

TRES Tidyverse Tutorial

Raphael, Pratik and Theo

2020-05-30

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 1. The new data frame: <code>tibble</code>	24
14	2.2 2. The concept of tidy data	26
15	2.3 3. Reshaping with <code>tidyr</code>	29
16	2.4 4. Extra: factors and the <code>forcats</code> package	35
17	2.5 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 Basic iteration with <code>map</code>	45
29	4.2 More <code>map</code> variants	49
30	4.3 Modification in place	49
31	4.4 Working with lists	49

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

48 **Join**

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

50 *Tentative dates.

51 Chapter 1

52 Reading files and string 53 manipulation



Every use case is ridiculous
until it happens to you.

54
55 Load the packages for the day.

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

56 1.1 Data import and export with readr

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

61 1.1.1 Reading data

62 Reading in a csv file with `readr` is done with the `read_csv` function, a faster alternative to
 63 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

```

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

89 Other variants include:

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

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

95 skipped before reading data.

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

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

100 1.1.2 Writing data

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

105 These functions are not covered here.

106 1.1.3 Reading and writing lines

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

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

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

113 The model summary can be written to file. When writing lines to file, BE AWARE OF THE
114 DIFFERENCES BETWEEN UNIX AND WINDOWS line separators. Usually, this causes no
115 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")
```

116 This model output can be read back in for display, and each line of the model output is an
117 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")
```

```

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

```

```

153 ## [1] "this string_this other string"
    # str_dup works like rep
    str_dup("this string", times = 3)
154 ## [1] "this stringthis stringthis string"
    # str_flatten works on lists and vectors
    str_flatten(string = as.list(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"
    str_flatten(string = letters, collapse = "-")
156 ## [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"
157 str_flatten is especially useful when displaying the type of an object that returns a list
158 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 = ", ")
159 ## [1] "tbl_df, tbl, data.frame"

```

1.2.2 Detecting strings

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

164 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")
165 ## [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")
166 ## [1] 5 3
    # vectorise over the pattern to count both a-s and b-s
    str_count(string = "ababababa", pattern = c("a", "b"))
167 ## [1] 5 4
168 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"))

```

```

169 ## [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"))

170 ## [1] 5 1 4 3

171 str_locate locates the search pattern in a string, and returns the start and end as a two
172 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")

173 ##      start end
174 ## [1,]      2  4

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

175 ## [1] TRUE

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

176 ## [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")))

177 ## [1] TRUE

178 Detect whether a string starts or ends with a pattern. Also vectorised. Both have a negate
179 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")

180 ## [1] FALSE FALSE  TRUE  TRUE

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

181 ## [1]  TRUE FALSE FALSE  TRUE

```

```

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

182 ## [1] FALSE TRUE TRUE FALSE

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

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

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

188 ## [1] "banana" "bananageddon is coming"

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

189 ## [1] 1 2

```

1.2.3 Matching strings

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

193 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")

194 ##      [,1]
195 ## [1,] "banana"
196 ## [2,] "banana"
197 ## [3,] "banana"

```

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

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

203 The return type is a list, each element is a character matrix where the first column is
 204 the string subset matching the full search pattern, and then as many columns as there
 205 are parts to the search pattern. The parts of interest in the search pattern are indicated
 206 by wrapping them in parentheses. For example, in the case below, wrapping `[-.]` in
 207 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"
```

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

219 The return type is a list, with one element per input string. Each element is a character
 220 matrix, where each row is one possible match, and each column after the first (the full
 221 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]+)")

222 ## [[1]]
223 ##           [,1]           [,2]  [,3]
224 ## [1,] "1970-somemonth"    "1970" "somemonth"
225 ## [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]+)")

