

TRES Tidyverse Tutorial

Raphael, Pratik and Theo

2020-06-03

Contents

2	Outline	5
3	About	5
4	Schedule	5
5	Possible extras	5
6	Join	6
7	1 Reading files and string manipulation	7
8	1.1 Data import and export with <code>readr</code>	7
9	1.2 String manipulation with <code>stringr</code>	10
10	1.3 String interpolation with <code>glue</code>	19
11	1.4 Strings in <code>ggplot</code>	20
12	2 Reshaping data tables in the tidyverse	23
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>	29
16	2.4 Extra: factors and the <code>forcats</code> package	35
17	2.5 External resources	39
18	3 Data manipulation with <code>dplyr</code>	41
19	3.1 Introduction	41
20	3.2 Example data of the day	41
21	3.3 Select variables with <code>select()</code>	44
22	3.4 Select observations with <code>filter()</code>	44
23	3.5 Create new variables with <code>mutate()</code>	44
24	3.6 Grouped results with <code>group_by()</code> and <code>summarise()</code>	44
25	3.7 Scoped variables	44
26	3.8 More !	44
27	4 Working with lists and iteration	45
28	4.1 Iteration with <code>map</code>	45
29	4.2 More <code>map</code> variants	49
30	4.3 Working with lists	53

Outline

This is the readable version of the TRES tidyverse tutorial.

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	03/07/20
Programming with the tidyverse	rlang	Pratik	10/07/20

Possible extras

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

⁴⁷ **Join**

⁴⁸ Join the Slack by clicking this link (Slack account required).

⁴⁹ *Tentative dates.

50 Chapter 1

51 Reading files and string 52 manipulation



Every use case is ridiculous
until it happens to you.

53
54 Load the packages for the day.

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

55 1.1 Data import and export with readr

56 Data in the wild with which ecologists and evolutionary biologists deal is most often in
57 the form of a text file, usually with the extensions .csv or .txt. Often, such data has to be
58 written to file from within R. readr contains a number of functions to help with reading
59 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)

## Parsed with column specification:
## cols(
##   mpg = col_double(),
##   cyl = col_double(),
##   disp = col_double(),
##   hp = col_double(),
##   drat = col_double(),
##   wt = col_double(),
##   qsec = col_double(),
##   vs = col_double(),
##   am = col_double(),
##   gear = col_double(),
##   carb = col_double()
## )

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

94 skipped before reading data.

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

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

99 1.1.2 Writing data

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

104 These functions are not covered here.

105 1.1.3 Reading and writing lines

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

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

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

112 The model summary can be written to file. When writing lines to file, BE AWARE OF THE
113 DIFFERENCES BETWEEN UNIX AND WINODWS line separators. Usually, this causes no
114 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")
```

115 This model output can be read back in for display, and each line of the model output is an
116 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")
```

```

117 ##
118 ## Call:
119 ## lm(formula = mpg ~ wt, data = mtcars)
120 ##
121 ## Residuals:
122 ##      Min       1Q   Median       3Q      Max
123 ## -4.5432 -2.3647 -0.1252  1.4096  6.8727
124 ##
125 ## Coefficients:
126 ##              Estimate Std. Error t value Pr(>|t|)
127 ## (Intercept)  37.2851     1.8776  19.858 < 2e-16 ***
128 ## wt          -5.3445     0.5591  -9.559 1.29e-10 ***
129 ## ---
130 ## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
131 ##
132 ## Residual standard error: 3.046 on 30 degrees of freedom
133 ## Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
134 ## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-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 = "_")

```

```

152 ## [1] "this string_this other string"
    # str_dup works like rep
    str_dup("this string", times = 3)
153 ## [1] "this stringthis stringthis string"
    # str_flatten works on lists and vectors
    str_flatten(string = as.list(letters), collapse = "_")
154 ## [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 = "-")
155 ## [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"
156 str_flatten is especially useful when displaying the type of an object that returns a list
157 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 = ", ")
158 ## [1] "tbl_df, tbl, data.frame"

```

1.2.2 Detecting strings

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

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")
164 ## [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")
165 ## [1] 5 3
    # vectorise over the pattern to count both a-s and b-s
    str_count(string = "ababababa", pattern = c("a", "b"))
166 ## [1] 5 4
167 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"))

```

```

168 ## [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"))

169 ## [1] 5 1 4 3

170 str_locate locates the search pattern in a string, and returns the start and end as a two
171 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")

172 ##      start end
173 ## [1,]      2  4

# str_detect detects a sequence in a string
str_detect(string = "Bananageddon is coming!",
           pattern = "na")

174 ## [1] TRUE

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

175 ## [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")))

176 ## [1] TRUE

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

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

179 ## [1] FALSE FALSE  TRUE  TRUE

# str_ends looks at the last character
str_ends(fruit, "e")

180 ## [1]  TRUE FALSE FALSE  TRUE

```

```

# an example of negate = TRUE
str_ends(fruit, "e", negate = TRUE)

## [1] FALSE TRUE TRUE FALSE

str_subset[WHICH IS NOT RELATED TO str_sub] helps with subsetting a character vec-
tor based on a str_detect predicate. In the example, all elements containing "banana"
are subset.

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

# 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"
```

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

The return type is a list, with one element per input string. Each element is a character matrix, where each row is one possible match, and each column after 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})(\\-|\\/)([a-z]+)")

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

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

## [[1]]
```

```

226 ##      [,1]      [,2] [,3]
227 ## [1,] "1970-somemonth" "1970" "somemonth"
228 ## [2,] "1990/anothermonth" "1990" "anothermonth"
229 ##
230 ## [[2]]
231 ##      [,1]      [,2] [,3]
232 ## [1,] "1990-somemonth" "1990" "somemonth"
233 ## [2,] "2001/anothermonth" "2001" "anothermonth"

```

234 1.2.4 Simpler pattern extraction

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

238 `str_extract` returns a character vector with the same length as the input string vector,
 239 while `str_extract_all` returns a list, with a character vector whose elements are the
 240 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]+)")

241 ## [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]+)")

242 ## [[1]]
243 ## [1] "1970-somemonth" "1990/anothermonth"
244 ##
245 ## [[2]]
246 ## [1] "1990-somemonth" "2001/anothermonth"

```

247 1.2.5 Breaking strings apart

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

251 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 = "[\\-|\\/]")

```

```

252 ## [[1]]
253 ## [1] "1970"      "somemonth" "01"
254 ##
255 ## [[2]]
256 ## [1] "1990"      "anothermonth" "01"

257 This can be useful in recovering simulation parameters from a filename, but may require
258 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 = "_")

259 ## [[1]]
260 ## [1] "sim"      "param1"  "0.01"    "param2"  "0.05"    "param3"  "0.01.ext"

# not really
str_split(filename,
           pattern = "sim_")

261 ## [[1]]
262 ## [1] ""
263 ## [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)")

264 ## [[1]]
265 ## [1] ""      ""      "0.01" "0.05" "0.01" ""

266 str_split_fixed split the string into as many pieces as specified, and can be especially
267 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)

268 ##      [,1]      [,2]
269 ## [1,] "dir_level_1" "dir_level_2/file.ext"

