

# TRES Tidyverse Tutorial

Raphael, Pratik and Theo

2020-05-31



# Contents

|    |   |           |
|----|---|-----------|
| 2  | <b>Outline</b>  | <b>5</b>  |
| 3  | About . . . . .   | 5         |
| 4  | Schedule . . . . .  | 5         |
| 5  | Possible extras . . . . .                                   | 6         |
| 6  | Join . . . . .  | 6         |
| 7  | <b>1 Reading files and string manipulation</b>              | <b>7</b>  |
| 8  | 1.1 Data import and export with <b>readr</b> . . . . .      | 7         |
| 9  | 1.2 String manipulation with <b>stringr</b> . . . . .       | 10        |
| 10 | 1.3 String interpolation with <b>glue</b> . . . . .         | 19        |
| 11 | 1.4 Strings in <b>ggplot</b> . . . . .                      | 20        |
| 12 | <b>2 Reshaping data tables in the tidyverse</b>             | <b>23</b> |
| 13 | 2.1 The new data frame: <b>tibble</b> . . . . .             | 24        |
| 14 | 2.2 The concept of tidy data . . . . .                      | 26        |
| 15 | 2.3 Reshaping with <b>tidyr</b> . . . . .                   | 29        |
| 16 | 2.4 Extra: factors and the <b>forcats</b> package . . . . . | 35        |
| 17 | 2.5 External resources . . . . .                            | 39        |
| 18 | <b>3 Working with lists and iteration</b>                   | <b>41</b> |
| 19 | 3.1 Basic iteration with <b>map</b> . . . . .               | 42        |
| 20 | 3.2 More <b>map</b> variants . . . . .                      | 45        |
| 21 | 3.3 Modification in place . . . . .                         | 46        |
| 22 | 3.4 Working with lists . . . . .                            | 46        |



# 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 |

## 35 Possible extras

- 36 • Reproducibility and package-making (with e.g. usethis)
- 37
- 38 • Embedding C++ code with Rcpp

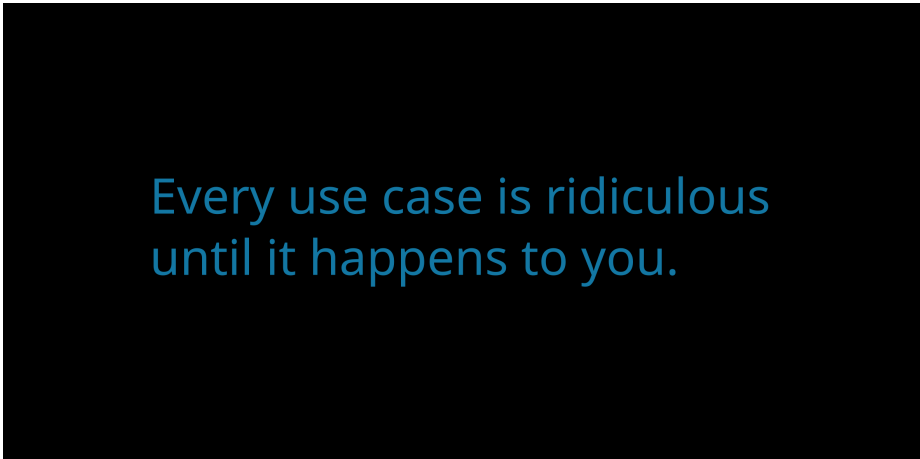
## 39 Join

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

41 \*Tentative dates.

## Chapter 1

# Reading files and string manipulation



Every use case is ridiculous  
until it happens to you.

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

### 1.1 Data import and export with readr

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

## 51 1.1.1 Reading data

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

```

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

56 # read the file in
some_example = read_csv(some_example)

57 ## Parsed with column specification:
58 ## cols(
59 ##   mpg = col_double(),
60 ##   cyl = col_double(),
61 ##   disp = col_double(),
62 ##   hp = col_double(),
63 ##   drat = col_double(),
64 ##   wt = col_double(),
65 ##   qsec = col_double(),
66 ##   vs = col_double(),
67 ##   am = col_double(),
68 ##   gear = col_double(),
69 ##   carb = col_double()
70 ## )

71 head(some_example)

72 ## # A tibble: 6 x 11
73 ##   mpg   cyl  disp    hp  drat    wt  qsec    vs  am  gear  carb
74 ##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
75 ## 1  21     6   160   110  3.9   2.62  16.5    0    1     4     4
76 ## 2  21     6   160   110  3.9   2.88  17.0    0    1     4     4
77 ## 3 22.8    4   108    93  3.85  2.32  18.6    1    1     4     1
78 ## 4 21.4    6   258   110  3.08  3.22  19.4    1    0     3     1
79 ## 5 18.7    8   360   175  3.15  3.44  17.0    0    0     3     2
80 ## 6 18.1    6   225   105  2.76  3.46  20.2    1    0     3     1

```

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

81 Other variants include:

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



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

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

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

### 93 1.1.2 Writing data

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

98 These functions are not covered here.

### 99 1.1.3 Reading and writing lines

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

105 This can be done using the `readr` functions `read_lines` and `write_lines`. Con-  
 106 sider the model summary from a simple linear model.