226 ## [[1]]
```

```

227 ##      [,1]      [,2] [,3]
228 ## [1,] "1970-somemonth" "1970" "somemonth"
229 ## [2,] "1990/anothermonth" "1990" "anothermonth"
230 ##
231 ## [[2]]
232 ##      [,1]      [,2] [,3]
233 ## [1,] "1990-somemonth" "1990" "somemonth"
234 ## [2,] "2001/anothermonth" "2001" "anothermonth"

```

235 1.2.4 Simpler pattern extraction

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

239 `str_extract` returns a character vector with the same length as the input string vector,
 240 while `str_extract_all` returns a list, with a character vector whose elements are the
 241 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]+)")

242 ## [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]+)")

243 ## [[1]]
244 ## [1] "1970-somemonth" "1990/anothermonth"
245 ##
246 ## [[2]]
247 ## [1] "1990-somemonth" "2001/anothermonth"

```

248 1.2.5 Breaking strings apart

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

252 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 = "[\\-|\\/]")

```

```

253 ## [[1]]
254 ## [1] "1970"      "somemonth" "01"
255 ##
256 ## [[2]]
257 ## [1] "1990"      "anothermonth" "01"

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

260 ## [[1]]
261 ## [1] "sim"      "param1"  "0.01"    "param2"  "0.05"    "param3"  "0.01.ext"

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

262 ## [[1]]
263 ## [1] ""
264 ## [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)")

265 ## [[1]]
266 ## [1] ""      ""      "0.01"  "0.05"  "0.01"  ""

267 str_split_fixed split the string into as many pieces as specified, and can be especially
268 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)

269 ##      [,1]      [,2]
270 ## [1,] "dir_level_1" "dir_level_2/file.ext"

```

271 1.2.6 Replacing string elements

```

272 str_replace is intended to replace the search pattern, and can be co-opted into the
273 task of recovering simulation parameters or other data from regularly named files.
274 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 = " ")

275 ## [1] " 0.01 0.05 0.01 "

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

278 Having replaced unwanted characters in the filename with spaces, str_trim offers a way
279 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

280 ## [1] "0.01 0.05 0.01"

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

281 ## [[1]]
282 ## [1] "0.01" "0.05" "0.01"

```

283 1.2.7 Subsetting within strings

284 When strings are highly regular, useful data can be extracted from a string using `str_sub`.
 285 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)

286 ## [1] "1970" "1990" "2010"

287 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)

288 ## [1] "01" "21" "31"

```

289 Finally, it's also possible to replace characters within a string based on the position. This
 290 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
291 ## [1] "1970-somemonth-01" "1990-anothermonth-01" "2010-thismonth-01"
```

292 1.2.8 Padding and truncating strings

293 Strings included in filenames or plots are often of unequal lengths, especially when they
 294 represent numbers. `str_pad` can pad strings with suitable characters to maintain equal
 295 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")
296 ## [1] "001" "010" "100"
```

297 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.")
298 ## [1] "bananas are great etc. etc."
```

299 1.2.9 Stringr aspects not covered here

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

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

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

308 1.3 String interpolation with glue

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

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

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

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

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

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

```

329 ## sim_data_param1_d_param2_4.ext
330 ## sim_data_param1_e_param2_5.ext

```

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

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

336 1.4 Strings in ggplot

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

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

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

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

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

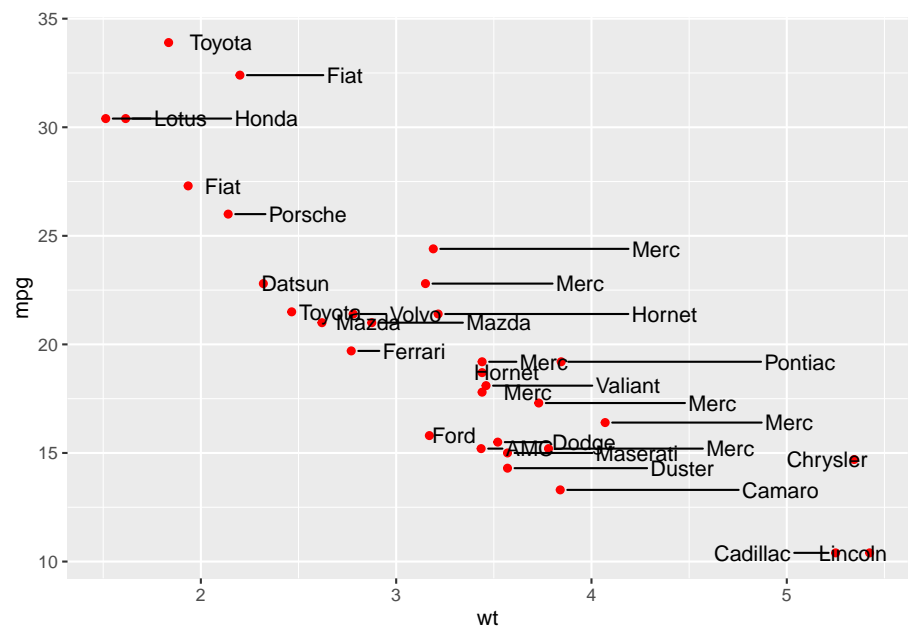
```