```

270 1.2.6 Replacing string elements

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


```

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

274 ## [1] " 0.01 0.05 0.01 "

275 str_remove is a wrapper around str_replace where the replacement is set to "". This
276 is not covered here.

277 Having replaced unwanted characters in the filename with spaces, str_trim offers a way
278 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

279 ## [1] "0.01 0.05 0.01"

# the result can be split on whitespaces to return useful data
str_split(filename_without_spaces, " ")

280 ## [[1]]
281 ## [1] "0.01" "0.05" "0.01"

```

282 1.2.7 Subsetting within strings

283 When strings are highly regular, useful data can be extracted from a string using `str_sub`.
 284 In the date-time example, the year is always represented by the first four characters.

```

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

285 ## [1] "1970" "1990" "2010"

286 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)

287 ## [1] "01" "21" "31"

```

288 Finally, it's also possible to replace characters within a string based on the position. This
 289 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
290 ## [1] "1970-somemonth-01" "1990-anothermonth-01" "2010-thismonth-01"
```

291 1.2.8 Padding and truncating strings

292 Strings included in filenames or plots are often of unequal lengths, especially when they
 293 represent numbers. `str_pad` can pad strings with suitable characters to maintain equal
 294 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")
295 ## [1] "001" "010" "100"
```

296 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.")
297 ## [1] "bananas are great etc. etc."
```

298 1.2.9 Stringr aspects not covered here

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

- 300 • `str_wrap` (of dubious use),
- 301 • `str_interp`, `str_glue*` (better to use `glue`; see below),
- 302 • `str_sort`, `str_order` (used in sorting a character vector),
- 303 • `str_to_case*` (case conversion), and

304 • `str_view*` (a graphical view of search pattern matches).
 305 • `word`, `boundary` etc. The use of `word` is covered below.
 306 `stringi`, of which `stringr` is a wrapper, offers a lot more flexibility and control.

307 1.3 String interpolation with glue

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

310 `glue` is as simple as the example shown.

```
311 # print that each car name is a car model
312 cars = rownames(head(mtcars))
313 glue('The {cars} is a car model')
314 ## The Mazda RX4 is a car model
315 ## The Mazda RX4 Wag is a car model
316 ## The Datsun 710 is a car model
317 ## The Hornet 4 Drive is a car model
318 ## The Hornet Sportabout is a car model
319 ## The Valiant is a car model
```

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

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

```
320 # use dataframes for now
321 parameter_combinations = data.frame(param1 = letters[1:5],
322                                     param2 = 1:5)
323
324 # for command line arguments or to start multiple job scripts on the cluster
325 glue_data(parameter_combinations,
326           'simulation-name {param1} {param2}')
327
328 ## simulation-name a 1
329 ## simulation-name b 2
330 ## simulation-name c 3
331 ## simulation-name d 4
332 ## simulation-name e 5
333
334 # for filenames
335 glue_data(parameter_combinations,
336           'sim_data_param1_{param1}_param2_{param2}.ext')
337
338 ## sim_data_param1_a_param2_1.ext
339 ## sim_data_param1_b_param2_2.ext
340 ## sim_data_param1_c_param2_3.ext
```

```
328 ## sim_data_param1_d_param2_4.ext
329 ## sim_data_param1_e_param2_5.ext
```

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

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

335 1.4 Strings in ggplot

336 `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.

339 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.

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

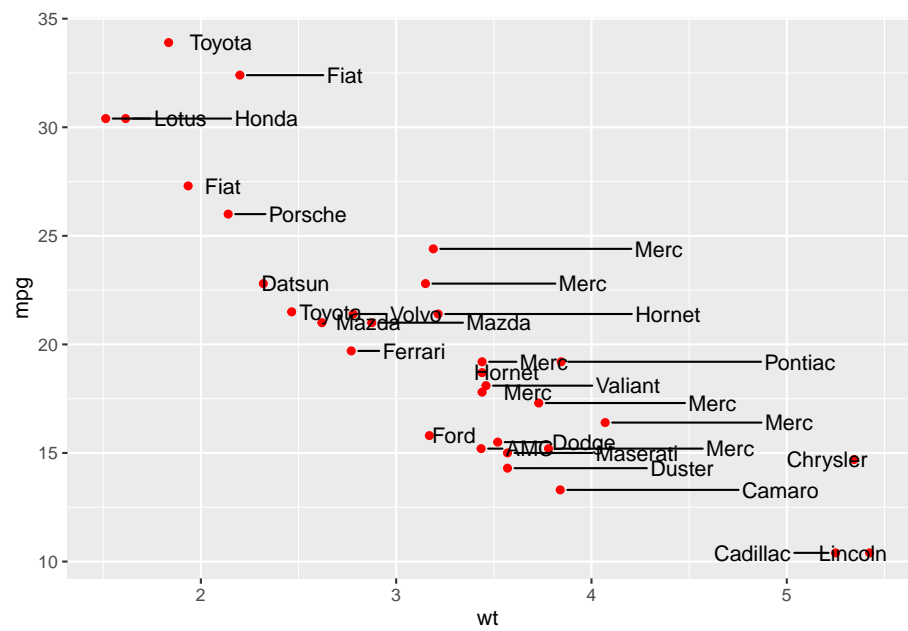
344 Here, the arguments to `geom_text_repel` are taken both from the `mtcars` data (position), as well as from the car brands extracted using the `stringr::word` (labels), which tries to separate strings based on a regular pattern.

347 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)
```



348

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

Chapter 2

Reshaping data tables in the tidyverse

Raphael Scherrer

Every use case is ridiculous
until it happens to you.