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

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

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

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

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

```

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

# use cat to show the model output as it would be on screen
cat(model_output, sep = "\n")

```

```

112 ##
113 ## Call:
114 ## lm(formula = mpg ~ wt, data = mtcars)
115 ##
116 ## Residuals:
117 ##      Min       1Q   Median       3Q      Max
118 ## -4.5432 -2.3647 -0.1252  1.4096  6.8727
119 ##
120 ## Coefficients:
121 ##              Estimate Std. Error t value Pr(>|t|)
122 ## (Intercept)  37.2851     1.8776  19.858 < 2e-16 ***
123 ## wt          -5.3445     0.5591  -9.559 1.29e-10 ***
124 ## ---
125 ## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
126 ##
127 ## Residual standard error: 3.046 on 30 degrees of freedom
128 ## Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
129 ## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10

```

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

## 134 Excel files

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

## 140 1.2 String manipulation with `stringr`

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

145 `stringr` functions begin with `str_`.

### 146 1.2.1 Putting strings together

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

```
148 # str_c works like paste(), choose a separator
149 str_c("this string", "this other string", sep = "_")

150 ## [1] "this string_this other string"

151 # str_dup works like rep
152 str_dup("this string", times = 3)

153 ## [1] "this stringthis stringthis string"

154 # str_flatten works on lists and vectors
155 str_flatten(string = as.list(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(string = letters, collapse = "-")

158 ## [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"

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

161 # get the class of a tibble and display it as a single string
162 class_tibble = class(tibble::tibble(a = 1))
163 str_flatten(string = class_tibble, collapse = ", ")

164 ## [1] "tbl_df, tbl, data.frame"
```

### 156 1.2.2 Detecting strings

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

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

```
161 # there should be 5 a-s here
162 str_count(string = "ababababa", pattern = "a")

163 ## [1] 5

164 # vectorise over the input string
165 # should return a vector of length 2, with integers 5 and 3
166 str_count(string = c("ababbababa", "banana"), pattern = "a")
```

```

162 ## [1] 5 3

      # vectorise over the pattern to count both a-s and b-s
      str_count(string = "ababababa", pattern = c("a", "b"))

163 ## [1] 5 4

164 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"))

165 ## [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"))

166 ## [1] 5 1 4 3

167 str_locate locates the search pattern in a string, and returns the start and
168 end as a two column matrix.

      # the behaviour of both str_locate and str_locate_all is
      # to find the first match by default
      str_locate(string = "banana", pattern = "ana")

169 ##          start end
170 ## [1,]         2   4

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

171 ## [1] TRUE

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

172 ## [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")))

173 ## [1] TRUE

```

174 Detect whether a string starts or ends with a pattern. Also vectorised. Both  
 175 have a `negate` argument, which returns the negative, i.e., returns `FALSE` if the  
 176 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")

177 ## [1] FALSE FALSE  TRUE  TRUE

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

178 ## [1]  TRUE FALSE FALSE  TRUE

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

179 ## [1] FALSE  TRUE  TRUE FALSE

str_subset [WHICH IS NOT RELATED TO str_sub] helps with subsetting a
180 character vector based on a str_detect predicate. In the example, all elements
181 containing “banana” are subset.

182 str_which has the same logic except that it returns the vector position and not
183 the elements.

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

185 ## [1] "banana"                "bananageddon is coming"

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

186 ## [1] 1 2
```

### 1.2.3 Matching strings

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

190 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")
```

```
191 ##      [,1]
192 ## [1,] "banana"
193 ## [2,] "banana"
194 ## [3,] "banana"
```

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

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

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

```
207 ##      [,1]      [,2]  [,3]
208 ## [1,] "1970-somemonth" "1970" "somemonth"
209 ## [2,] "1990-anothermonth" "1990" "anothermonth"
210 ## [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+)")
```

```
211 ##      [,1]      [,2]  [,3] [,4]
212 ## [1,] "1970-somemonth" "1970" "-" "somemonth"
213 ## [2,] "1990-anothermonth" "1990" "-" "anothermonth"
214 ## [3,] "2010-thismonth" "2010" "-" "thismonth"
```

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

218 The return type is a list, with one element per input string. Each element is a  
 219 character matrix, where each row is one possible match, and each column after  
 220 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]]
##      [,1]      [,2]  [,3]
## [1,] "1970-somemonth" "1970" "somemonth"
## [2,] "1990/anothermonth" "1990" "anothermonth"
##
## [[2]]
##      [,1]      [,2]  [,3]
## [1,] "1990-somemonth" "1990" "somemonth"
## [2,] "2001/anothermonth" "2001" "anothermonth"

```

#### 234 1.2.4 Simpler pattern extraction

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

238 `str_extract` returns a character vector with the same length as the input string  
 239 vector, while `str_extract_all` returns a list, with a character vector whose  
 240 elements are the matches.

```

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

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  
 249 from a path, or when working sequences separated by a known pattern generally,  
 250 `str_split` can help 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  
 258 require some knowledge of `regex`.

```

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

```

```

# not quite there
str_split(filename, pattern = "_")

```

```

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 splits the string into as many pieces as specified, and can be
267 especially useful dealing with filepaths.

```

```

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

```

```

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  
 272 into the task of recovering simulation parameters or other data from regularly  
 273 named files. `str_replace_all` works the same way but replaces all matches of  
 274 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  
 277 `""`. This is not covered here.

278 Having replaced unwanted characters in the filename with spaces, `str_trim`  
 279 offers a way to remove leading and trailing whitespaces.