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)

```



349

350 This is not a good looking plot, because it breaks other rules of plot design, such as
351 whether this sort of plot should be made at all. Labels and text need to be applied
352 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 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]

```

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

```

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

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

449 2.2 2. The concept of tidy data

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

455 Say we have a dataset of several morphometrics measured on Darwin’s finches in the Gala-
 456 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
```

```

457 ## # A tibble: 100 x 5
458 ##       id beak_length beak_width body_weight island
459 ##   <int>      <dbl>      <dbl>      <dbl> <chr>
460 ## 1     1         4.98         1.99        11.0 Isabela
461 ## 2     2         4.93         1.83         8.51 Isabela
462 ## 3     3         5.19         1.79         9.68 Isabela
463 ## 4     4         4.95         1.96         5.99 Isabela
464 ## 5     5         5.09         2.01         7.38 Isabela
465 ## 6     6         4.87         2.26         9.51 Isabela
466 ## 7     7         4.99         2.13         5.69 Isabela
467 ## 8     8         4.96         1.96        16.1 Isabela
468 ## 9     9         4.96         2.01         7.56 Isabela
469 ## 10    10         4.99         1.97         6.85 Isabela
470 ## # ... with 90 more rows

```

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

476 The tidy equivalent of this dataset would be:

```

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

```

```

477 ## # A tibble: 300 x 4
478 ##       id island variable    value
479 ##   <int> <chr>   <chr>      <dbl>
480 ## 1     1 Isabela beak_length 4.98
481 ## 2     1 Isabela beak_width  1.99
482 ## 3     1 Isabela body_weight 11.0
483 ## 4     2 Isabela beak_length 4.93
484 ## 5     2 Isabela beak_width  1.83
485 ## 6     2 Isabela body_weight 8.51
486 ## 7     3 Isabela beak_length 5.19

```

```

487 ## 8      3 Isabela beak_width  1.79
488 ## 9      3 Isabela body_weight 9.68
489 ## 10     4 Isabela beak_length 4.95
490 ## # ... with 290 more rows

```

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

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

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

505 Another common example of wide-or-long dilemma is when dealing with *contingency ta-*
 506 *bles*. This would be our case, for example, if we asked how many observations we have for
 507 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

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

```

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

```

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

517 ## # A tibble: 6 x 3
518 ##   island    variable     n
519 ##   <chr>    <chr>    <int>
520 ## 1 Isabela beak_length  50
521 ## 2 Santa Cruz beak_length  50
522 ## 3 Isabela  beak_width   50

```

```

523 ## 4 Santa Cruz beak_width      50
524 ## 5 Isabela   body_weight      50
525 ## 6 Santa Cruz body_weight      50

```

526 2.3 3. Reshaping with tidyr

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

536 2.3.1 3.1. Pivoting

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