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

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

2.1 The new data frame: tibble

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

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

```
##      who chapt
## 1 Pratik  1, 4
## 2  Theo   3
## 3  Raph  2, 5
```

```
# Or an equivalent tibble
tibble(who = c("Pratik", "Theo", "Raph"), chapt = c("1, 4", "3", "2, 5"))
```

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

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

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

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

```
tribble(
  ~who, ~chapt,
  "Pratik", "1, 4",
  "Theo", "3",
  "Raph", "2, 5"
)
```

```
## # A tibble: 3 x 2
##   who    chapt
##   <chr> <chr>
## 1 Pratik 1, 4
## 2 Theo   3
```


[illegible]

```

425 ## 1 -79.6    2.13 -2.15 -2.71 -0.702 -0.315 -0.0987 -0.0779 -0.200 -
426 0.290
427 ## 2 -79.6    2.15 -2.22 -2.18 -0.884 -0.453 -0.00355 -0.0957 -0.353 -
428 0.193
429 ## 3 -134.    -5.06 -2.14  0.346  1.11  1.17    0.00576  0.136  -
430 0.198 0.0763
431 ## 4  8.52  45.0  1.23  0.827  0.424 -0.0579 -0.0243  0.221  0.356 -
432 0.0906
433 ## 5 129.    30.8  3.34 -0.521 0.737 -0.333  0.106 -0.0530 0.153 -0.189
434 ## 6 -23.2  35.1 -3.26  1.40  0.803 -0.0884 0.239  0.424  0.101 -
435 0.0377
436 ## 7 159.   -32.3  0.649 0.199 0.786 0.0687 -0.530 -0.0593 0.221 -
437 0.313
438 ## 8 -113.   39.7 -0.465 0.338 -1.24 0.280 -0.146  0.320  0.279 0.190
439 ## 9 -104.    7.51 -1.59 4.02 -1.14 0.0279 0.595 -0.233 -0.126 -0.349
440 ## 10 -67.0    -6.21 -3.61 -0.320 -0.960 -0.529 -0.0174  -
441 0.182  0.543  0.412
442 ## # ... with 22 more rows, and 1 more variable: PC11 <dbl>

```

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

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

448 2.2 The concept of tidy data

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

454 Say we have a dataset of several morphometrics measured on Darwin’s finches in the Gala-
 455 pagos 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,
  beak_length = beak_lengths,

```

```

    beak_width = beak_widths,
    body_weight = body_weights,
    island = islands
  )

```

```
# Snapshot
```

```
data
```

```

456 ## # A tibble: 100 x 5
457 ##       id beak_length beak_width body_weight island
458 ##   <int>      <dbl>      <dbl>      <dbl> <chr>
459 ## 1     1         5.18         2.02        10.3  Isabela
460 ## 2     2         4.97         1.83        17.5  Isabela
461 ## 3     3         4.86         2.09         6.34  Isabela
462 ## 4     4         5.05         2.04         3.44  Isabela
463 ## 5     5         4.95         1.91         9.50  Isabela
464 ## 6     6         4.97         1.96        10.8  Isabela
465 ## 7     7         5.13         2.03        18.8  Isabela
466 ## 8     8         5.03         1.96        10.7  Isabela
467 ## 9     9         5.08         1.94        15.9  Isabela
468 ## 10    10         4.80         2.02         6.28  Isabela
469 ## # ... with 90 more rows

```

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

475 The tidy equivalent of this dataset would be:

```

data <- pivot_longer(
  data,
  cols = c("beak_length", "beak_width", "body_weight"),
  names_to = "variable"
)
data

```

```

476 ## # A tibble: 300 x 4
477 ##       id island variable    value
478 ##   <int> <chr>   <chr>      <dbl>
479 ## 1     1  Isabela beak_length  5.18
480 ## 2     1  Isabela beak_width   2.02
481 ## 3     1  Isabela body_weight 10.3
482 ## 4     2  Isabela beak_length  4.97
483 ## 5     2  Isabela beak_width   1.83
484 ## 6     2  Isabela body_weight 17.5
485 ## 7     3  Isabela beak_length  4.86

```

```

486 ## 8      3 Isabela beak_width  2.09
487 ## 9      3 Isabela body_weight 6.34
488 ## 10     4 Isabela beak_length 5.05
489 ## # ... with 290 more rows

```

490 where each *measurement* (and not each *individual*) is now the unit of observation (the rows).
 491 We will come back to the `pivot_longer` function later.

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

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

504 Another common example of wide-or-long dilemma is when dealing with *contingency ta-*
 505 *bles*. This would be our case, for example, if we asked how many observations we have for
 506 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

507 ##           variable
508 ## island    beak_length beak_width body_weight
509 ##  Isabela           50           50           50
510 ##  Santa Cruz          50           50           50

```

511 A variety of statistical tests can be used on contingency tables such as Fisher's exact test,
 512 the chi-square test or the binomial test. Contingency tables are in the wide format by con-
 513 struction, but they too can be pivoted to the long format, and the tidyverse manipulation
 514 tools will expect you to do so. Actually, `tibble` knows that very well and does it by default
 515 if you convert your `table` into a tibble:

```

# Contingency table is pivoted to the long-format automatically
as_tibble(ctg)

516 ## # A tibble: 6 x 3
517 ##   island    variable     n
518 ##   <chr>    <chr>    <int>
519 ## 1 Isabela beak_length  50
520 ## 2 Santa Cruz beak_length  50
521 ## 3 Isabela  beak_width   50

```

```

522 ## 4 Santa Cruz beak_width      50
523 ## 5 Isabela   body_weight      50
524 ## 6 Santa Cruz body_weight      50

```

525 2.3 Reshaping with tidyr

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

535 2.3.1 Pivoting

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

```

pivot_wider(
  data,
  names_from = "variable",
  values_from = "value",
  id_cols = c("id", "island")
)

```

```

540 ## # A tibble: 100 x 5
541 ##       id island beak_length beak_width body_weight
542 ##   <int> <chr>      <dbl>      <dbl>      <dbl>
543 ## 1     1 Isabela      5.18      2.02      10.3
544 ## 2     2 Isabela      4.97      1.83      17.5
545 ## 3     3 Isabela      4.86      2.09      6.34
546 ## 4     4 Isabela      5.05      2.04      3.44
547 ## 5     5 Isabela      4.95      1.91      9.50
548 ## 6     6 Isabela      4.97      1.96      10.8
549 ## 7     7 Isabela      5.13      2.03      18.8
550 ## 8     8 Isabela      5.03      1.96      10.7
551 ## 9     9 Isabela      5.08      1.94      15.9
552 ## 10    10 Isabela      4.80      2.02      6.28
553 ## # ... with 90 more rows

```

554 The order of the columns is not exactly as it was, but this should not matter in a data
 555 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     1 Isabela beak_length  5.18
## 2     1     1 Isabela beak_width  2.02
## 3     1     1 Isabela body_weight 10.3
## 4     2     2 Isabela beak_length  4.97
## 5     2     2 Isabela beak_width  NA
## 6     2     2 Isabela body_weight 17.5
## 7     3     3 Isabela beak_length  4.86
## 8     3     3 Isabela beak_width  2.09
## 9     3     3 Isabela body_weight  6.34
## 10    4     4 Isabela beak_length  5.05
## # ... with 290 more rows
```

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     1 Isabela beak_length  5.18
## 2     1     1 Isabela beak_width  2.02
## 3     1     1 Isabela body_weight 10.3
## 4     2     2 Isabela beak_length  4.97
## 5     2     2 Isabela body_weight 17.5
## 6     3     3 Isabela beak_length  4.86
## 7     3     3 Isabela beak_width  2.09
## 8     3     3 Isabela body_weight  6.34
## 9     4     4 Isabela beak_length  5.05
## 10    5     5 Isabela beak_length  4.95
## # ... with 190 more rows
```

