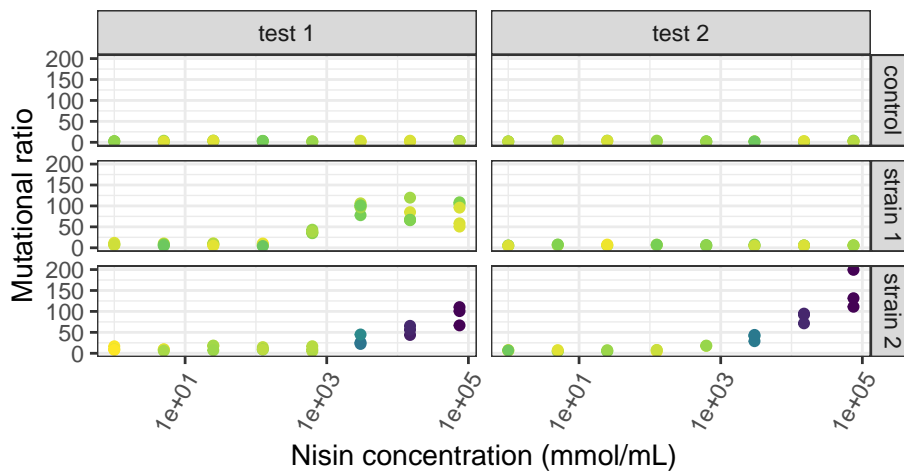
1364 And to selectively hide *some* legends, use guides:

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

1365

A very important experiment

So important it deserves a subtitle

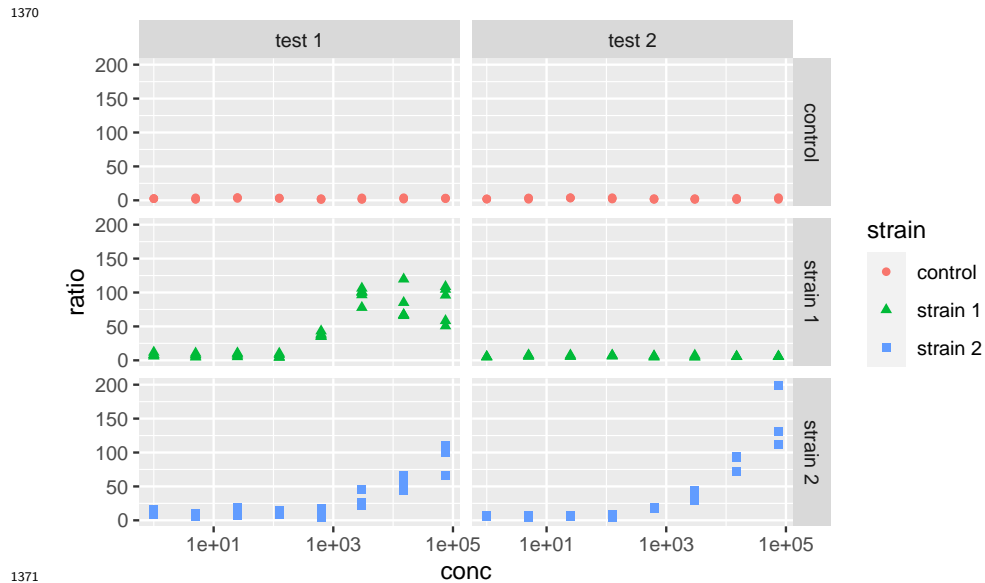


1366

1367 It is also important to remember that `ggplot2` will try to combine legends  
 1368 together whenever it can. If the same variable is mapped to two different aes-  
 1369 thetics, e.g. shape and color, only one legend will appear:

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

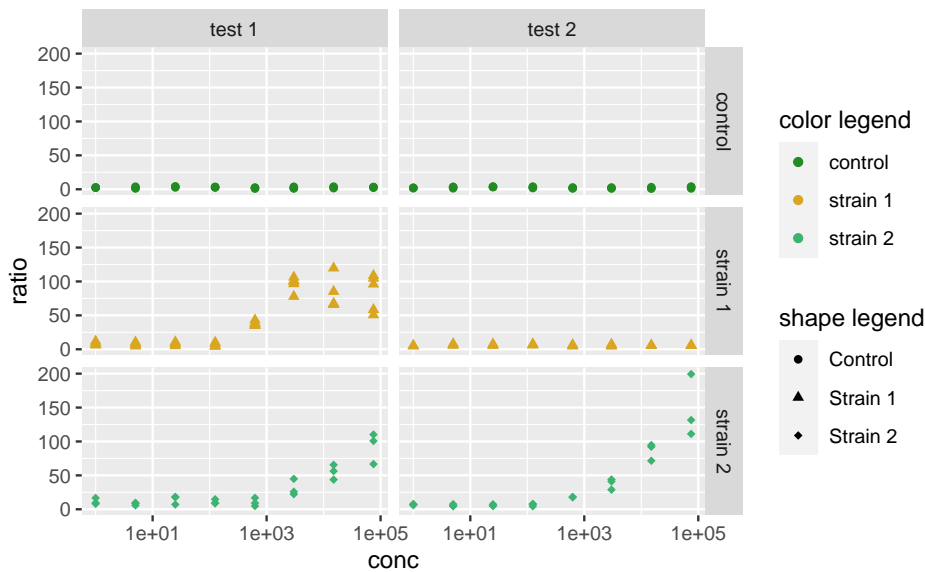
```
facet_grid(strain ~ assay) +
scale_x_log10()
```