```

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

```

```

541 ## # A tibble: 100 x 5
542 ##       id island beak_length beak_width body_weight
543 ##   <int> <chr>      <dbl>      <dbl>      <dbl>
544 ## 1     1 Isabela      4.98      1.99      11.0
545 ## 2     2 Isabela      4.93      1.83      8.51
546 ## 3     3 Isabela      5.19      1.79      9.68
547 ## 4     4 Isabela      4.95      1.96      5.99
548 ## 5     5 Isabela      5.09      2.01      7.38
549 ## 6     6 Isabela      4.87      2.26      9.51
550 ## 7     7 Isabela      4.99      2.13      5.69
551 ## 8     8 Isabela      4.96      1.96     16.1
552 ## 9     9 Isabela      4.96      2.01      7.56
553 ## 10    10 Isabela      4.99      1.97      6.85
554 ## # ... with 90 more rows

```

555 The order of the columns is not exactly as it was, but this should not matter in a data
 556 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 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 NA
## 2     1     1 Isabela beak_width  1.99
## 3     1     1 Isabela body_weight NA
## 4     2     2 Isabela beak_length  4.93
## 5     2     2 Isabela beak_width  NA
## 6     2     2 Isabela body_weight NA
## 7     3     3 Isabela beak_length  5.19
## 8     3     3 Isabela beak_width  NA
## 9     3     3 Isabela body_weight  9.68
## 10    4     4 Isabela beak_length  4.95
## # ... 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_width  1.99
## 2     2     2 Isabela beak_length  4.93
## 3     3     3 Isabela beak_length  5.19
## 4     3     3 Isabela body_weight  9.68
## 5     4     4 Isabela beak_length  4.95
## 6     4     4 Isabela body_weight  5.99
## 7     5     5 Isabela beak_length  5.09
## 8     5     5 Isabela beak_width  2.01
## 9     5     5 Isabela body_weight  7.38
## 10    6     6 Isabela beak_width  2.26
## # ... with 190 more rows
```

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

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

```
593 ## # A tibble: 300 x 4
594 ##       id island variable      value
595 ##   <int> <chr>   <chr>      <dbl>
596 ## 1     1   1 Isabela beak_length -999
597 ## 2     2   1 Isabela beak_width  1.99
598 ## 3     3   1 Isabela body_weight -999
599 ## 4     4   2 Isabela beak_length  4.93
600 ## 5     5   2 Isabela beak_width -999
601 ## 6     6   2 Isabela body_weight -999
602 ## 7     7   3 Isabela beak_length  5.19
603 ## 8     8   3 Isabela beak_width -999
604 ## 9     9   3 Isabela body_weight  9.68
605 ## 10    4 Isabela beak_length  4.95
606 ## # ... with 290 more rows
```

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

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

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

```
610 ## # A tibble: 300 x 4
611 ##       id island variable      value
612 ##   <int> <chr>   <chr>      <dbl>
613 ## 1     1   1 Isabela beak_length NA
614 ## 2     2   1 Isabela beak_width  1.99
615 ## 3     3   1 Isabela body_weight  1.99
616 ## 4     4   2 Isabela beak_length  4.93
617 ## 5     5   2 Isabela beak_width  4.93
618 ## 6     6   2 Isabela body_weight  4.93
619 ## 7     7   3 Isabela beak_length  5.19
620 ## 8     8   3 Isabela beak_width  5.19
621 ## 9     9   3 Isabela body_weight  9.68
622 ## 10    4 Isabela beak_length  4.95
623 ## # ... with 290 more rows
```