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

```
replace_na(data, replace = list(value = -999))
```

```
592 ## # A tibble: 300 x 4
593 ##       id island variable      value
594 ##   <int> <chr>   <chr>      <dbl>
595 ## 1     1   1 Isabela beak_length  5.18
596 ## 2     2   1 Isabela beak_width  2.02
597 ## 3     3   1 Isabela body_weight 10.3
598 ## 4     4   2 Isabela beak_length  4.97
599 ## 5     5   2 Isabela beak_width -999
600 ## 6     6   2 Isabela body_weight 17.5
601 ## 7     7   3 Isabela beak_length  4.86
602 ## 8     8   3 Isabela beak_width  2.09
603 ## 9     9   3 Isabela body_weight  6.34
604 ## 10    4 Isabela beak_length  5.05
605 ## # ... with 290 more rows
```

606 where the replace argument takes a named list, and the names should refer to the
607 columns to apply the replacement to.

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

```
fill(data, value)
```

```
609 ## # A tibble: 300 x 4
610 ##       id island variable      value
611 ##   <int> <chr>   <chr>      <dbl>
612 ## 1     1   1 Isabela beak_length  5.18
613 ## 2     2   1 Isabela beak_width  2.02
614 ## 3     3   1 Isabela body_weight 10.3
615 ## 4     4   2 Isabela beak_length  4.97
616 ## 5     5   2 Isabela beak_width  4.97
617 ## 6     6   2 Isabela body_weight 17.5
618 ## 7     7   3 Isabela beak_length  4.86
619 ## 8     8   3 Isabela beak_width  2.09
620 ## 9     9   3 Isabela body_weight  6.34
621 ## 10    4 Isabela beak_length  5.05
622 ## # ... with 290 more rows
```

623 Note that most functions in the tidyverse take a tibble as their first argument, and columns
624 to which to apply the functions are usually passed as “objects” rather than character
625 strings. In the above example, we passed the value column as value, not “value”. These
626 column-objects are called by the tidyverse functions *in the context* of the data (the tibble)
627 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     1 Isabela beak_length  5.18     7     1  2019
## 2     2     1 Isabela beak_width   2.02    18    12  2020
## 3     3     1 Isabela body_weight  10.3     4     8  2019
## 4     4     2 Isabela beak_length  4.97     5     8  2020
## 5     5     2 Isabela beak_width    NA    27    12  2020
## 6     6     2 Isabela body_weight  17.5    27     6  2019
## 7     7     3 Isabela beak_length  4.86    29     4  2020
## 8     8     3 Isabela beak_width   2.09    18    12  2019
## 9     9     3 Isabela body_weight   6.34    21     2  2019
## 10    4     4 Isabela beak_length  5.05     9     2  2019
## # ... with 290 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     1 Isabela beak_length  5.18 7-1-2019
## 2     2     1 Isabela beak_width   2.02 18-12-2020
## 3     3     1 Isabela body_weight  10.3 4-8-2019
## 4     4     2 Isabela beak_length  4.97 5-8-2020
## 5     5     2 Isabela beak_width    NA 27-12-2020
## 6     6     2 Isabela body_weight  17.5 27-6-2019
## 7     7     3 Isabela beak_length  4.86 29-4-2020
## 8     8     3 Isabela beak_width   2.09 18-12-2019
## 9     9     3 Isabela body_weight   6.34 21-2-2019
## 10    4     4 Isabela beak_length  5.05 9-2-2019
## # ... with 290 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     1 Isabela beak_length  5.18 7      1    2019
## 2     2     1 Isabela beak_width   2.02 18     12    2020
## 3     3     1 Isabela body_weight  10.3 4       8    2019
## 4     4     2 Isabela beak_length   4.97 5       8    2020
## 5     5     2 Isabela beak_width    NA  27     12    2020
## 6     6     2 Isabela body_weight  17.5 27      6    2019
## 7     7     3 Isabela beak_length   4.86 29      4    2020
## 8     8     3 Isabela beak_width   2.09 18     12    2019
## 9     9     3 Isabela body_weight   6.34 21      2    2019
## 10    10     4 Isabela beak_length   5.05 9       2    2019
## # ... with 290 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     1 Isabela beak_length  5.18 7
## 2     1     1 Isabela beak_length  5.18 1
## 3     1     1 Isabela beak_length  5.18 2019
## 4     1     1 Isabela beak_width   2.02 18
## 5     1     1 Isabela beak_width   2.02 12
## 6     1     1 Isabela beak_width   2.02 2020
## 7     1     1 Isabela body_weight  10.3 4
## 8     1     1 Isabela body_weight  10.3 8
## 9     1     1 Isabela body_weight  10.3 2019
## 10    2     2 Isabela beak_length   4.97 5
## # ... with 890 more rows
```

2.3.4 Expanding tables using combinations

Sometimes one may need to quickly create a table with all combinations of a set of variables. We could generate a tibble with all combinations of island-by-morphometric using `expand_grid`:

```

expand_grid(
  island = c("Isabela", "Santa Cruz"),
  variable = c("beak_length", "beak_width", "body_weight")
)

```

```

701 ## # A tibble: 6 x 2
702 ##   island      variable
703 ##   <chr>      <chr>
704 ## 1 Isabela  beak_length
705 ## 2 Isabela  beak_width
706 ## 3 Isabela  body_weight
707 ## 4 Santa Cruz beak_length
708 ## 5 Santa Cruz beak_width
709 ## 6 Santa Cruz body_weight

```

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

```

expand(data, island, variable)

```

```

712 ## # A tibble: 6 x 2
713 ##   island      variable
714 ##   <chr>      <chr>
715 ## 1 Isabela  beak_length
716 ## 2 Isabela  beak_width
717 ## 3 Isabela  body_weight
718 ## 4 Santa Cruz beak_length
719 ## 5 Santa Cruz beak_width
720 ## 6 Santa Cruz body_weight

```

721 As an extension of this, the function `complete` can come particularly handy if we need to
 722 add missing combinations to our tibble:

```

complete(data, island, variable)

```

```

723 ## # A tibble: 300 x 5
724 ##   island variable      id value date
725 ##   <chr>  <chr>      <int> <dbl> <chr>
726 ## 1 Isabela beak_length      1  5.18 7-1-2019
727 ## 2 Isabela beak_length      2  4.97 5-8-2020
728 ## 3 Isabela beak_length      3  4.86 29-4-2020
729 ## 4 Isabela beak_length      4  5.05 9-2-2019
730 ## 5 Isabela beak_length      5  4.95 14-9-2019
731 ## 6 Isabela beak_length      6  4.97 30-4-2019
732 ## 7 Isabela beak_length      7  5.13 8-8-2020
733 ## 8 Isabela beak_length      8  5.03 9-6-2019
734 ## 9 Isabela beak_length      9  5.08 9-5-2019
735 ## 10 Isabela beak_length     10  4.80 22-5-2019
736 ## # ... with 290 more rows

```

737 which does nothing here because we already have all combinations of island and vari-
738 able.

739 2.3.5 Nesting

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

743 2.4 Extra: factors and the forcats package

```
library(forcats)
```

744 Categorical variables can be stored in R as character strings in character or factor ob-
745 jects. A factor looks like a character, but it actually is an integer vector, where each
746 integer is mapped to a character label. With this respect it is sort of an enhanced ver-
747 sion of character. For example,

```
my_char_vec <- c("Pratik", "Theo", "Raph")
my_char_vec
```

```
## [1] "Pratik" "Theo"   "Raph"
```

749 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
```

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

```
factor(c(my_char_vec, "Raph"))
```

```
## [1] Pratik Theo   Raph   Raph
## Levels: Pratik Raph Theo
```

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

```
levels(my_fact_vec)
```

```
## [1] "Pratik" "Raph"   "Theo"
```

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

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

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

774 2.4.1 Reordering a factor

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

```
fct_relevel(my_fact_vec, c("Pratik", "Theo", "Raph"))
```

```
776 ## [1] Pratik Theo   Raph
777 ## Levels: Pratik Theo Raph
```

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

```
fct_inorder(my_fact_vec)
```

```
780 ## [1] Pratik Theo   Raph
781 ## Levels: Pratik Theo Raph
```

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

```
fct_rev(my_fact_vec)
```

```
783 ## [1] Pratik Theo   Raph
784 ## Levels: Theo Raph Pratik
```

785 Factor reordering may come useful when plotting categorical variables, for example. Say
 786 we want to plot `beak_length` against `island` in our finch dataset:

```
library(ggplot2)
ggplot(data[data$variable == "beak_length",], aes(x = island, y = value)) +
  geom_violin()
```

```
787 ## Warning: Removed 31 rows containing non-finite values (stat_ydensity).
```



788

789 We could use factor reordering to change the order of the violins:

```
data$island <- fct_relevel(data$island, c("Santa Cruz", "Isabela"))
ggplot(data[data$variable == "beak_length",], aes(x = island, y = value)) +
  geom_violin()
```

790 ## Warning: Removed 31 rows containing non-finite values (stat_ydensity).



791

792 Lots of other variants exist for reordering (e.g. reordering by association with a variable),
 793 which we do not cover here. Please refer to the cheatsheet or the online documentation
 794 for more examples.

795 2.4.2 Factor levels

796 One can change the levels of a factor using `fct_recode`:

```
fct_recode(
  my_fact_vec,
  "Pratik Gupte" = "Pratik",
  "Theo Pannetier" = "Theo",
  "Raphael Scherrer" = "Raph"
)
```

```
797 ## [1] Pratik Gupte    Theo Pannetier   Raphael Scherrer
```

```
798 ## Levels: Pratik Gupte Raphael Scherrer Theo Pannetier
```

799 or collapse factor levels together using `fct_collapse`:

```
fct_collapse(my_fact_vec, EU = c("Theo", "Raph"), NonEU = "Pratik")
```

```
800 ## [1] NonEU EU      EU
```

```
801 ## Levels: NonEU EU
```

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

2.4.3 Bonus: dropping levels

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

```
data <- data[data$island == "Santa Cruz",]  
unique(data$island) # Isabela is gone from the labels
```

```
## [1] Santa Cruz
```

```
## Levels: Santa Cruz Isabela
```

```
levels(data$island) # but not from the levels
```

```
## [1] "Santa Cruz" "Isabela"
```

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

```
data <- droplevels(data)  
levels(data$island)
```

```
## [1] "Santa Cruz"
```

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

2.5 External resources

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

- The `readr/tibble/tidyr` and `forcats` cheatsheets.
- This link on the concept of tidy data
- The `tibble`, `tidyr` and `forcats` websites

Chapter 3

Data manipulation with dplyr

```
# load the tidyverse
library(tidyverse)

## -- Attaching packages -----
tidyverse 1.3.0 --

## v purrr 0.3.4      v dplyr 0.8.5

## -- Conflicts ----- tidyverse_conflicts() -
-
## x dplyr::collapse() masks glue::collapse()
## x dplyr::filter()   masks stats::filter()
## x dplyr::lag()      masks stats::lag()
```

3.1 Introduction

Reminders from last weeks: pipe operator, tidy tables, ggplot

Why dplyr ? dplyr vs base R

3.2 Example data of the day

Through this tutorial, we will be using mammal trait data from the Phylacine database.
The dataset contains information on mass, diet, life habit, etc, for more than all living
species of mammals. Let's have a look.

```
phylacine <- readr::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
```

```

842 ##   <chr>      <chr>      <chr>      <chr>      <chr>      <dbl> <dbl>
843 ## 1 Abditomys_l~ Rodentia Muridae  Abditomys latidens      1      0
844 ## 2 Abeomelomys~ Rodentia Muridae  Abeomelo~ sevia      1      0
845 ## 3 Abrawayaomy~ Rodentia Cricetidae Abrawaya~ ruschii      1      0
846 ## 4 Abrocoma_be~ Rodentia Abrocomid~ Abrocoma bennettii      1      0
847 ## 5 Abrocoma_bo~ Rodentia Abrocomid~ Abrocoma boliviensis      1      0
848 ## 6 Abrocoma_bu~ Rodentia Abrocomid~ Abrocoma budini      1      0
849 ## 7 Abrocoma_ci~ Rodentia Abrocomid~ Abrocoma cinerea      1      0
850 ## 8 Abrocoma_fa~ Rodentia Abrocomid~ Abrocoma famatina      1      0
851 ## 9 Abrocoma_sh~ Rodentia Abrocomid~ Abrocoma shistacea      1      0
852 ## 10 Abrocoma_us~ Rodentia Abrocomid~ Abrocoma uspollata      1      0
853 ## # ... with 5,821 more rows, and 17 more variables: Freshwater <dbl>,
854 ## #   Aerial <dbl>, Life.Habit.Method <chr>, Life.Habit.Source <chr>,
855 ## #   Mass.g <dbl>, Mass.Method <chr>, Mass.Source <chr>, Mass.Comparison <chr>,
856 ## #   Mass.Comparison.Source <chr>, Island.Endemicity <chr>,
857 ## #   IUCN.Status.1.2 <chr>, Added.IUCN.Status.1.2 <chr>, Diet.Plant <dbl>,
858 ## #   Diet.Vertebrate <dbl>, Diet.Invertebrate <dbl>, Diet.Method <chr>,
859 ## #   Diet.Source <chr>

```

Note the friendly output given by the tibble (as opposed to a `data.frame`). `readr` automatically stores the content it reads in a tibble, tidyverse oblige. You should know however that `dplyr` doesn't require your data to be in a tibble, a regular `data.frame` will work just as fine.