1372 But this behavior can be controlled. You can use the arguments of the `scale_`  
 1373 functions to pass custom titles and labels to the legends. And if the legends  
 1374 mapping to the same variable have different titles or labels, they will be shown  
 1375 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")
  )
```





1377

1378 Note that you can also use this trick to combine different legends together, by  
 1379 giving them the same titles and labels.

## 1380 5.9 Combining plots

1381 This was more or less what you need to know to be operational when plotting  
 1382 *single* ggplots. But what if the facetting option is not enough, and you want to  
 1383 combine multiple plots into a single figure? `ggplot2` itself does not do that, but  
 1384 the good news is, there are many packages that do. Those include `patchwork`,  
 1385 `cowplot`, `grid`, `gridExtra`, `egg` or `aplot` (and probably more).

1386 One term that these packages often use is `grob`. A `grob` is a ggplot-like object,  
 1387 such as a `ggplot` but could also be a single text label in the middle of a plotting  
 1388 window. These packages essentially assemble grobs together.

1389 `patchwork` is personally my favorite so I will focus on this one here. It has  
 1390 the advantage to automatically align the frames of the different plots across the  
 1391 different subplots (I found that this is not entirely true when combining `ggtree`  
 1392 objects with other plots, `aplot` is better for this specific case). It also has an  
 1393 excellent, succinct documentation.

1394 Let us look at an example, where we assign the previous plot to `p1` and make a  
 1395 new plot to combine it with, called `p2`:

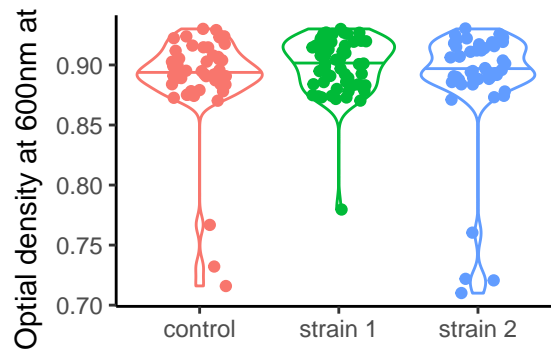
```
p1 <- p
p2 <- ggplot(data2, aes(x = strain, y = OD600_T1, color = strain)) +
  geom_violin(draw_quantiles = 0.5) +
  geom_jitter(width = 0.2) +
  theme_classic() +
```

```

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

```

1396

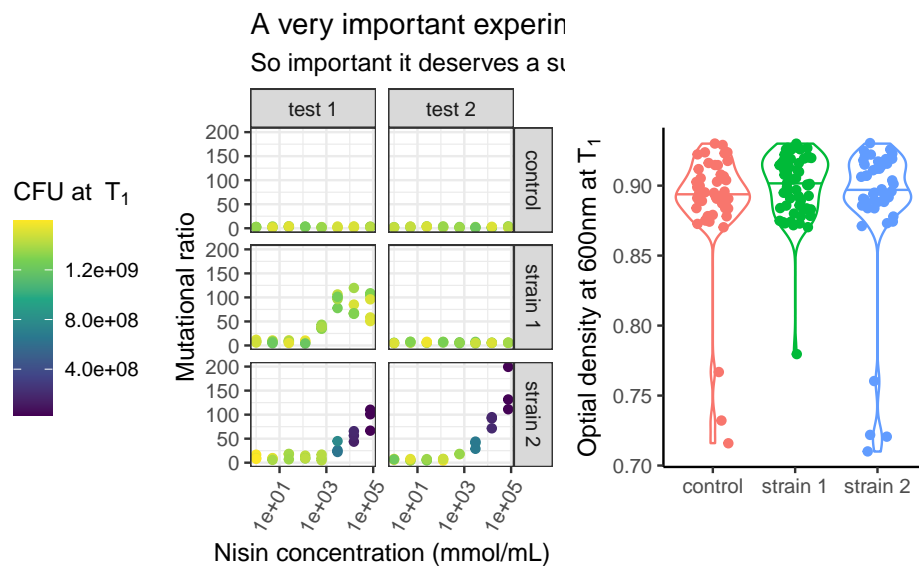
1397 In `patchwork`, we would combine both using:

```

library(patchwork)
p1 + p2

```

1398



1399

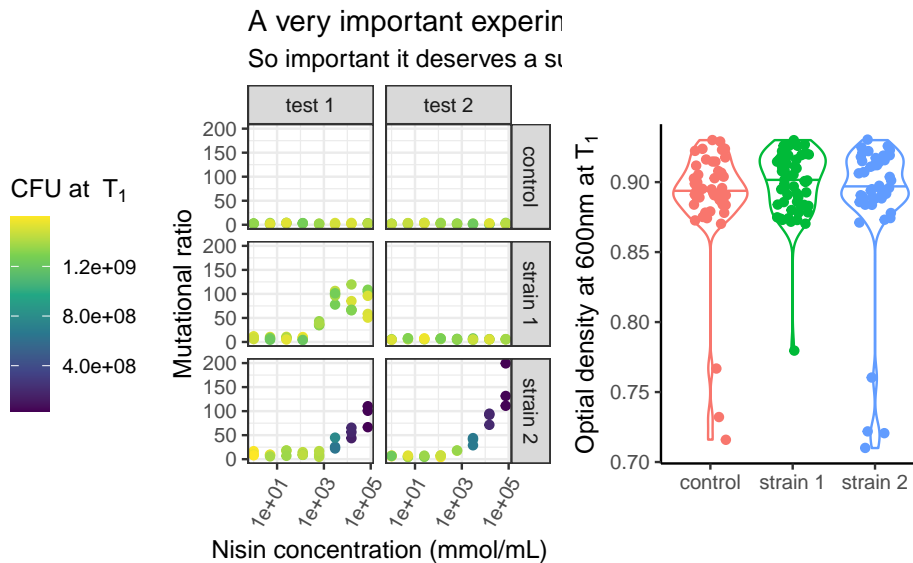
1400 `patchwork` uses operators such as `+`, `/` or `|` to assemble the plots in various  
 1401 layouts. It looks simple, but a caveat of this approach is that it may become  
 1402 tedious when assembling, e.g. 15 small plots, or plots from a list of unknown  
 1403 length. The programmatic equivalent of the above example is:

```

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

```

1404



1405

1406 More customization can be added to the previous combination of plots, such as  
 1407 layout specifications, e.g. controlling the position and dimension of the different  
 1408 plots, or annotations, e.g. global title, labelling each plot or capturing the leg-  
 1409 ends of all the plots and show it as one global legend). But this is a `ggplot2`  
 1410 tutorial and we just want you to know that `patchwork` and friends exist, so go  
 1411 check them out to know more about what they can do!

## 1412 5.10 Saving a plot

1413 Last but not least, ggplots have their own saving function: `ggsave` (it also works  
 1414 on combinations of ggplots made by `patchwork` or `cowplot`), which guesses the  
 1415 extension of your figure (e.g. `.png` or `.pdf`) from the file name you provide. You  
 1416 can also give it specific `width`, `height` and `dpi` (resolution) parameter values.

## 1417 5.11 High throughput plotting workflow

1418 As we mentioned in the part about combining plots, sometimes we want to do  
 1419 things many times (in my case I often make 100 times the same figure, just for  
 1420 different replicate simulations). Of course we would not copy and paste many  
 1421 times the same snippet of code, or write 100 times + to assemble some plots (by  
 1422 now we are advanced R users, after all). This is where we can make use, again,  
 1423 of the combination of tidyverse tools, and especially `purrr`.

1424 Let us make a function that plots the number of CFU against the optical density,  
 1425 faceted by time point (so, that function expects a time point-wise dataset, such  
 1426 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")
}

```

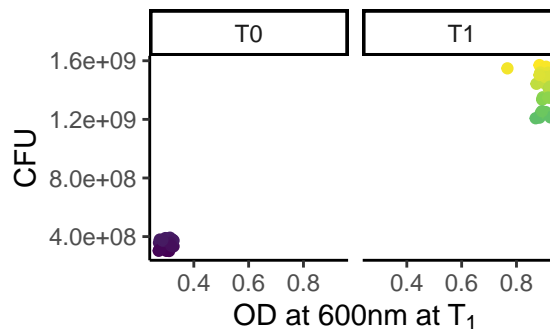
1427 Note that this does not plot anything, it is just a function that will if called on  
 1428 a dataset.