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

2.3.3 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 NA      10    12  2020
## 2     2     1 Isabela beak_width  1.99     2    10  2019
## 3     3     1 Isabela body_weight NA      15     9  2019
## 4     4     2 Isabela beak_length  4.93    26     6  2019
## 5     5     2 Isabela beak_width  NA      29     9  2020
## 6     6     2 Isabela body_weight  NA      26     7  2020
## 7     7     3 Isabela beak_length  5.19    20     2  2019
## 8     8     3 Isabela beak_width  NA       8    10  2019
## 9     9     3 Isabela body_weight  9.68    15    11  2020
## 10    10     4 Isabela beak_length  4.95    24    10  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 NA  10-12-2020
## 2     2     1 Isabela beak_width  1.99 2-10-2019
## 3     3     1 Isabela body_weight NA  15-9-2019
## 4     4     2 Isabela beak_length  4.93 26-6-2019
## 5     5     2 Isabela beak_width  NA  29-9-2020
## 6     6     2 Isabela body_weight  NA  26-7-2020
## 7     7     3 Isabela beak_length  5.19 20-2-2019
## 8     8     3 Isabela beak_width  NA  8-10-2019
## 9     9     3 Isabela body_weight  9.68 15-11-2020
## 10    10     4 Isabela beak_length  4.95 24-10-2019
## # ... 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 NA     10    12   2020
## 2     2     1 Isabela beak_width  1.99  2     10   2019
## 3     3     1 Isabela body_weight NA     15     9   2019
## 4     4     2 Isabela beak_length  4.93 26     6   2019
## 5     5     2 Isabela beak_width  NA    29     9   2020
## 6     6     2 Isabela body_weight  NA    26     7   2020
## 7     7     3 Isabela beak_length  5.19 20     2   2019
## 8     8     3 Isabela beak_width  NA     8    10   2019
## 9     9     3 Isabela body_weight  9.68 15    11   2020
## 10    10     4 Isabela beak_length  4.95 24    10   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 NA     10
## 2     1     1 Isabela beak_length NA     12
## 3     1     1 Isabela beak_length NA    2020
## 4     1     1 Isabela beak_width  1.99  2
## 5     1     1 Isabela beak_width  1.99 10
## 6     1     1 Isabela beak_width  1.99 2019
## 7     1     1 Isabela body_weight NA     15
## 8     1     1 Isabela body_weight NA     9
## 9     1     1 Isabela body_weight NA    2019
## 10    2     2 Isabela beak_length  4.93 26
## # ... with 890 more rows
```

2.3.4 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")
)

```

```

702 ## # A tibble: 6 x 2
703 ##   island      variable
704 ##   <chr>      <chr>
705 ## 1 Isabela  beak_length
706 ## 2 Isabela  beak_width
707 ## 3 Isabela  body_weight
708 ## 4 Santa Cruz beak_length
709 ## 5 Santa Cruz beak_width
710 ## 6 Santa Cruz body_weight

```

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

```

expand(data, island, variable)

```

```

713 ## # A tibble: 6 x 2
714 ##   island      variable
715 ##   <chr>      <chr>
716 ## 1 Isabela  beak_length
717 ## 2 Isabela  beak_width
718 ## 3 Isabela  body_weight
719 ## 4 Santa Cruz beak_length
720 ## 5 Santa Cruz beak_width
721 ## 6 Santa Cruz body_weight

```

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

```

complete(data, island, variable)

```

```

724 ## # A tibble: 300 x 5
725 ##   island variable      id value date
726 ##   <chr>  <chr>      <int> <dbl> <chr>
727 ## 1 Isabela beak_length      1 NA    10-12-2020
728 ## 2 Isabela beak_length      2  4.93 26-6-2019
729 ## 3 Isabela beak_length      3  5.19 20-2-2019
730 ## 4 Isabela beak_length      4  4.95 24-10-2019
731 ## 5 Isabela beak_length      5  5.09 17-2-2019
732 ## 6 Isabela beak_length      6 NA    12-6-2019
733 ## 7 Isabela beak_length      7  4.99 27-11-2020
734 ## 8 Isabela beak_length      8  4.96 17-3-2020
735 ## 9 Isabela beak_length      9  4.96 24-5-2019
736 ## 10 Isabela beak_length     10  4.99 8-3-2020
737 ## # ... with 290 more rows

```

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

740 2.3.5 3.5. Nesting

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

744 2.4 4. Extra: factors and the forcats package

```
library(forcats)
```

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

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

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

750 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
```

753 is a factor, of which the *labels* are displayed. The *levels* of the factor are the unique values
754 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
```

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

```
levels(my_fact_vec)
```

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

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

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

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

775 2.4.1 4.1. Reordering a factor

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

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

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

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

```
fct_inorder(my_fact_vec)
```