```

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

filename_without_spaces = str_trim(filename_with_spaces)
filename_without_spaces

```

```

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  
 285 `str_sub`. In the date-time example, the year is always represented by the first  
 286 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)
```

```
287 ## [1] "1970" "1990" "2010"
```

288 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)
```

```
289 ## [1] "01" "21" "31"
```

290 Finally, it's also possible to replace characters within a string based on the  
 291 position. This requires using the assignment operator `<-`.

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

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

```
date_times
```

```
292 ## [1] "1970-somemonth-01" "1990-anothermonth-01" "2010-thismonth-01"
```

### 293 1.2.8 Padding and truncating strings

294 Strings included in filenames or plots are often of unequal lengths, especially  
 295 when they represent numbers. `str_pad` can pad strings with suitable characters  
 296 to maintain equal length filenames, with which it is easier to work.

```
# pad so all values have three digits
str_pad(string = c("1", "10", "100"),
        width = 3,
```

```

    side = "left",
    pad = "0")
297 ## [1] "001" "010" "100"
298 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.")
299 ## [1] "bananas are great etc. etc."

```

### 1.2.9 Stringr aspects not covered here

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

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

## 1.3 String interpolation with glue

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

`glue` is as simple as the example shown.

```

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

```

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

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

```

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

323 # for command line arguments or to start multiple job scripts on the cluster
glue_data(parameter_combinations,
           'simulation-name {param1} {param2}')

324 ## simulation-name a 1
325 ## simulation-name b 2
326 ## simulation-name c 3
327 ## simulation-name d 4
328 ## simulation-name e 5

329 # for filenames
glue_data(parameter_combinations,
           'sim_data_param1_{param1}_param2_{param2}.ext')

330 ## sim_data_param1_a_param2_1.ext
331 ## sim_data_param1_b_param2_2.ext
332 ## sim_data_param1_c_param2_3.ext
333 ## sim_data_param1_d_param2_4.ext
334 ## sim_data_param1_e_param2_5.ext

```

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

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

## 338 1.4 Strings in ggplot

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

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

347 More examples can be found on the ggrep1 website.

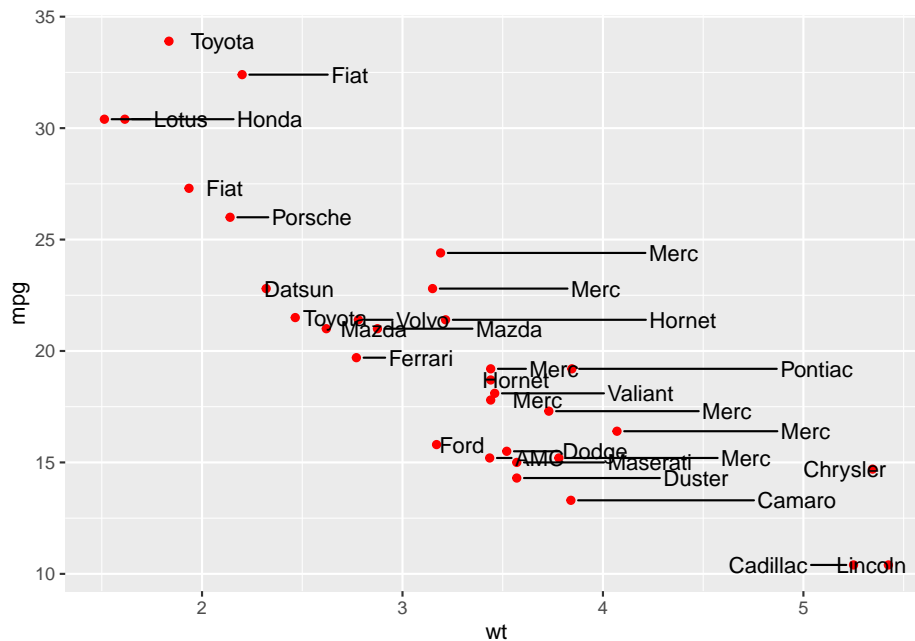
348 Here, the arguments to `geom_text_repel` are taken both from the `mtcars` data  
 349 (position), as well as from the car brands extracted using the `stringr::word`  
 350 (labels), which tries to separate strings based on a regular pattern.

351 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)
```



352

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

356 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 subsetting, among others. Most functions working with `data.frame` will work with `tibble` and vice versa. Use the `as*` family of functions to switch back and forth between the two if needed, using e.g. `as.data.frame` or `as_tibble`.

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

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

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

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



```

395 ## 2 Theo    3
396 ## 3 Raph    2, 5

```

397 As a rule of thumb, try to convert your tables to tibbles whenever you can,  
 398 especially when the original table is *not* a data frame. For example, the prin-  
 399 cipal component analysis function `prcomp` outputs a `matrix` of coordinates in  
 400 principal component-space.