1429 The objective is to apply this function to each **strain-essay** combination, thus  
 1430 getting *one plot* per combination. We can check that this function works as  
 1431 expected for a single combination using our friend **dplyr**:

```

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

```



1432

1433 which works because **plot\_this** takes a data frame as first argument.

1434 Now that we are happy with our single-plot function, we **tidyr::nest** our  
 1435 data frame into all the relevant combinations of **strain** and **assay**, and we  
 1436 **purrr::map** through the resulting list-column to produce many ggplots in one  
 1437 go:

```

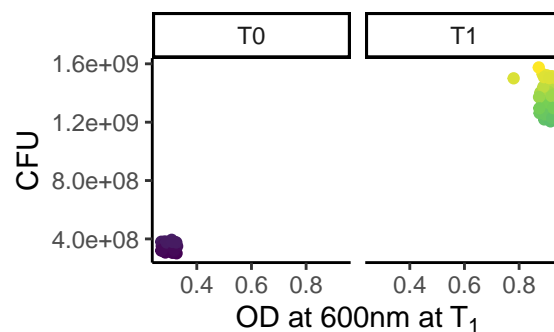
newdata <- data %>%
  group_by(assay, strain) %>%
  nest() %>%
  mutate(fig = map(data, plot_this))
newdata
#> # A tibble: 6 x 4
#> # Groups:   strain, assay [6]
#>   strain assay data fig

```

```
#>   <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>
```

1438 where the new list-column `fig` is a list of `ggplot` objects, that we can check  
1439 individually:

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



1440

1441 Looks purrrfect.

1442 If you ask yourself why going through this hassle with only two assays and three  
1443 strains, just think about a case where you would have hundreds of e.g. simula-  
1444 tions, sequences, field sites or study species.

1445 Let us go a bit further. Now we want to combine plots for each `strain` into one  
1446 figure per `assay`. We also want to give the resulting combined plot a figure file  
1447 name, and save all the figures. There we go:

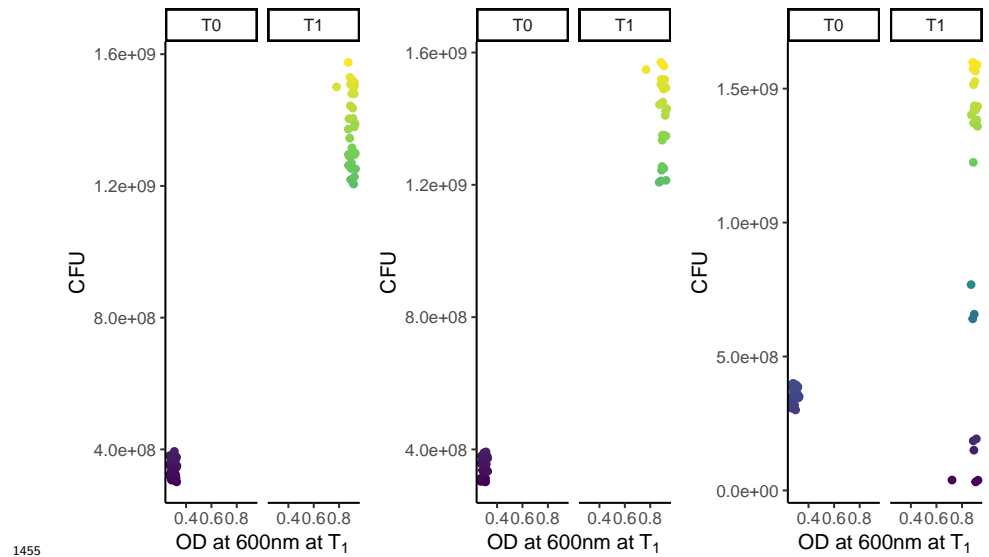
```
newdata <- newdata %>%
  select(-data) %>% # just to clean up a bit
  group_by(assay) %>%
  nest() %>%
  mutate(combifig = map(data, ~ wrap_plots(.x$fig)))
newdata
#> # A tibble: 2 x 3
#> # Groups:   assay [2]
#>   assay data          combifig
#>   <chr> <list>         <list>
#> 1 test 1 <tibble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

1448 Note that we use the *formula*-way of passing functions to `map` (using `~`), which is  
1449 more succinct than the *lambda* way (using an anonymous function `function(x)`)

1450 `wrap_plots(x)`), and where `.x` is interpreted as an element of the list we iterate  
 1451 through (here the list-column `data`). Please refer to the `purrr` documentation  
 1452 for more details.

1453 As we can see, we have created a new list-column `combifig`, filled with  
 1454 `patchwork` objects, i.e. combined plots:

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



1456 We could of course further customize the assembly of plots, but we refer the  
 1457 reader to the `patchwork` documentation for this.

1458 Last step, preparing file names and saving the figures, using old friends from  
 1459 the tidyverse:

```
library(glue)
newdata %>%
  mutate(filename = glue("data/figure_{str_replace(assay, ' ', '_')}.png")) %>%
  mutate(saved = walk2(filename, combifig, ggsave))
#> # A tibble: 2 x 5
#> # Groups:   assay [2]
#>   assay data          combifig filename          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~
```

## 1460 5.12 Want more?

1461 `ggplot2` is undoubtedly one of the largest chunks of the tidyverse. Here we  
 1462 tried to provide a global understanding of how it works, but we could not dig

1463 into all possible functions it has (this would take us days). Hopefully now you  
1464 are armed with the necessary knowledge to be able to find the missing pieces  
1465 you need.

1466 Some things, however, are missing from **ggplot2**. Fortunately, there are many of  
1467 extensions building on **ggplot2** that respect the same grammar. Some of them  
1468 implement new geoms (e.g. such as **ggbridges** for ridge-density plots, **ggradar**  
1469 for radial plots, or **gghalves** for mixes of geoms), others combine plots together  
1470 (examples cited above), offer more complex themes (e.g. **ggnewscale** for multi-  
1471 ple scales of the same type to coexist, or **ggdark** for a dark background), deal  
1472 with complicated objects that are not trivial to fit in data frames (e.g. **ggtree**  
1473 for tree-like objects or **ggraph** for networks), or provide shortcuts to quickly pro-  
1474 duce publication-ready figures for common plot layouts and their corresponding  
1475 statistical analyses (e.g. **ggpubr**, **ggrapid** or **GGally**). There are even packages  
1476 for animated graphics (**gganimate**), interactive plot building (**esquisse**) or 3D  
1477 surface plotting (**rayshader**). See the links below!

## 1478 5.13 References

- 1479 • The **ggplot2** website where you can find links to other resources
- 1480 • The **ggplot2** cheatsheet
- 1481 • The dedicated chapter in R for Data Science
- 1482 • A non-exhaustive list of extensions at this link
- 1483 • The R graph gallery for inspiration
- 1484 • Hadley's article explaining the grammar of graphics
- 1485 • The **patchwork** documentation
- 1486 • The **ggtree** and **ggraph** packages





## Chapter 6

Regular expressions and  
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 develop-

1502 ment of any code. It is estimated that 80-90% of all the time, we are debugging  
1503 our code. Being good at testing, is the way to become faster.

### 1504 6.1.3 Exercise: Spot the pattern

1505 Each line below is or contains a pattern. Observe what your brain does when  
1506 interpreting a line. If it thinks: ‘Hey, that’s a [pattern], because [regular expres-  
1507 sion]’, you got your answer!

1508 Answer, for each line: (1) what is it? (2) why did you think to?

1509 Lithium-ion batteries play a central role in the world of technology.  
1510 But would it be safe to simply orbit the planet?  
1511 Your Next Samsung Phone May Not Come With a Charger in the Box  
1512 The restrictions would end 90 days after Portland's state of emergency order lifts.  
1513 Those cities have limits at 15%.

### 1514 6.1.4 Answers: Spot the pattern

1515 Lithium-ion batteries play a central role in the world of technology.  
1516 This is a sentence, because it ends with a dot.  
1517 But would it be safe to simply orbit the planet?  
1518 This is a question, because it ends with a question mark.  
1519 Your Next Samsung Phone May Not Come With a Charger in the Box  
1520 This is a title, because most words start with an uppercase character.  
1521 The restrictions would end 90 days after Portland's state of emergency order lifts.  
1522 This is a sentence with a decimal number, because there is a sequence of char-  
1523 acters that consists out of numbers only.  
1524 Those cities have limits at 15%.  
1525 This is a sentence with a percentage, because there is a sequence of characters  
1526 that consists out of numbers only, with a percentage sign connected to it.  
1527 • Texts from <https://slashdot.org>.

### 1528 6.1.5 What are regular expressions?

1529 A regular expression ‘is a sequence of characters that define a search pattern’.  
1530 Such a pattern may be a zip code, a date, or any other text of which you can  
1531 say: ‘this is not just text, it is a [something]’.

1532 For example, take a Dutch zip code: 9747 AG. Dutch zip code have four digits,  
1533 a space and then two uppercase alphabet characters.

### 6.1.6 Applications

DNA data:

```
>KU215420.1|Felinecoronavirus|Feliscatus|Belgium|2013|Envelope
ATGATGTTTCCTAGGGCATTACTATCATAGATGACCATGGTATGGTTGTTAGTGCTCTC
>KP143511.1|Felinecoronavirus|Feliscatus|UnitedKingdom|2013|Envelope
ATGATGTTTCCTAGGGCATTACTATCATAGACGACCATGGTATGGTTGTTAGTGCTCTC
```

Protein data:

```
>sp|P0DTC2|SPIKE_SARS2 Spike glycoprotein OS=Severe acute respiratory syndrome coronavirus
MFVFLVLLPLVSSQCVNLTTRTQLPPAYTNSFTRGVYYPDKVFRSSVLHSTQDLFLPFFS
>sp|P0DTC5|VME1_SARS2 Membrane protein OS=Severe acute respiratory syndrome coronavirus
MADSNGTITVEELKKLLEQWNLVIGFLFTWICLLQFAYANRNRFLYIIKLIFLWLLWPV
```

### 6.1.7 Multiple in R



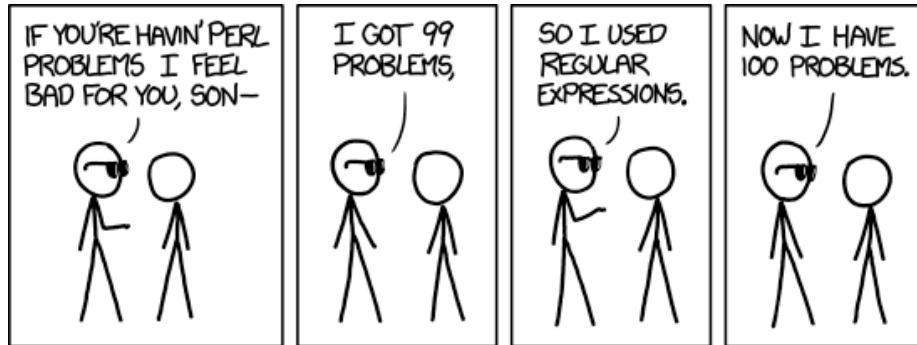
The ‘stringr’ logo. ‘stringr’ is part of the Tidyverse

R functions to work with regular expressions:

- `stringr::string_`
- `egrep`
- `gsub`

```
library(stringr)
```

### 6.1.8 Dangers of regexes



1554 'Perl problems', from <https://xkcd.com/1171/>

1555 Regexes have different dialects, such as POSIX and perl. Within R, there are  
1556 the base R dialect and the Tidyverse dialect.

1557 I define UNIX as 30 definitions of regular expressions living under  
1558 one roof.

1559 Donald Knuth. Digital Typography, ch. 33, p. 649 (1999)

1560 We'll have to test!

## 1561 6.2 Testing



1562

1563 From George Dinwiddie's blog, [http://blog.gdinwiddie.com/2012/](http://blog.gdinwiddie.com/2012/12/26/tdd-hat/)  
1564 12/26/tdd-hat/

### 1565 6.2.1 Why test?

1566 Testers don't like to break things; they like to dispel the illusion that  
1567 things work.

1568 Kaner, Bach, Pettichord

- 1569 • To be sure your code is correct
- 1570 • Spend less time fixing bugs
- 1571 • Unit of communication
- 1572 • Clean software interface

### 1573 6.2.2 Our first test

1574 The `testthat` package is the Tidyverse package to write tests.