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

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

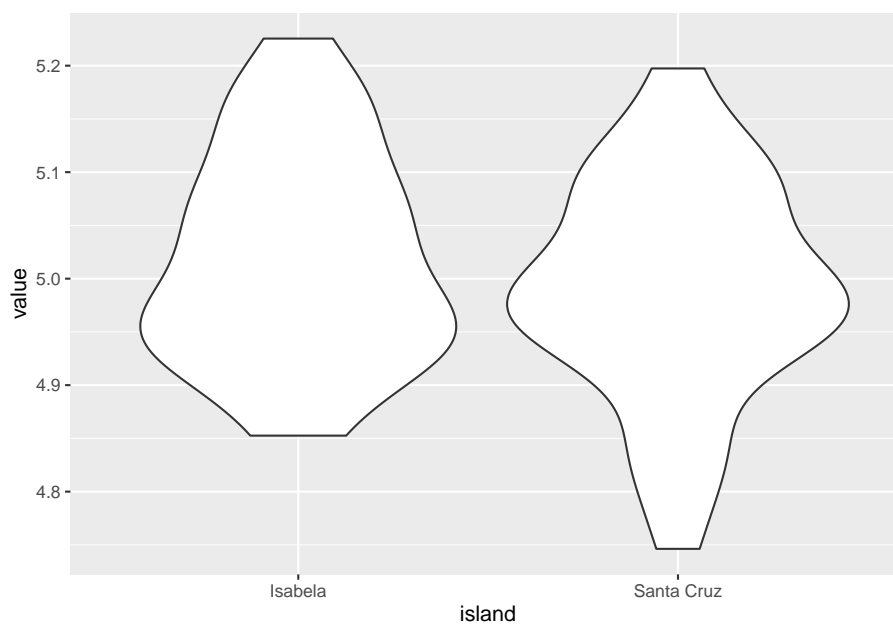
```
fct_rev(my_fact_vec)
```

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

786 Factor reordering may come useful when plotting categorical variables, for example. Say
 787 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()
```

```
788 ## Warning: Removed 29 rows containing non-finite values (stat_ydensity).
```

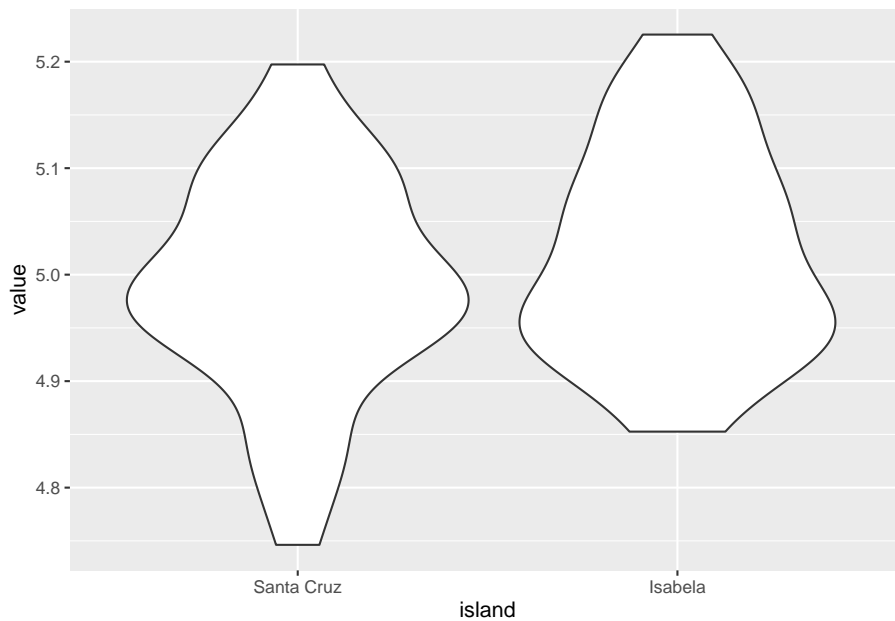


789

790 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()
```

791 ## Warning: Removed 29 rows containing non-finite values (stat_ydensity).