Most of the `dplyr` verbs covered in the next sections assume your data is *tidy*: wide format, variables as column, 1 observation per row. Not that they won't work if your data isn't tidy, but the results could be very different from what I'm going to show here. Fortunately, the phylacine trait dataset appears to be tidy: there is one unique entry for each species.

The first operation I'm going to run on this table is changing the names with `rename()`. Some people prefer their tea without sugar, and I prefer my variable names without uppercase characters, dots or (if possible) numbers. This will give me the opportunity to introduce the trivial syntax of `dplyr` verbs.

```

phylacine <- phylacine %>%
  dplyr::rename(
    "binomial" = Binomial.1.2,
    "order" = Order.1.2,
    "family" = Family.1.2,
    "genus" = Genus.1.2,
    "species" = Species.1.2,
    "terrestrial" = Terrestrial,
    "marine" = Marine,
    "freshwater" = Freshwater,
    "aerial" = Aerial,
    "life_habit_method" = Life.Habit.Method,
    "life_habit_source" = Life.Habit.Source,
    "mass_g" = Mass.g,

```

```

    "mass_method" = Mass.Method,
    "mass_source" = Mass.Source,
    "mass_comparison" = Mass.Comparison,
    "mass_comparison_source" = Mass.Comparison.Source,
    "island_endemicity" = Island.Endemicity,
    "iucn_status" = IUCN.Status.1.2, # not even for acronyms
    "added_iucn_status" = Added.IUCN.Status.1.2,
    "diet_plant" = Diet.Plant,
    "diet_vertibrate" = Diet.Vertibrate,
    "diet_invertebrate" = Diet.Invertebrate,
    "diet_method" = Diet.Method,
    "diet_source" = Diet.Source
  )

```

872 For convenience, I'm going to use the pipe operator (`%>%`) that we've seen before, through
 873 this chapter. All `dplyr` functions are built to work with the pipe (i.e, their first argument is
 874 always data), but again, this is not compulsory. I could do

```

phylacine <- dplyr::rename(
  data = phylacine,
  "binomial" = Binomial.1.2,
  # ...
)

```

875 Note how columns are referred to. Once the data has been passed as an argument, no need
 876 to refer to it anymore, `dplyr` understands that you're dealing with variables inside that
 877 data frame. So drop that `data$var`, `data[, "var"]`, and, if you've read *The R book*, forget
 878 the very existence of `attach()`.

879 Finally, I should mention that you can refer to variables names either with strings or di-
 880 rectly as objects, whether you're reading or creating them:

```

phylacine2 <- readr::read_csv("data/phylacine_traits.csv")

phylacine2 %>%
  dplyr::rename(
    # this works
    binomial = Binomial.1.2
  )
phylacine2 %>%
  dplyr::rename(
    # this works too!
    binomial = "Binomial.1.2"
  )
phylacine2 %>%
  dplyr::rename(
    # guess what
    "binomial" = "Binomial.1.2"
  )

```

)

881 **3.3 Select variables with `select()`**

882 **3.4 Select observations with `filter()`**

883 **3.5 Create new variables with `mutate()`**

884 can also edit existing ones

885 drop existing variables with `transmute()`

886 **3.6 Grouped results with `group_by()` and `summarise()`**

887 **3.7 Scoped variables**

```
data(mtcars)
mtcars %>% select_all(toupper)

is_whole <- function(x) all(floor(x) == x)
mtcars %>% select_if() # select integers only

mtcars %>% select_at(vars(-contains("ar")))
mtcars %>% select_at(vars(-contains("ar"), starts_with("c")))
```

888 **3.8 More !**

889 dolla sign x point operator variables values -> `dplyr::distinct()` eq. to `base::unique()` sam-
890 `ple()` `slice()`

891 Chapter 4

892 Working with lists and iteration



Every use case is ridiculous
until it happens to you.

893

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

894 4.1 Iteration with map

895 Iteration in base R is commonly done with `for` and `while` loops. There is no readymade al-
896 ternative to `while` loops in the tidyverse. However, the functionality of `for` loops is spread
897 over the `map` family of functions from `purrr`.

898 `purrr` functions are *functionals*, i.e., functions that take another function as an argument.
899 The closest equivalent in R is the `*apply` family of functions: `apply`, `lapply`, `vapply` and
900 so on.

901 A good reason to use `purrr` functions instead of base R functions is their consistent and

902 clear naming, which always indicates how they should be used. This is explained in the
903 examples below.

904 These reasons, as well as how `map` is different from `for` and `lapply` are best explained in
905 the **Advanced R Book**.

906 4.1.1 Basic use of `map`

907 `map` works on any list-like object, which includes vectors, and always returns a list. `map`
908 takes two arguments, the object on which to operate, and the function to apply to each
909 element.

```
# get the square root of each integer 1 - 10
some_numbers = 1:10
map(some_numbers, sqrt)

910 ## [[1]]
911 ## [1] 1
912 ##
913 ## [[2]]
914 ## [1] 1.414214
915 ##
916 ## [[3]]
917 ## [1] 1.732051
918 ##
919 ## [[4]]
920 ## [1] 2
921 ##
922 ## [[5]]
923 ## [1] 2.236068
924 ##
925 ## [[6]]
926 ## [1] 2.44949
927 ##
928 ## [[7]]
929 ## [1] 2.645751
930 ##
931 ## [[8]]
932 ## [1] 2.828427
933 ##
934 ## [[9]]
935 ## [1] 3
936 ##
937 ## [[10]]
938 ## [1] 3.162278
```



```

954 ##      <dbl> <list>                <int>      <dbl> <chr>
955 ## 1      6 <tibble [7 x 11]>          7      19.7 Mazda RX4
956 ## 2      4 <tibble [11 x 11]>         11      26.7 Datsun 710
957 ## 3      8 <tibble [14 x 11]>         14      15.1 Hornet Sportabout

```

958 `map` accepts multiple functions that are applied in sequence to the input list-like object,
 959 but this is confusing to the reader and ill advised.