```
library(testthat)
```

1575 All test functions start with `expect_`, for example:





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

## 1576 6.3 Detect a full match

1577 Here, we will detect simple patterns using `str_which`.

1578 Tip: run `?str_which` for its documentation.

## Detect Matches

	TRUE TRUE FALSE TRUE	<b>str_detect</b> (string, <b>pattern</b> ) Detect the presence of a pattern match in a string. <code>str_detect(fruit, "a")</code>
	1 2 4	<b>str_which</b> (string, <b>pattern</b> ) Find the indexes of strings that contain a pattern match. <code>str_which(fruit, "a")</code>
	0 3 1 2	<b>str_count</b> (string, <b>pattern</b> ) Count the number of matches in a string. <code>str_count(fruit, "a")</code>
	start and 2 4 4 7 NA NA 3 4	<b>str_locate</b> (string, <b>pattern</b> ) Locate the positions of pattern matches in a string. Also <b>str_locate_all</b> . <code>str_locate(fruit, "a")</code>

1579

1580 From 'Work with Strings Cheatsheet', [https://rstudio.com/](https://rstudio.com/resources/cheatsheets)  
 1581 resources/cheatsheets

### 1582 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)
)
```

### 1583 6.3.2 Example exercise: is\_a\_one

1584 Write a function called `is_a_one` that detects if a string is one one, using  
 1585 `str_subset`.

1586 Use the anchors as shown on the cheatsheet to specify that the complete string,  
 1587 from begin to the end, must consist out of characters

**regex**

**^a**

**a\$**

**matches**

start of string

end of string

1588

1589 From ‘Work with Strings Cheatsheet’, [https://rstudio.com/](https://rstudio.com/resources/cheatsheets)  
 1590 resources/cheatsheets

1591 These tests must pass:

```
1592 expect_true(is_a_one("1"))
1593 expect_false(is_a_one("X"))
1594 expect_false(is_a_one("11"))
1595 expect_false(is_a_one(c("1", "1")))
1596 expect_false(is_a_one(integer(0)))
1597 expect_false(is_a_one(NULL))
1598 expect_false(is_a_one(NA))
1599 expect_false(is_a_one(Inf))
```

#### 1600 6.3.2.1 Answer is\_a\_one

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

expect_true(is_a_one("1"))
expect_false(is_a_one("X"))
expect_false(is_a_one("11"))
expect_false(is_a_one(c("1", "1")))
expect_false(is_a_one(integer(0)))
expect_false(is_a_one(NULL))
expect_false(is_a_one(NA))
expect_false(is_a_one(Inf))
```

#### 1601 6.3.3 Exercise: is\_a\_digit

1602 Write a function called `is_a_digit` that detects if a string is one digit.

1603 Use the regex pattern as shown on the cheatsheet to specify a digit:





1604

1605 From 'Work with Strings Cheatsheet', [https://rstudio.com/](https://rstudio.com/resources/cheatsheets)  
 1606 resources/cheatsheets

1607 These tests must pass:

```

1608 expect_true(is_a_digit("0"))
1609 expect_true(is_a_digit("1"))
1610 expect_false(is_a_digit(""))
1611 expect_false(is_a_digit("X"))
1612 expect_false(is_a_digit(c("1", "2")))
1613 expect_false(is_a_digit(character(0)))
1614 expect_false(is_a_digit(NULL))
1615 expect_false(is_a_digit(NA))
1616 expect_false(is_a_digit(Inf))

```

1617 **6.3.3.1 Answer: is\_a\_digit**

```

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

expect_true(is_a_digit("0"))
expect_true(is_a_digit("1"))
expect_false(is_a_digit(""))
expect_false(is_a_digit("12"))
expect_false(is_a_digit("X"))
expect_false(is_a_digit(c("1", "2")))
expect_false(is_a_digit(character(0)))
expect_false(is_a_digit(NULL))
expect_false(is_a_digit(NA))
expect_false(is_a_digit(Inf))

```

1618 **6.3.4 Exercise: is\_a\_word**1619 Write a function called `is_a_word` that detects if a string is a word.

1620 To simplify now, a word is defined as:

- 1621 • Having one or more lowercase characters
- 1622 • Having no dashes, nor numbers

1623 Use the quantifiers as shown on the cheatsheet to specify that one needs one or  
 1624 more characters:

**regexp****matches****a?**

zero or one

**a\***

zero or more

**a+**

one or more

**a{n}**exactly **n****a{n, }****n** or more**a{n, m}**between **n** and **m**

1625

1626 From ‘Work with Strings Cheatsheet’, [https://rstudio.com/](https://rstudio.com/resources/cheatsheets)  
 1627 resources/cheatsheets

1628 These tests must pass:

```
1629 expect_true(is_a_word("a"))
1630 expect_true(is_a_word("an"))
1631 expect_true(is_a_word("apple"))
1632 expect_false(is_a_word("X"))
1633 expect_false(is_a_word("XX"))
1634 expect_false(is_a_word("Hi"))
1635 expect_false(is_a_word("hI"))
1636 expect_false(is_a_word("hoWdy"))
1637 expect_false(is_a_word(c("an", "apple")))
1638 expect_false(is_a_word(character(0)))
1639 expect_false(is_a_word(NULL))
1640 expect_false(is_a_word(NA))
1641 expect_false(is_a_word(Inf))
```

#### 1642 6.3.4.1 Answer: is\_a\_word

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

expect_true(is_a_word("a"))
expect_true(is_a_word("an"))
expect_true(is_a_word("apple"))
expect_false(is_a_word("X"))
expect_false(is_a_word("XX"))
expect_false(is_a_word("Hi"))
expect_false(is_a_word("hI"))
expect_false(is_a_word("hoWdy"))
expect_false(is_a_word(c("an", "apple")))
expect_false(is_a_word(character(0)))
expect_false(is_a_word(NULL))
expect_false(is_a_word(NA))
expect_false(is_a_word(Inf))
```

### 1643 6.3.5 Exercise: is\_dna\_sequence

1644 Write a function called `is_dna_sequence` that detects if a string is a DNA  
 1645 sequence.

1646 To simplify now, a DNA sequence is defined as:

- 1647 • There are four characters, one per nucleotides
- 1648 • These characters are uppercase (A, C, G and T)

1649 Use the alternates as shown on the cheatsheet to specify that each character  
 1650 must be one 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

1652 From ‘Work with Strings Cheatsheet’, [https://rstudio.com/](https://rstudio.com/resources/cheatsheets)  
 1653 [resources/cheatsheets](https://rstudio.com/resources/cheatsheets)