792

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

796 2.4.2 4.2. Factor levels

797 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"
)
```

```
798 ## [1] Pratik Gupte    Theo Pannetier   Raphael Scherrer
```

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

800 or collapse factor levels together using `fct_collapse`:

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

```
801 ## [1] NonEU EU      EU
```

```
802 ## Levels: NonEU EU
```

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

806 **2.4.3 4.3. Bonus: dropping levels**

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

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

810 ## [1] Santa Cruz

811 ## Levels: Santa Cruz Isabela

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

812 ## [1] "Santa Cruz" "Isabela"

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

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

815 ## [1] "Santa Cruz"

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

819 **2.5 5. External resources**

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

- 821 • The `readr/tibble/tidyr` and `forcats` cheatsheets.
- 822 • This link on the concept of tidy data
- 823 • 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
```

```

843 ##   <chr>      <chr>      <chr>      <chr>      <chr>      <dbl> <dbl>
844 ## 1 Abditomys_l~ Rodentia Muridae  Abditomys latidens      1      0
845 ## 2 Abeomelomys~ Rodentia Muridae  Abeomelo~ sevia      1      0
846 ## 3 Abrawayaomy~ Rodentia Cricetidae Abrawaya~ ruschii      1      0
847 ## 4 Abrocoma_be~ Rodentia Abrocomid~ Abrocoma bennettii      1      0
848 ## 5 Abrocoma_bo~ Rodentia Abrocomid~ Abrocoma boliviensis      1      0
849 ## 6 Abrocoma_bu~ Rodentia Abrocomid~ Abrocoma budini      1      0
850 ## 7 Abrocoma_ci~ Rodentia Abrocomid~ Abrocoma cinerea      1      0
851 ## 8 Abrocoma_fa~ Rodentia Abrocomid~ Abrocoma famatina      1      0
852 ## 9 Abrocoma_sh~ Rodentia Abrocomid~ Abrocoma shistacea      1      0
853 ## 10 Abrocoma_us~ Rodentia Abrocomid~ Abrocoma uspollata      1      0
854 ## # ... with 5,821 more rows, and 17 more variables: Freshwater <dbl>,
855 ## #   Aerial <dbl>, Life.Habit.Method <chr>, Life.Habit.Source <chr>,
856 ## #   Mass.g <dbl>, Mass.Method <chr>, Mass.Source <chr>, Mass.Comparison <chr>,
857 ## #   Mass.Comparison.Source <chr>, Island.Endemicity <chr>,
858 ## #   IUCN.Status.1.2 <chr>, Added.IUCN.Status.1.2 <chr>, Diet.Plant <dbl>,
859 ## #   Diet.Vertebrate <dbl>, Diet.Invertebrate <dbl>, Diet.Method <chr>,
860 ## #   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
  )

```

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

```

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

```

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

880 Finally, I should mention that you can refer to variables names either with strings or di-
 881 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"
  )

```

)

882 **3.3 Select variables with `select()`**

883 **3.4 Select observations with `filter()`**

884 **3.5 Create new variables with `mutate()`**

885 can also edit existing ones

886 drop existing variables with `transmute()`

887 **3.6 Grouped results with `group_by()` and `summarise()`**

888 **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")))
```

889 **3.8 More !**

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

891

Chapter 4

Working with lists and iteration

Every use case is ridiculous
until it happens to you.

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

4.1 Basic iteration with map

Iteration in base R is commonly done with `for` and `while` loops. There is no readymade alternative to `while` loops in the tidyverse. However, the functionality of `for` loops is spread over the `map` family of functions.

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

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

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

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

907 4.1.1 `map` basic use

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

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

911 ## [[1]]
912 ## [1] 1
913 ##
914 ## [[2]]
915 ## [1] 1.414214
916 ##
917 ## [[3]]
918 ## [1] 1.732051
919 ##
920 ## [[4]]
921 ## [1] 2
922 ##
923 ## [[5]]
924 ## [1] 2.236068
925 ##
926 ## [[6]]
927 ## [1] 2.44949
928 ##
929 ## [[7]]
930 ## [1] 2.645751
931 ##
932 ## [[8]]
933 ## [1] 2.828427
934 ##
935 ## [[9]]
936 ## [1] 3
937 ##
938 ## [[10]]
939 ## [1] 3.162278
```

4.1.2 map variants returning vectors

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

```
# use map_dbl to get a vector of square roots
some_numbers = 1:10
map_dbl(some_numbers, sqrt)

## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
## [9] 3.000000 3.162278

# map_chr will convert the output to a character
map_chr(some_numbers, sqrt)

## [1] "1.000000" "1.414214" "1.732051" "2.000000" "2.236068" "2.449490"
## [7] "2.645751" "2.828427" "3.000000" "3.162278"

# map_int will NOT round the output to an integer

# 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
```

Integrating map and tidyr::nest

The example show how each map variant can be used. This integrates `tidyr::nest` with `map`, and the two are especially complementary.