960 4.1.4 `map` variants returning dataframes

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

```

# split mtcars into 3 dataframes, one per cylinder number
some_list = split(mtcars, mtcars$cyl)

```

```

# get the first two rows of each dataframe
map_df(some_list, head, n = 2)

```

```

963 ##      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
964 ## 1  22.8   4 108.0  93 3.85 2.320 18.61  1  1   4    1
965 ## 2  24.4   4 146.7  62 3.69 3.190 20.00  1  0   4    2
966 ## 3  21.0   6 160.0 110 3.90 2.620 16.46  0  1   4    4
967 ## 4  21.0   6 160.0 110 3.90 2.875 17.02  0  1   4    4
968 ## 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0   3    2
969 ## 6  14.3   8 360.0 245 3.21 3.570 15.84  0  0   3    4

```

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

972 `map_dfr` behaves the same as `map_df`.

```

# the same as above but with a pipe
some_list %>%
  map_dfr(head, n = 2)

```

```

973 ##      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
974 ## 1  22.8   4 108.0  93 3.85 2.320 18.61  1  1   4    1
975 ## 2  24.4   4 146.7  62 3.69 3.190 20.00  1  0   4    2
976 ## 3  21.0   6 160.0 110 3.90 2.620 16.46  0  1   4    4
977 ## 4  21.0   6 160.0 110 3.90 2.875 17.02  0  1   4    4
978 ## 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0   3    2
979 ## 6  14.3   8 360.0 245 3.21 3.570 15.84  0  0   3    4

```

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

```

some_list %>%
  map_dfc(head, n = 2)

```



```

982 ##   mpg cyl  disp hp drat   wt  qsec vs am gear carb mpg1 cyl1 disp1 hp1 drat1
983 ## 1 22.8   4 108.0 93 3.85 2.32 18.61 1 1   4    1  21    6  160 110   3.9
984 ## 2 24.4   4 146.7 62 3.69 3.19 20.00 1 0   4    2  21    6  160 110   3.9
985 ##   wt1 qsec1 vs1 am1 gear1 carb1 mpg2 cyl2 disp2 hp2 drat2 wt2 qsec2 vs2 am2
986 ## 1 2.620 16.46  0   1     4     4 18.7   8  360 175  3.15 3.44 17.02  0   0
987 ## 2 2.875 17.02  0   1     4     4 14.3   8  360 245  3.21 3.57 15.84  0   0
988 ##   gear2 carb2
989 ## 1       3     2
990 ## 2       3     4

```

4.1.5 Selective mapping

map_at and map_if work like other *_at and *_if functions.

Here, map_if is used to run a linear model only on those dataframes which have sufficient data. The predicate is specified by .p.

```

# split mtcars by cylinder number and run an lm only if there are more than 10 rows
data <- nest(mtcars, data = -cyl)

data <- mutate(data,
  model = map_if(.x = data,
    .p = function(x){
      nrow(x) > 10
    },
    .f = function(x){
      lm(mpg ~ wt, data = x)
    })

# check the data structure
data

995 ## # A tibble: 3 x 3
996 ##   cyl data          model
997 ##   <dbl> <list>         <list>
998 ## 1     6 <tibble [7 x 10]> <tibble [7 x 10]>
999 ## 2     4 <tibble [11 x 10]> <lm>
1000 ## 3     8 <tibble [14 x 10]> <lm>

```

map_at works on specific elements of a list or vector. Come back to this, it's not particularly useful.

4.2 More map variants

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

```

# consider 2 vectors and replicate the simple vector addition using map2
map2(.x = 1:5,
     .y = 6:10,
     .f = sum)

1007 ## [[1]]
1008 ## [1] 7
1009 ##
1010 ## [[2]]
1011 ## [1] 9
1012 ##
1013 ## [[3]]
1014 ## [1] 11
1015 ##
1016 ## [[4]]
1017 ## [1] 13
1018 ##
1019 ## [[5]]
1020 ## [1] 15

```

1021 4.2.1 Mapping over two inputs with map2

1022 map2 has the same variants as map, allowing for different return types. Here map2_int
 1023 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)

1024 ## [1] 7 9 11 13 15

```

1025 map2 doesn't have _at and _if variants.

1026 One use case for map2 is to deal with both a list element and its index, as shown in the
 1027 example. This may be necessary when the list index is removed in a split or nest. This
 1028 can also be done with imap, where the index is referred to as .y.

```

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

# a not particularly useful example
map2(this_list, names(this_list),
     function(x, y) {
       glue::glue('{x} : {y}')
     })

1029 ## $a

```

```

1030 ## first letter : a
1031 ##
1032 ## $b
1033 ## second letter : b

    # imap can also do this
    imap(this_list,
          function(x, .y){
            glue::glue('{x} : {.y}')
          })

1034 ## $a
1035 ## first letter : a
1036 ##
1037 ## $b
1038 ## second letter : b

```

1039 4.2.2 Mapping over multiple inputs with pmap

1040 pmap instead operates on a list of multiple list-like objects, and also comes with the same
 1041 return type variants as map. The example shows both aspects of pmap using pmap_chr.

```

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

    # print a few statements
    pmap_chr(list(list_01, list_02, list_03),
              function(l1, l2, l3){
                glue::glue('number {l1}, letter {l2}, colour {l3}')
              })

1042 ## [1] "number 1, letter a, colour #FF0000FF"
1043 ## [2] "number 2, letter b, colour #00FF00FF"
1044 ## [3] "number 3, letter c, colour #0000FFFF"

```

1045 4.2.3 Mapping at depth

1046 Lists are often nested, that is, a list element may itself be a list. It is possible to map a
 1047 function over elements as a specific depth.

1048 In the example, mtcars is split by cylinders, and then by gears, creating a two-level list,
 1049 with the second layer operated on.

```

    # use map to make a 2 level list
    this_list = split(mtcars, mtcars$cyl) %>%
      map(function(df){ split(df, df$gear) })

```

```

# map over the second level to count the number of
# cars with N gears in the set of cars with M cylinders
# display only for cyl = 4
map_depth(this_list[1], 2, nrow)
1050 ## $`4`
1051 ## $`4`$`3`
1052 ## [1] 1
1053 ##
1054 ## $`4`$`4`
1055 ## [1] 8
1056 ##
1057 ## $`4`$`5`
1058 ## [1] 2

```

1059 4.2.4 Iteration without a return

1060 `map` and its variants have a return type, which is either a list or a vector. However, it is
 1061 often necessary to iterate a function over a list-like object for that function's side effects,
 1062 such as printing a message to screen, plotting a series of figures, or saving to file.

1063 `walk` is the function for this task. It has only the variants `walk2`, `iwalk`, and `pwalk`, whose
 1064 logic is similar to `map2`, `imap`, and `pmap`. In the example, the function applied to each list
 1065 element is intended to print a message.