```

# Perform a PCA on mtcars
pca_scores <- prcomp(mtcars)$x
head(pca_scores) # looks like a data frame or a tibble...

401 ##           PC1      PC2      PC3      PC4      PC5
402 ## Mazda RX4      -79.596425  2.132241 -2.153336 -2.7073437 -0.7023522
403 ## Mazda RX4 Wag  -79.598570  2.147487 -2.215124 -2.1782888 -0.8843859
404 ## Datsun 710      -133.894096 -5.057570 -2.137950  0.3460330  1.1061111
405 ## Hornet 4 Drive    8.516559 44.985630  1.233763  0.8273631  0.4240145
406 ## Hornet Sportabout 128.686342 30.817402  3.343421 -0.5211000  0.7365801
407 ## Valiant        -23.220146 35.106518 -3.259562  1.4005360  0.8029768
408 ##           PC6      PC7      PC8      PC9      PC10
409 ## Mazda RX4      -0.31486106 -0.098695018 -0.07789812 -0.2000092 -0.29008191
410 ## Mazda RX4 Wag  -0.45343873 -0.003554594 -0.09566630 -0.3533243 -0.19283553
411 ## Datsun 710      1.17298584  0.005755581  0.13624782 -0.1976423  0.07634353
412 ## Hornet 4 Drive  -0.05789705 -0.024307168  0.22120800  0.3559844 -0.09057039
413 ## Hornet Sportabout -0.33290957  0.106304777 -0.05301719  0.1532714 -0.18862217
414 ## Valiant        -0.08837864  0.238946304  0.42390551  0.1012944 -0.03769010
415 ##           PC11
416 ## Mazda RX4      0.1057706
417 ## Mazda RX4 Wag  0.1069047
418 ## Datsun 710      0.2668713
419 ## Hornet 4 Drive  0.2088354
420 ## Hornet Sportabout -0.1092563
421 ## Valiant        0.2757693

class(pca_scores) # but is actually a matrix

422 ## [1] "matrix"

# Convert to tibble
as_tibble(pca_scores)

423 ## # A tibble: 32 x 11
424 ##           PC1      PC2      PC3      PC4      PC5      PC6      PC7      PC8      PC9      PC10
425 ##           <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
426 ## 1  -79.6      2.13 -2.15 -2.71 -0.702 -0.315 -0.0987 -0.0779 -0.200 -0.290
427 ## 2  -79.6      2.15 -2.22 -2.18 -0.884 -0.453 -0.00355 -0.0957 -0.353 -0.193
428 ## 3 -134.     -5.06 -2.14  0.346  1.11  1.17  0.00576  0.136 -0.198  0.0763
429 ## 4   8.52    45.0  1.23  0.827  0.424 -0.0579 -0.0243  0.221  0.356 -0.0906
430 ## 5  129.     30.8  3.34 -0.521  0.737 -0.333  0.106 -0.0530  0.153 -0.189

```

```

431 ## 6 -23.2 35.1 -3.26 1.40 0.803 -0.0884 0.239 0.424 0.101 -0.0377
432 ## 7 159. -32.3 0.649 0.199 0.786 0.0687 -0.530 -0.0593 0.221 -0.313
433 ## 8 -113. 39.7 -0.465 0.338 -1.24 0.280 -0.146 0.320 0.279 0.190
434 ## 9 -104. 7.51 -1.59 4.02 -1.14 0.0279 0.595 -0.233 -0.126 -0.349
435 ## 10 -67.0 -6.21 -3.61 -0.320 -0.960 -0.529 -0.0174 -0.182 0.543 0.412
436 ## # ... with 22 more rows, and 1 more variable: PC11 <dbl>

```

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

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

## 442 2.2 The concept of tidy data

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

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

```

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

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

# Snapshot
data

451 ## # A tibble: 100 x 5
452 ##       id beak_length beak_width body_weight island

```

```

453 ##      <int>      <dbl>      <dbl>      <dbl> <chr>
454 ##    1      1      5.02      1.97      10.5 Isabela
455 ##    2      2      5.06      2.04      16.9 Isabela
456 ##    3      3      4.86      1.97      11.5 Isabela
457 ##    4      4      4.91      2.00       5.73 Isabela
458 ##    5      5      5.11      2.06      11.7 Isabela
459 ##    6      6      5.06      1.94      12.5 Isabela
460 ##    7      7      4.98      2.02       8.29 Isabela
461 ##    8      8      5.01      2.12       4.63 Isabela
462 ##    9      9      4.95      2.02      14.0 Isabela
463 ##   10     10      4.79      1.98       8.71 Isabela
464 ## # ... with 90 more rows

```

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

The tidy equivalent of this dataset would be:

```

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

471 ## # A tibble: 300 x 4
472 ##       id island variable    value
473 ##   <int> <chr>   <chr>      <dbl>
474 ## 1     1 Isabela beak_length  5.02
475 ## 2     1 Isabela beak_width   1.97
476 ## 3     1 Isabela body_weight 10.5
477 ## 4     2 Isabela beak_length  5.06
478 ## 5     2 Isabela beak_width   2.04
479 ## 6     2 Isabela body_weight 16.9
480 ## 7     3 Isabela beak_length  4.86
481 ## 8     3 Isabela beak_width   1.97
482 ## 9     3 Isabela body_weight 11.5
483 ## 10    4 Isabela beak_length  4.91
484 ## # ... with 290 more rows

```

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

As you can see our tibble now has three times as many rows and fewer columns. This format is rather unintuitive and not optimal for display. However, it provides a very standardized and consistent way of organizing data that will be

understood (and expected) by pretty much all functions in the tidyverse. This makes the tidyverse tools work well together and reduces the time you would otherwise spend reformatting your data from one tool to the next.

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

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

```
# Make a contingency table
ctg <- with(data, table(island, variable))
ctg
```

|            | variable                           |
|------------|------------------------------------|
| island     | beak_length beak_width body_weight |
| Isabela    | 50 50 50                           |
| Santa Cruz | 50 50 50                           |

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

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

| # | island     | variable    | n     |
|---|------------|-------------|-------|
| # | <chr>      | <chr>       | <int> |
| 1 | Isabela    | beak_length | 50    |
| 2 | Santa Cruz | beak_length | 50    |
| 3 | Isabela    | beak_width  | 50    |
| 4 | Santa Cruz | beak_width  | 50    |
| 5 | Isabela    | body_weight | 50    |
| 6 | Santa Cruz | body_weight | 50    |

## 522 2.3 Reshaping with `tidyr`

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

### 533 2.3.1 Pivoting

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