1654 These tests must pass:

```

1655 expect_true(is_dna_sequence("A"))
1656 expect_true(is_dna_sequence("AC"))
1657 expect_true(is_dna_sequence("ACG"))
1658 expect_true(is_dna_sequence("ACGT"))
1659 expect_false(is_dna_sequence("a"))
1660 expect_false(is_dna_sequence("Ax"))
1661 expect_false(is_dna_sequence("xA"))
1662 expect_false(is_dna_sequence("AxA"))
1663 expect_false(is_dna_sequence(c("A", "CGT")))
1664 expect_false(is_dna_sequence(character(0)))
1665 expect_false(is_dna_sequence(NULL))
1666 expect_false(is_dna_sequence(NA))
1667 expect_false(is_dna_sequence(Inf))

```

#### 1668 6.3.5.1 Answer: is\_dna\_sequence

```

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

expect_true(is_dna_sequence("A"))
expect_true(is_dna_sequence("AC"))
expect_true(is_dna_sequence("ACG"))
expect_true(is_dna_sequence("ACGT"))
expect_false(is_dna_sequence("a"))
expect_false(is_dna_sequence("Ax"))
expect_false(is_dna_sequence("xA"))
expect_false(is_dna_sequence("AxA"))
expect_false(is_dna_sequence(c("A", "CGT")))
expect_false(is_dna_sequence(character(0)))
expect_false(is_dna_sequence(NULL))

```

```
expect_false(is_dna_sequence(NA))
expect_false(is_dna_sequence(Inf))
```

## 6.4 Extract a pattern

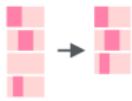
Here, we will extract a pattern using `str_match`.

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) ([^ ]+)")`

From ‘Work with Strings Cheatsheet’, <https://rstudio.com/resources/cheatsheets>

### 6.4.1 Context

Here we will work on a DNA sequence:

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

```
#> [9] "TTTGGCTCCTGTTGATAATTATATTGATATTGTTTTCAATAGCATTGCTAAATGTTATT"
#> [10] "AAGTTATGCATGGTATGTTGCAATTTGGGTAAGACTATTATAGTATTACCTGCACGCCAT"
```

1677 The data encoded in this text:

```
1678 >[DNA sequence number]|[virus name]|[host species name]|[country of host]|[year]|[prot
1679 [DNA sequence]
```

## 1680 6.4.2 str\_match

1681 `str_match` returns:

- 1682 • a matrix
- 1683 • a row per line of text, containing the match or an NA
- 1684 • a column per submatch (see later)

1685 For example, using a ‘everything’ pattern, we get:

```
matches <- stringr::str_match(text, ".*")
expect_is(matches, "matrix")
expect_equal(nrow(matches), length(text))
expect_equal(ncol(matches), 1)
head(matches)
#>      [,1]
#> [1,] ">KX722530.1/Felinecoronavirus/Feliscatus/Denmark/2015/Envelope"
#> [2,] "ATGATGTTTCCTAGGGCTTTTACTATCATAGATGACCATGGTATGGTTGTAAGCGTCTTC"
#> [3,] "TTCTGGCTCCTGTTGATAATTATATTGATATTGTTTTCAATAGCATTGCTAAATGTTATT"
#> [4,] "AAGTTATGCATGGTTTGTGCAATCTGGGTAAGACTATTATAGTACTACCTGCACGCCAT"
#> [5,] "GCATATGATGCCTACAAGACTTTTATGCAAATTAAGGCATATAATCCCGACGAAGCACTT"
#> [6,] "TTGGTTTGA"
```

1686 Using a pattern that is specific for the DNA sequence descriptors, we get NAs:

```
matches <- stringr::str_match(text, ">.*")
expect_is(matches, "matrix")
expect_equal(nrow(matches), length(text))
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
```

1687 Using round brackets, the matrix gives one extra column per sub-match. Here,  
1688 we select for all info after the >:

```

matches <- stringr::str_match(text, ">(.*?)")
expect_is(matches, "matrix")
expect_equal(nrow(matches), length(text))
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

```

1689 Select the second column:

```

matches <- matches[, 2]
expect_is(matches, "character")
expect_equal(length(matches), 180)
head(matches, n = 8)
#> [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

```

1690 Get rid of the NAs using purrr:

```

matches <- purrr::discard(matches, is.na)
expect_is(matches, "character")
expect_equal(length(matches), 30)
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"
```

1691 All of this in one go:

```
matches <- purrr::discard(
  stringr::str_match(text, ">(.*?)")[, 2],
  is.na
)
expect_equal(length(matches), 30)
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"
```

### 1692 6.4.3 Example exercise: extract\_dna\_sequence\_numbers

1693 Extract the DNA sequence numbers.

1694 Hint:

- 1695 • it is the text between > and |Felinecoronavirus.
- 1696 • Use \\| in your regex to indicate you want the pipe character ( as a|b is  
1697 the regex for ‘a or b’)

1698 These tests must pass:

```
1699 expect_equal(30, length(extract_dna_sequence_numbers(text)))
1700 expect_equal("KX722530.1", extract_dna_sequence_numbers(text)[1])
1701 expect_equal("KP143511.1", extract_dna_sequence_numbers(text)[30])
```

### 1702 6.4.4 Answer: extract\_dna\_sequence\_numbers

```
extract_dna_sequence_numbers <- function(text) {
  purrr::discard(
    stringr::str_match(text, ">(.*?)\\|Felinecoronavirus.*")[, 2],
    is.na
  )
}

expect_equal(30, length(extract_dna_sequence_numbers(text)))
expect_equal("KX722530.1", extract_dna_sequence_numbers(text)[1])
expect_equal("KP143511.1", extract_dna_sequence_numbers(text)[30])
```

1703 The regex ">(.\*?)\\|.\*" would not work, because the asterisk is *greedy*.



1704 **6.4.5 Example exercise: `extract_year`**

1705 Extract the year the DNA sequence has been obtained. Convert the text to a  
1706 number.

1707 Hint: the year is always 4 consecutive numbers, from 1993 to (and including)  
1708 2016.

1709 These tests must pass:

```
1710 expect_equal(30, length(extract_year(text)))  
1711 expect_equal(2015, extract_year(text)[1])  
1712 expect_equal(2013, extract_year(text)[30])
```