```
# nest mtcars into a list of dataframes based on number of cylinders
some_data = as_tibble(mtcars, rownames = "car_name") %>%
  group_by(cyl) %>%
  nest()

# get the number of rows per dataframe
# the mean mileage
# and the first car

some_data = some_data %>%
  mutate(n_rows = map_int(data, nrow),
         mean_mpg = map_dbl(data, ~mean(.$mpg)),
         first_car = map_chr(data, ~first(.$car_name)))

some_data

## # A tibble: 3 x 5
## # Groups:   cyl [3]
##   cyl data              n_rows mean_mpg first_car

```

```

955 ##      <dbl> <list>                <int>      <dbl> <chr>
956 ## 1      6 <tibble [7 x 11]>          7      19.7 Mazda RX4
957 ## 2      4 <tibble [11 x 11]>         11      26.7 Datsun 710
958 ## 3      8 <tibble [14 x 11]>         14      15.1 Hornet Sportabout

```

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

961 4.1.3 `map` variants returning dataframes

962 `map_df` returns data frames, and by default binds dataframes by rows, while `map_dfr` does
 963 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)

```

```

964 ##      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
965 ## 1  22.8   4 108.0  93 3.85 2.320 18.61  1  1   4    1
966 ## 2  24.4   4 146.7  62 3.69 3.190 20.00  1  0   4    2
967 ## 3  21.0   6 160.0 110 3.90 2.620 16.46  0  1   4    4
968 ## 4  21.0   6 160.0 110 3.90 2.875 17.02  0  1   4    4
969 ## 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0   3    2
970 ## 6  14.3   8 360.0 245 3.21 3.570 15.84  0  0   3    4

```

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

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

```

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

```

```

974 ##      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
975 ## 1  22.8   4 108.0  93 3.85 2.320 18.61  1  1   4    1
976 ## 2  24.4   4 146.7  62 3.69 3.190 20.00  1  0   4    2
977 ## 3  21.0   6 160.0 110 3.90 2.620 16.46  0  1   4    4
978 ## 4  21.0   6 160.0 110 3.90 2.875 17.02  0  1   4    4
979 ## 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0   3    2
980 ## 6  14.3   8 360.0 245 3.21 3.570 15.84  0  0   3    4

```

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

```

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

```



```

983 ##   mpg cyl  disp hp drat   wt  qsec vs am gear carb mpg1 cyl1 disp1 hp1 drat1
984 ## 1 22.8   4 108.0 93 3.85 2.32 18.61 1 1   4    1  21    6  160 110   3.9
985 ## 2 24.4   4 146.7 62 3.69 3.19 20.00 1 0   4    2  21    6  160 110   3.9
986 ##   wt1 qsec1 vs1 am1 gear1 carb1 mpg2 cyl2 disp2 hp2 drat2 wt2 qsec2 vs2 am2
987 ## 1 2.620 16.46  0  1    4    4 18.7   8  360 175  3.15 3.44 17.02  0  0
988 ## 2 2.875 17.02  0  1    4    4 14.3   8  360 245  3.21 3.57 15.84  0  0
989 ##   gear2 carb2
990 ## 1       3     2
991 ## 2       3     4

```

992 4.1.4 Selective mapping

993 • map_at and map_if

994 4.2 More map variants

995 4.2.1 map2

996 imap here

997 4.2.2 pmap

998 4.2.3 walk

999 walk2 and pwalk

1000 4.3 Modification in place

1001 modify

1002 4.4 Working with lists

1003 4.4.1 Filtering lists

1004 4.4.2 Summarising lists

1005 4.4.3 Reduction and accumulation

1006 4.4.4 Miscellaneous operation