```
539 pivot_wider(  

  data,  

  names_from = "variable",  

  values_from = "value",  

  id_cols = c("id", "island")  

  )
```

```
## # A tibble: 100 x 5  

##       id island beak_length beak_width body_weight  

##   <int> <chr>      <dbl>      <dbl>      <dbl>  

## 1     1     1 Isabela      5.02      1.97      10.5  

## 2     2     2 Isabela      5.06      2.04      16.9  

## 3     3     3 Isabela      4.86      1.97      11.5  

## 4     4     4 Isabela      4.91      2.00       5.73  

## 5     5     5 Isabela      5.11      2.06      11.7  

## 6     6     6 Isabela      5.06      1.94      12.5  

## 7     7     7 Isabela      4.98      2.02       8.29  

## 8     8     8 Isabela      5.01      2.12       4.63  

## 9     9     9 Isabela      4.95      2.02      14.0  

## 10    10    10 Isabela      4.79      1.98      8.71  

## # ... with 90 more rows
```

553 The order of the columns is not exactly as it was, but this should not matter in  
 554 a data analysis pipeline where you should access columns by their names. It is

straightforward to change the order of the columns, but this is more within the scope of the `dplyr` package.

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

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

### 2.3.2 Handling missing values

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

```
# We replace 100 random observations by NAs
ii <- sample(nrow(data), 100)
data$value[ii] <- NA
data
```

```
## # A tibble: 300 x 4
##       id island variable    value
##   <int> <chr>   <chr>    <dbl>
## 1     1     1 Isabela beak_length  5.02
## 2     2     1 Isabela beak_width   NA
## 3     3     1 Isabela body_weight 10.5
## 4     4     2 Isabela beak_length  5.06
## 5     5     2 Isabela beak_width   2.04
## 6     6     2 Isabela body_weight 16.9
## 7     7     3 Isabela beak_length  4.86
## 8     8     3 Isabela beak_width   1.97
## 9     9     3 Isabela body_weight   NA
## 10    10     4 Isabela beak_length  4.91
## # ... 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  5.02
## 2     2     1 Isabela body_weight 10.5
## 3     3     2 Isabela beak_length  5.06
## 4     4     2 Isabela beak_width   2.04
## 5     5     2 Isabela body_weight 16.9
## 6     6     3 Isabela beak_length  4.86
## 7     7     3 Isabela beak_width   1.97
```

```

589 ## 8      4 Isabela beak_length  4.91
590 ## 9      4 Isabela beak_width   2.00
591 ## 10     4 Isabela body_weight  5.73
592 ## # ... with 190 more rows

```

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

```

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

594 ## # A tibble: 300 x 4
595 ##       id island variable      value
596 ##   <int> <chr>   <chr>      <dbl>
597 ## 1     1  1 Isabela beak_length  5.02
598 ## 2     1  1 Isabela beak_width -999
599 ## 3     1  1 Isabela body_weight 10.5
600 ## 4     2  2 Isabela beak_length  5.06
601 ## 5     2  2 Isabela beak_width   2.04
602 ## 6     2  2 Isabela body_weight 16.9
603 ## 7     3  3 Isabela beak_length  4.86
604 ## 8     3  3 Isabela beak_width   1.97
605 ## 9     3  3 Isabela body_weight -999
606 ## 10    4  4 Isabela beak_length  4.91
607 ## # ... with 290 more rows

```

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

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

```

      fill(data, value)

611 ## # A tibble: 300 x 4
612 ##       id island variable      value
613 ##   <int> <chr>   <chr>      <dbl>
614 ## 1     1  1 Isabela beak_length  5.02
615 ## 2     1  1 Isabela beak_width  5.02
616 ## 3     1  1 Isabela body_weight 10.5
617 ## 4     2  2 Isabela beak_length  5.06
618 ## 5     2  2 Isabela beak_width   2.04
619 ## 6     2  2 Isabela body_weight 16.9
620 ## 7     3  3 Isabela beak_length  4.86
621 ## 8     3  3 Isabela beak_width   1.97
622 ## 9     3  3 Isabela body_weight  1.97
623 ## 10    4  4 Isabela beak_length  4.91
624 ## # ... with 290 more rows

```

625 Note that most functions in the tidyverse take a tibble as their first argument,  
 626 and columns to which to apply the functions are usually passed as “objects”  
 627 rather than character strings. In the above example, we passed the `value`

column as `value`, not `"value"`. These column-objects are called by the tidyverse functions *in the context* of the data (the tibble) they belong to.

### 2.3.3 Splitting and combining cells

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

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

```
# Sample random dates for each observation
data$day <- sample(30, nrow(data), replace = TRUE)
data$month <- sample(12, nrow(data), replace = TRUE)
data$year <- sample(2019:2020, nrow(data), replace = TRUE)
data
```