1713 **6.4.6 Answer: `extract_year`**

```
extract_year <- function(text) {  
  as.numeric(  
    purrr::discard(  
      stringr::str_match(text, ".*(?:digit:){4}).*")[, 2],  
      is.na  
    )  
  )  
}
```

```
expect_equal(30, length(extract_year(text)))  
expect_equal(2015, extract_year(text)[1])  
expect_equal(2013, extract_year(text)[30])
```

1714 Here, the simple regex worked, *because* the asterisk is greedy.

## 1715 6.5 Other functions

### 1716 6.5.1 Mutate

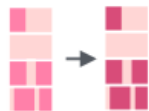
## Mutate Strings



**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", "-")`

1717

1718 From 'Work with Strings Cheatsheet', [https://rstudio.com/](https://rstudio.com/resources/cheatsheets)  
 1719 resources/cheatsheets

```
s <- "UnitedKingdom"
t <- stringr::str_replace(
  s,
  "([:upper:][:lower:]+)([:upper:][:lower:]+)",
  "\\1 \\2"
)
expect_equal("United Kingdom", t)
```

### 1720 6.5.2 testthat::expect\_match

1721 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)
}
```

1722 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"
```

1723 Take a look at `?testthat::expect_match`:

1724 Details

1725 `expect_match()` is a wrapper around `grepl()`. See its documentation  
1726 for more detail about the individual arguments.

1727 Use the base R regex dialect:

```
1728 expect_match(  
1729   get_version(),  
1730   "1\\.\\.\\.[:digit:]"  
1731 )
```

## 1732 6.6 Bigger picture

### 1733 6.6.1 Develop in packages

- 1734 • Also when ‘just’ doing data analysis

## 1735 6.7 Regex usage beyond R is common

1736 Command-line tools with regular expressions:

- 1737 • `grep`, `egrep`
- 1738 • `sed`
- 1739 • `dir/ls`

### 1740 6.7.1 Don’t overdo it



1742 ‘Regex Golf’, from <https://xkcd.com/1313/>

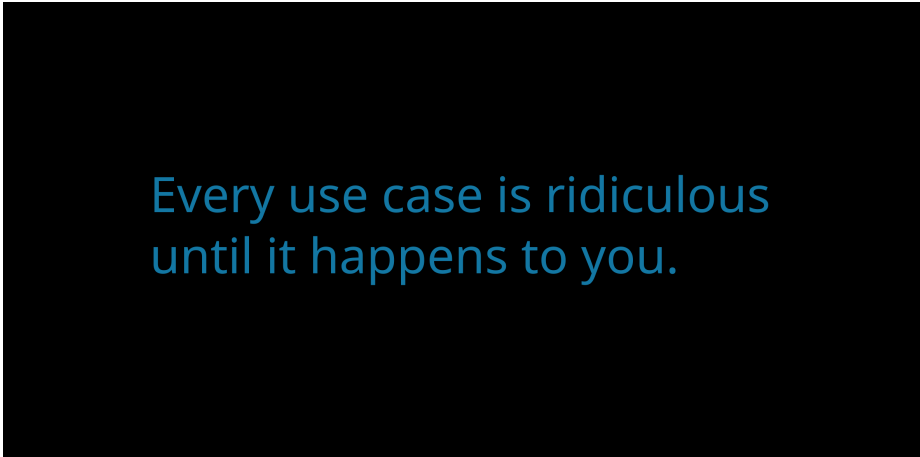
1743 Regex golf is a challenge to create the shortest regexes possible.

## 1744 6.8 Resources

- 1745 • RStudio cheatsheets, including the ‘Work with Strings Cheatsheet’

## 1746 Chapter 7

# 1747 Programming in the 1748 *tidyverse*



Every use case is ridiculous  
until it happens to you.

1749  
1750 Load the packages for the day.

```
library(tidyverse)  
library(rlang)
```

1751 A function to look at errors.

```
try_this <- function(ex) {  
  tryCatch(  
    expr = {  
      ex  
    },  
    error = function(e) {
```

```

    print(glue::glue(as.character(e), "\n"))
  }
)
}

```

## 1752 7.1 An explanation of the problem

### 1753 7.1.1 What the issue is

1754 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)

```

1755 Examine `small_mammals` and `small_mammals_too` to check whether they are  
1756 as expected.

```

# count rows
map_int(list(sm_1 = small_mammals, sm2 = small_mammals_too),
        nrow)
#> sm_1 sm2
#> 4381  0

```

1757 The difference in the number of rows is because `dplyr::filter` could not un-  
1758 derstand the string `"Mass.g"` as a variable in the dataframe.

1759 This is because the `tidyverse`, through its `tidyselect` package, makes a dis-  
1760 tinction between `"Mass.g"`, and `Mass.g`.

1761 A better explanation of (some of) the theory behind this can be found here:  
1762 Programming with `dplyr`.

1763 The same issue arises with functions such as `dplyr::summarise` and  
1764 `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

```

### 1765 7.1.2 Why the issue is a problem

1766 Consider an analysis pipeline as follows.

1767 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

```

1768 Now consider that this analysis pipeline is repeated many times in your docu-  
 1769 ment. Consider also that a well intentioned person has renamed the dataframe  
 1770 columns.

```

data <- data %>%
  `colnames<-`(str_replace_all(colnames(data), "\\.", "_") %>%
               str_to_lower %>%
               str_remove("_1_2"))

```

1771 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.

```

1772 This illustrates the problem in part: when the columns to be operated upon are  
 1773 *unknown to the programmer*, much of basic `tidyverse` code cannot be gener-  
 1774 alised to be used with any dataframe.

### 1775 7.1.3 Passing variables as strings is (also) an issue

1776 The variables to be operated on could be given as strings, perhaps as the ar-  
 1777 gument to a function, or as a global variable. This way, a single global vector  
 1778 could contain the grouping variables for all further `summarise` procedures.

1779 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.

```

1780 In the case of a standard `filter %>% group %>% summarise` pipeline, the func-  
 1781 tion's operations are evident. It must filter a dataframe based on a/some col-  
 1782 umn(s), and then summarise by groups. The filter to be applied, the variables  
 1783 to group by, and the variables to be summarised should be passed as function  
 1784 arguments — just how this is to be done is not immediately obvious.

## 1785 7.2 Flexible selection is easy

1786 Selection often precedes data operations, but is not part of the pipeline dealt  
 1787 with further.

1788 This is because `dplyr::select` appears to work on both quoted and unquoted  
 1789 variables, but in general some useful `select` helpers such as `dplyr::all_of`  
 1790 should be used. These straightforward helper functions significantly expand



1791 `select`'s flexibility and ease of use, and are not covered here. See the `select`  
 1792 help for more information.

## 1793 7.3 A first attempt at a flexible function

1794 The attempt below to write such a function, which gives the mean and confidence  
 1795 intervals 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))

}
```

### 1796 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`.
```

1797 This function initially failed because `filter` could not find `mass_g` in the  
 1798 dataframe. This is because `mass_g` is treated as an independent R object, while  
 1799 the function should instead treat it as a variable in a dataframe.

1800 The difference between so-called `data` and `environment` variables is explained  
 1801 better at the `rlang` and `tidyeval` websites and tutorials linked at the end of  
 1802 this chapter. It is this difference that prevents `filter` from correctly interpreting  
 1803 `mass_g`.

### 7.3.2 Passing arguments as strings doesn't help

The example below tries to get `filter` to work. What could be tried? One option is to 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`.
```

While this doesn't work, it is on the right track, which is that the `filters` argument needs some extra work beyond changing the type.