```

this_list = split(mtcars, mtcars$cyl)

iwalk(this_list,
      function(df, .y){
        message(glue::glue('{nrow(df)} cars with {.y} cylinders'))
      })
1066 ## 11 cars with 4 cylinders
1067 ## 7 cars with 6 cylinders
1068 ## 14 cars with 8 cylinders

```

1069 4.2.5 Modify rather than map

1070 When the return type is expected to be the same as the input type, that is, a list returning
 1071 a list, or a character vector returning the same, `modify` can help with keeping strictly to
 1072 those expectations.

1073 In the example, simply adding 2 to each vector element produces an error, because the
 1074 output is a numeric, or double. `modify` helps ensure some type safety in this way.

```

vec = as.integer(1:10)

tryCatch(

```

```

expr = {
  # this is what we want you to look at

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

  },

  # do not pay attention to this
  error = function(e){
    print(toString(e))
  }
)
1075 ## [1] "Error: Can't coerce element 1 from a double to a integer\n"
1076 Converting the output to an integer, which was the original input type, serves as a solution.
      modify(vec, function(x) { as.integer(x + 2) })
1077 ## [1] 3 4 5 6 7 8 9 10 11 12

1078 A note on invoke
1079 invoke used to be a wrapper around do.call, and can still be found with its family of
1080 functions in purrr. It is however retired in favour of functionality already present in map
1081 and rlang::exec, the latter of which will be covered in another session.

```

1082 4.3 Working with lists

1083 `purrr` has a number of functions to work with lists, especially lists that are not nested
 1084 list-columns in a tibble.

1085 4.3.1 Filtering lists

1086 Lists can be filtered on any predicate using `keep`, while the special case `compact` is applied
 1087 when the empty elements of a list are to be filtered out. `discard` is the opposite of `keep`,
 1088 and keeps only elements not satisfying a condition. Again, the predicate is specified by
 1089 `.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 })

1090 ## $a
1091 ## [1] 1
1092 ##
1093 ## $c
1094 ## [1] 2

1095 head_while is bit of an odd case, which returns all elements of a list-like object in se-
1096 quence until the first one fails to satisfy a predicate, specified by .p.

1:10 %>%
  head_while(.p = function(x) x < 5)

1097 ## [1] 1 2 3 4

```

1098 4.3.2 Summarising lists

1099 The purrr functions every, some, has_element, detect, detect_index, and vec_depth
 1100 help determine whether a list passes a certain logical test or not. These are seldom used
 1101 and are not discussed here.

1102 4.3.3 Reduction and accumulation

1103 reduce helps combine elements along a list using a specific function. Consider the exam-
 1104 ple 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)

1105 ## [1] 1 2 3 3 4 5 6 7 8 9 10

```

1106 The way reduce works is to take the first element, a in the example, and find its intersec-
 1107 tion with b, and to take the result and find its intersection with c.

```

this_list = list(a = 1:3, b = 3:6, c = 3:10)

reduce(this_list, intersect)

1108 ## [1] 3

```

1109 accumulate works very similarly, except it retains the intermediate products. The first
 1110 element is retained as is. accumulate2 and reduce2 work on two lists, following the same
 1111 logic as map2 etc. Both functions can be used in much more complex ways than demon-
 1112 strated 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")

1113 ## $a
1114 ## [1] 1 2 3
1115 ##
1116 ## $b
1117 ## [1] 1 2 3 4 5 6
1118 ##
1119 ## $c
1120 ## [1] 1 2 3 4 5 6 7 8 9 10
1121 ##
1122 ## $d
1123 ## [1] 1 2 3 4 5 6 7 8 9 10 12

```

1124 4.3.4 Miscellaneous operation

1125 purrr offers a few more functions to work with lists (or list like objects). `prepend` works
 1126 very similarly to `append`, except it adds to the head of a list. `splice` adds multiple objects
 1127 together in a list. `splice` will break the existing list structure of input lists.

```

# use prepend to add values to the head of a list
prepend(x = list("a", "b"), values = list("1", "2"))

1128 ## [[1]]
1129 ## [1] "1"
1130 ##
1131 ## [[2]]
1132 ## [1] "2"
1133 ##
1134 ## [[3]]
1135 ## [1] "a"
1136 ##
1137 ## [[4]]
1138 ## [1] "b"

# use splice to add multiple elements together
splice(list("a", "b"), list("1", "2"), "something else")

1139 ## [[1]]
1140 ## [1] "a"
1141 ##
1142 ## [[2]]
1143 ## [1] "b"
1144 ##
1145 ## [[3]]
1146 ## [1] "1"

```

```

1147 ##
1148 ## [[4]]
1149 ## [1] "2"
1150 ##
1151 ## [[5]]
1152 ## [1] "something else"

1153 flatten has a similar behaviour, and converts a list of vectors or list of lists to a single
1154 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

1155 ## $a
1156 ## [1] "a" "a" "a"
1157 ##
1158 ## $b
1159 ## [1] "b" "b" "b" "b"

# use flatten chr to get a character vector
flatten_chr(this_list)

1160 ## [1] "a" "a" "a" "b" "b" "b" "b"

1161 transpose shifts the index order in multi-level lists. This is seen in the example, where
1162 the gear goes from being the index of the second level to the index of the first.

this_list = split(mtcars, mtcars$cyl) %>%
  map(function(df) split(df, df$gear))

# from a list of lists where cars are divided by cylinders and then
# gears, this is now a list of lists where cars are divided by
# gears and then cylinders
transpose(this_list[1])

1163 ## $`3`
1164 ## $`3`$`4`
1165 ##           mpg cyl  disp hp drat   wt  qsec vs am gear carb
1166 ## Toyota Corona 21.5   4 120.1 97   3.7 2.465 20.01  1  0    3    1
1167 ##
1168 ##
1169 ## $`4`
1170 ## $`4`$`4`
1171 ##           mpg cyl  disp hp drat   wt  qsec vs am gear carb
1172 ## Datsun 710    22.8   4 108.0 93  3.85  2.320 18.61  1  1    4    1
1173 ## Merc 240D     24.4   4 146.7 62  3.69  3.190 20.00  1  0    4    2
1174 ## Merc 230      22.8   4 140.8 95  3.92  3.150 22.90  1  0    4    2
1175 ## Fiat 128      32.4   4  78.7 66  4.08  2.200 19.47  1  1    4    1

```



```

1176 ## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1   4   2
1177 ## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1   4   1
1178 ## Fiat X1-9       27.3   4  79.0  66 4.08 1.935 18.90  1  1   4   1
1179 ## Volvo 142E      21.4   4 121.0 109 4.11 2.780 18.60  1  1   4   2
1180 ##
1181 ##
1182 ## $`5`
1183 ## $`5`$`4`
1184 ##           mpg cyl  disp  hp drat   wt  qsec vs am gear carb
1185 ## Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.7  0  1   5   2
1186 ## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1   5   2

```