```
## # A tibble: 300 x 7
##       id island variable    value  day month  year
##   <int> <chr>   <chr>    <dbl> <int> <int> <int>
## 1     1     1 Isabela beak_length  5.02   25     4  2020
## 2     2     1 Isabela beak_width   NA    12    10  2019
## 3     3     1 Isabela body_weight 10.5     8     2  2019
## 4     4     2 Isabela beak_length  5.06   26     3  2020
## 5     5     2 Isabela beak_width   2.04    4     4  2020
## 6     6     2 Isabela body_weight 16.9    28    10  2020
## 7     7     3 Isabela beak_length  4.86   16    12  2020
## 8     8     3 Isabela beak_width   1.97    2    11  2019
## 9     9     3 Isabela body_weight   NA    19     2  2019
## 10    10     4 Isabela beak_length  4.91   12     4  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  5.02 25-4-2020
## 2     2     1 Isabela beak_width   NA 12-10-2019
## 3     3     1 Isabela body_weight 10.5 8-2-2019
## 4     4     2 Isabela beak_length  5.06 26-3-2020
## 5     5     2 Isabela beak_width   2.04 4-4-2020
## 6     6     2 Isabela body_weight 16.9 28-10-2020
```



```

661 ## 7      3 Isabela beak_length  4.86 16-12-2020
662 ## 8      3 Isabela beak_width   1.97 2-11-2019
663 ## 9      3 Isabela body_weight NA   19-2-2019
664 ## 10     4 Isabela beak_length  4.91 12-4-2019
665 ## # ... 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"))

668 ## # A tibble: 300 x 7
669 ##       id island variable    value day  month year
670 ##   <int> <chr>   <chr>    <dbl> <chr> <chr> <chr>
671 ## 1     1     1 Isabela beak_length  5.02 25    4    2020
672 ## 2     1     1 Isabela beak_width   NA  12   10    2019
673 ## 3     1     1 Isabela body_weight 10.5  8    2    2019
674 ## 4     2     2 Isabela beak_length  5.06 26    3    2020
675 ## 5     2     2 Isabela beak_width   2.04 4     4    2020
676 ## 6     2     2 Isabela body_weight 16.9 28   10    2020
677 ## 7     3     3 Isabela beak_length  4.86 16   12    2020
678 ## 8     3     3 Isabela beak_width   1.97 2    11    2019
679 ## 9     3     3 Isabela body_weight NA   19    2    2019
680 ## 10    4     4 Isabela beak_length  4.91 12    4    2019
681 ## # ... 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)

686 ## # A tibble: 900 x 5
687 ##       id island variable    value date
688 ##   <int> <chr>   <chr>    <dbl> <chr>
689 ## 1     1     1 Isabela beak_length  5.02 25
690 ## 2     1     1 Isabela beak_length  5.02 4
691 ## 3     1     1 Isabela beak_length  5.02 2020
692 ## 4     1     1 Isabela beak_width   NA  12
693 ## 5     1     1 Isabela beak_width   NA  10
694 ## 6     1     1 Isabela beak_width   NA  2019
695 ## 7     1     1 Isabela body_weight 10.5  8
696 ## 8     1     1 Isabela body_weight 10.5  2
697 ## 9     1     1 Isabela body_weight 10.5 2019
698 ## 10    2     2 Isabela beak_length  5.06 26
699 ## # ... 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")
)

## # A tibble: 6 x 2
##   island      variable
##   <chr>      <chr>
## 1 Isabela    beak_length
## 2 Isabela    beak_width
## 3 Isabela    body_weight
## 4 Santa Cruz beak_length
## 5 Santa Cruz beak_width
## 6 Santa Cruz body_weight
```

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

```
expand(data, island, variable)

## # A tibble: 6 x 2
##   island      variable
##   <chr>      <chr>
## 1 Isabela    beak_length
## 2 Isabela    beak_width
## 3 Isabela    body_weight
## 4 Santa Cruz beak_length
## 5 Santa Cruz beak_width
## 6 Santa Cruz body_weight
```

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

```
complete(data, island, variable)

## # A tibble: 300 x 5
##   island variable      id value date
##   <chr>  <chr>      <int> <dbl> <chr>
## 1 Isabela beak_length     1  5.02 25-4-2020
## 2 Isabela beak_length     2  5.06 26-3-2020
## 3 Isabela beak_length     3  4.86 16-12-2020
## 4 Isabela beak_length     4  4.91 12-4-2019
## 5 Isabela beak_length     5 NA    8-5-2019
```

```

734 ## 6 Isabela beak_length      6  5.06 8-3-2020
735 ## 7 Isabela beak_length      7  4.98 30-2-2019
736 ## 8 Isabela beak_length      8  5.01 20-9-2019
737 ## 9 Isabela beak_length      9  4.95 30-2-2019
738 ## 10 Isabela beak_length     10  4.79 18-7-2019
739 ## # ... with 290 more rows

```

```

740 which does nothing here because we already have all combinations of island
741 and variable.

```

### 742 2.3.5 Nesting

```

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

```

## 747 2.4 Extra: factors and the forcats package

```

library(forcats)

```

```

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

```

```

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

```

```

752 ## [1] "Pratik" "Theo"   "Raph"

```

```

753 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

```

```

754 ## [1] Pratik Theo   Raph
755 ## Levels: Pratik Raph Theo

```

```

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

```

```

factor(c(my_char_vec, "Raph"))

```

```

759 ## [1] Pratik Theo   Raph   Raph
760 ## Levels: Pratik Raph Theo

```

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

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

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

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

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

### 2.4.1 Reordering a factor

Use **fct\_relevel** to manually change the order of the levels:

```
fct_relevel(my_fact_vec, c("Pratik", "Theo", "Raph"))
## [1] Pratik Theo   Raph
## Levels: Pratik Theo Raph
```

Alternatively, use **fct\_inorder** to set the order of the levels to the order in which they appear:

```
fct_inorder(my_fact_vec)
## [1] Pratik Theo   Raph
## Levels: Pratik Theo Raph
```

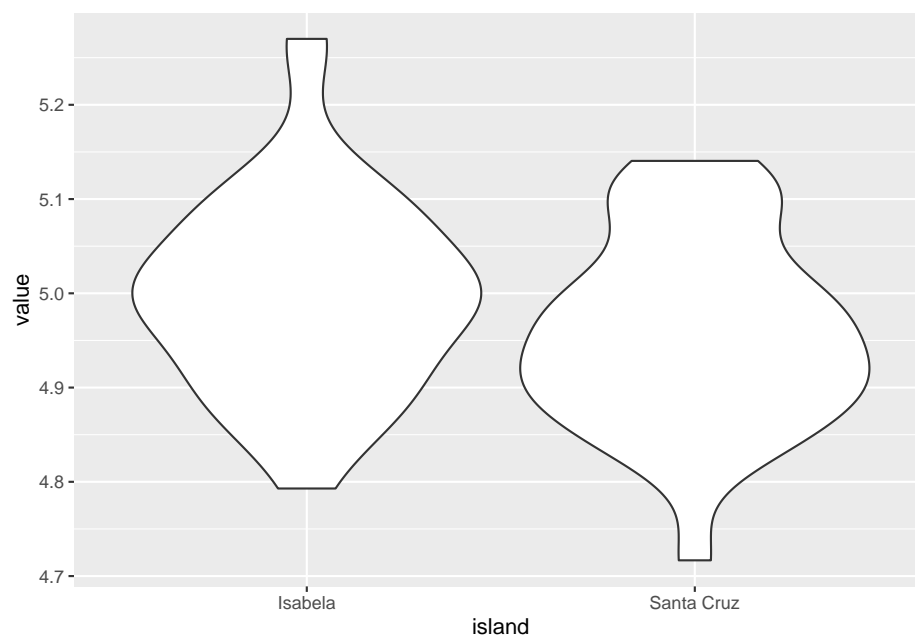
or **fct\_rev** to reverse the order of the levels:

```
fct_rev(my_fact_vec)
## [1] Pratik Theo   Raph
## Levels: Theo Raph Pratik
```

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

793 ## Warning: Removed 29 rows containing non-finite values (stat\_ydensity).



794

795 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()
```

796 ## Warning: Removed 29 rows containing non-finite values (stat\_ydensity).



797

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

## 801 2.4.2 Factor levels

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

```
803 fct_recode(  

  my_fact_vec,  

  "Pratik Gupte" = "Pratik",  

  "Theo Pannetier" = "Theo",  

  "Raphael Scherrer" = "Raph"  

  )  

804 ## [1] Pratik Gupte      Theo Pannetier    Raphael Scherrer  

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

806 or collapse factor levels together using `fct_collapse`:

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

808 ## [1] NonEU EU      EU  

809 ## Levels: NonEU EU
```

808 Again, we do not provide an exhaustive list of `forcats` functions here but the  
 809 most usual ones, to give a glimpse of many things that one can do with factors.

810 So, if you are dealing with factors, remember that `forcats` may have handy  
811 tools for you.

### 812 2.4.3 Bonus: dropping levels

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

```
data <- data[data$island == "Santa Cruz",]  
unique(data$island) # Isabela is gone from the labels  
  
816 ## [1] Santa Cruz  
817 ## Levels: Santa Cruz Isabela  
  
levels(data$island) # but not from the levels  
  
818 ## [1] "Santa Cruz" "Isabela"
```

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

```
data <- droplevels(data)  
levels(data$island)  
  
821 ## [1] "Santa Cruz"
```

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

## 825 2.5 External resources

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

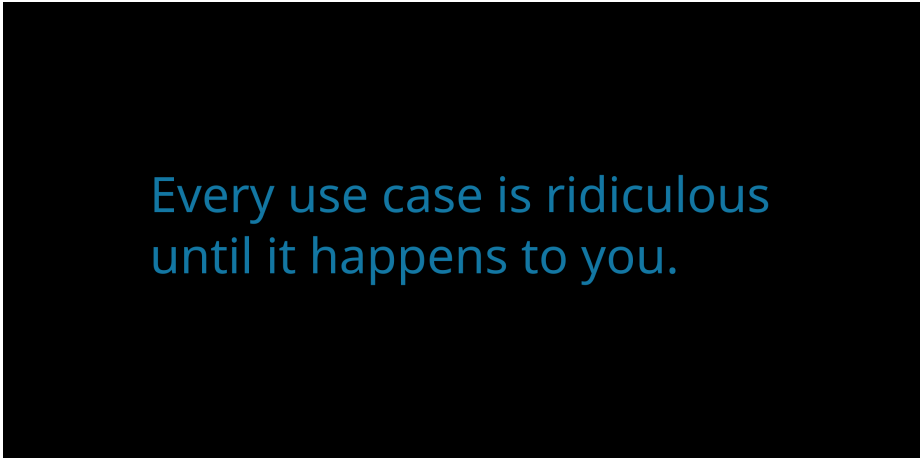
- 827 • The `readr/tibble/tidyr` and `forcats` cheatsheets.
- 828 • This link on the concept of tidy data
- 829 • The tibble, tidyr and forcats websites





## 830 Chapter 3

# 831 Working with lists and 832 iteration



Every use case is ridiculous  
until it happens to you.

```
833 # load the tidyverse  
library(tidyverse)  
  
834 ## -- Attaching packages ----- tidyverse 1.3.0 --  
  
835 ## v purrr 0.3.3      v dplyr 0.8.5  
  
836 ## -- Conflicts ----- tidyverse_conflicts() --  
837 ## x dplyr::collapse() masks glue::collapse()  
838 ## x dplyr::filter()   masks stats::filter()  
839 ## x dplyr::lag()      masks stats::lag()
```

### 840 3.1 Basic iteration with map

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

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

847 A good reason to use `purrr` functions instead of base R functions is their consis-  
848 tent and clear naming, which always indicates how they should be used. This  
849 is explained in the examples below.