### 7.3.3 None of the other arguments will be successful

`filter` was the first failure, after which it stopped further evaluation, but none of the steps of the custom function would have worked, for the same reason `filter` would not have worked: all the arguments need some work before they can be passed to their respective functions.

## 7.4 Flexible filtering in a function

The first thing to try is to change how `filter` uses the argument passed to it. Here, the argument `filters` is passed as a character vector, and is set by default to filter out mammals with masses below 1 kg.

The argument could be passed as a list, but the `rlang::parse_exprs` function works on vectors, not lists. The conversion between them is trivial for single level lists with atomic types (`purrr::as_vector`).

### A brief detour: Expressions in R

A full explanation of R works under the hood would take a very long time. A working knowledge of how this working can be exploited is usually sufficient to use most of R's functionality.

R expressions are one such. They represent a promise of R code, but without being evaluated. Any string can be parsed (interpreted) as an R expression.

What does `rlang::parse_exprs` do? It interprets a string as an R command. This expression can then be evaluated later. Consider the following, where `a` is assigned the numeric 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

```

1831 Here, `a + 3` was converted to an expression in the second command, and only  
 1832 evaluated in the third.

### 1833 Unquoting with !!!

1834 R expressions underlie R code. Their evaluation can be forced inside another  
 1835 function using the special operators `!!` and `!!!`, for single and multiple R  
 1836 expressions respectively.

#### 1837 7.4.1 Flexible filtering using expressions

1838 Consider the case where mammals below 1 kg body mass are to be excluded.  
 1839 The `dplyr` code would look like this:

```
1840 filter(data, mass_g > 1000)
```

1841 This fixes both the variable to be filtered by, as well as the cut-off value. This  
 1842 can be made 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)
}

```

1843 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

```

1844 The function `filter` correctly processes the string passed to filter the data.

## 1845 7.5 Flexible grouping in a function

1846 Just as the exact filtering approach can be controlled from a single string vector  
 1847 in the example above, the grouping variables can also be stored and passed  
 1848 as arguments using the `...` (dots) argument. Dots are a convenient way of  
 1849 referring to all unnamed arguments of a function. Here, they are used to accept  
 1850 the grouping variables.

### 1851 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)
}

```

1852 Try the function again, and check the grouping variables.

```

custom_summary(data,
               filters = c("mass_g > 1000"),
               order, family) %>%

  group_vars()
#> [1] "order" "family"

```

### 1853 7.5.2 Passing grouping variables as strings

1854 In the previous example, the grouping variables were passed as unquoted vari-  
 1855 ables, then `enquo`-ted and parsed, after which they were applied. An alternative  
 1856 way of passing arguments to a function is as a string vector, i.e., `grouping_vars`  
 1857 = `c("var_a", "var_b")`.

1858 This can be done by interpreting the string vector as R symbols using  
 1859 `rlang::syms`. It could also be done by treating them as a full expression using  
 1860 the previously covered `rlang::parse_exprs`. However, both methods must use  
 1861 an unquoting-splice (`!!!`), i.e., force the evaluation of a list of R expressions.

### 1862 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.

```

#### 1863 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

```

## 1864 7.6 Flexible summarising in a function

1865 Summarising using string expressions has been around in the `tidyverse` for a  
 1866 very long time, and `summarise_at` is a function most users are familiar with,  
 1867 along with its variants `summarise_if`, `summarise_all`

### 1868 7.6.1 Using `dplyr::summarise_at`

1869 Simply pass a string vector to the `.vars` argument of `summarise_at`, while  
 1870 passing a list, 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
```

## 1871 7.6.2 Using the `across` argument for summary variables

1872 `dplyr` 1.0.0 had `summarise_*` superseded by the `across` argument to  
 1873 `summarise`. This works somewhat differently. The example below shows how  
 1874 the mean of a trait of mammal groups can be found.

1875 This example makes use of embracing using `{{ }}`, where the double curly  
 1876 braces indicate a promise, i.e., an expectation that such a variable will exist in  
 1877 the function environment.

```
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
```

1878 `across` also accepts multiple functions just as `summarise_*` did. This works as  
 1879 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

```

### 7.6.3 Summarise multiple variables using ...

Here, the unquoted and unnamed variables passed to the function are captured by ... and enquos-ed, i.e. their evaluation is delayed. Then the variables are forcibly evaluated within the mean function, and this expression is captured using expr. Since there are 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

```

#### 1886 **expr and enqu**

1887 **expr** and **enquo** are essentially the same, defusing/quoting (delaying evaluation)  
 1888 of R code. **expr** works on expressions supplied by the primary user, while **enquo**  
 1889 works on arguments passed to a function. When in doubt, ask whether the  
 1890 expression to be quoted has entered the function environment as an argument.  
 1891 If yes, use **enquo**, and if not **expr**. The plural forms **enquos** and **exprs** exist for  
 1892 multiple arguments.

#### 1893 **7.6.3.1 Correct the names of summary variables**

1894 The example above returns summary variables that are not assigned a name.  
 1895 The **enquos** function can assign the name from the variable names, so  
 1896 **mean(mass\_g)** is returned as **mass\_g**. Since it is useful to add a tag to make  
 1897 clear what the summary variable is (mean, variance etc.) an extra **glue** step is  
 1898 added to assign informative names to the summary 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

```

#### 1899 7.6.4 Summarise with multiple functions

1900 The final step is to pass multiple summary functions to the summary variables.  
 1901 Unlike the earlier example using `summarise(across(vars, funs))`, the goal  
 1902 here is to apply one function to each variable.

1903 This is done by passing the functions and the variables on which they should op-  
 1904 erate as strings, and using string interpolation via `glue` to construct a coherent  
 1905 R expression. This 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

1906

1907

1908

1909

1910

- **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/>