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

#### 852 3.1.1 map basic use

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

```

856 # get the square root of each integer 1 - 10
857 some_numbers = 1:10
858 map(some_numbers, sqrt)
859 ## [[1]]
860 ## [1] 1
861 ##
862 ## [[2]]
863 ## [1] 1.414214
864 ##
865 ## [[3]]
866 ## [1] 1.732051
867 ##
868 ## [[4]]
869 ## [1] 2
870 ##
871 ## [[5]]
872 ## [1] 2.236068
873 ##
874 ## [[6]]
875 ## [1] 2.44949
876 ##
877 ## [[7]]
878 ## [1] 2.645751

```

```

876 ##
877 ## [[8]]
878 ## [1] 2.828427
879 ##
880 ## [[9]]
881 ## [1] 3
882 ##
883 ## [[10]]
884 ## [1] 3.162278

```

### 885 3.1.2 map variants returning vectors

886 Though `map` always returns a list, it has variants named `map_*` where the suffix  
 887 indicates the return type. `map_chr`, `map_dbl`, `map_int`, and `map_lgl` return  
 888 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)

889 ## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
890 ## [9] 3.000000 3.162278

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

891 ## [1] "1.000000" "1.414214" "1.732051" "2.000000" "2.236068" "2.449490"
892 ## [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)

893 ## [1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE

```

### 894 Integrating map and tidy::nest

895 The example show how each map variant can be used. This integrates  
 896 `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
897 ## # A tibble: 3 x 5
898 ## # Groups:   cyl [3]
899 ##   cyl data                                n_rows mean_mpg first_car
900 ##   <dbl> <list>                            <int>   <dbl> <chr>
901 ## 1     6 <tibble [7 x 11]>                     7      19.7 Mazda RX4
902 ## 2     4 <tibble [11 x 11]>                    11      26.7 Datsun 710
903 ## 3     8 <tibble [14 x 11]>                    14      15.1 Hornet Sportabout
904 map accepts multiple functions that are applied in sequence to the input list-like
905 object, but this is confusing to the reader and ill advised.

```

### 3.1.3 map variants returning dataframes

```

907 map_df returns data frames, and by default binds dataframes by rows, while
908 map_dfr does this explicitly, and map_dfc does returns a dataframe bound by
909 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)
910 ##   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
911 ## 1 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
912 ## 2 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
913 ## 3 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
914 ## 4 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
915 ## 5 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
916 ## 6 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4

```

```

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

```

```

919 map_dfr behaves the same as map_df.

```

```

# the same as above but with a pipe
some_list %>%

```

```
map_dfr(head, n = 2)
```

```
920 ##      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
921 ## 1 22.8   4 108.0  93 3.85 2.320 18.61 1 1   4   1
922 ## 2 24.4   4 146.7  62 3.69 3.190 20.00 1 0   4   2
923 ## 3 21.0   6 160.0 110 3.90 2.620 16.46 0 1   4   4
924 ## 4 21.0   6 160.0 110 3.90 2.875 17.02 0 1   4   4
925 ## 5 18.7   8 360.0 175 3.15 3.440 17.02 0 0   3   2
926 ## 6 14.3   8 360.0 245 3.21 3.570 15.84 0 0   3   4
```

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

```
some_list %>%
```

```
  map_dfc(head, n = 2)
```

```
929 ##      mpg cyl  disp hp drat    wt  qsec vs am gear carb mpg1 cyl1 disp1 hp1 drat1
930 ## 1 22.8   4 108.0  93 3.85 2.32 18.61 1 1   4   1  21   6  160 110   3.9
931 ## 2 24.4   4 146.7  62 3.69 3.19 20.00 1 0   4   2  21   6  160 110   3.9
932 ##      wt1 qsec1 vs1 am1 gear1 carb1 mpg2 cyl2 disp2 hp2 drat2 wt2 qsec2 vs2 am2
933 ## 1 2.620 16.46   0   1     4     4 18.7   8  360 175  3.15 3.44 17.02   0   0
934 ## 2 2.875 17.02   0   1     4     4 14.3   8  360 245  3.21 3.57 15.84   0   0
935 ##      gear2 carb2
936 ## 1         3     2
937 ## 2         3     4
```

### 938 3.1.4 Selective mapping

- 939 • `map_at` and `map_if`

## 940 3.2 More map variants

### 941 3.2.1 `map2`

942 `imap` here

### 943 3.2.2 `pmap`

### 944 3.2.3 `walk`

945 `walk2` and `pwalk`

### 946 **3.3** Modification in place

947 `modify`

## 948 **3.4** Working with lists

### 949 **3.4.1** Filtering lists

### 950 **3.4.2** Summarising lists

### 951 **3.4.3** Reduction and accumulation

### 952 **3.4.4** Miscellaneous operation