

TRES Tidyverse Tutorial

Raphael and Pratik

2020-05-30

Contents

2	Outline	5
3	About	5
4	Schedule	5
5	Possible extras	6
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 Working with lists and iteration	23
13	2.1 Basic iteration with <code>map</code>	24
14	2.2 More <code>map</code> variants	27
15	2.3 Modification in place	27
16	2.4 Working with lists	28

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	Raphael + 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

29 **Possible extras**

- 30 • Reproducibility and package-making (with e.g. usethis)
- 31
- 32 • Embedding C++ code with Rcpp

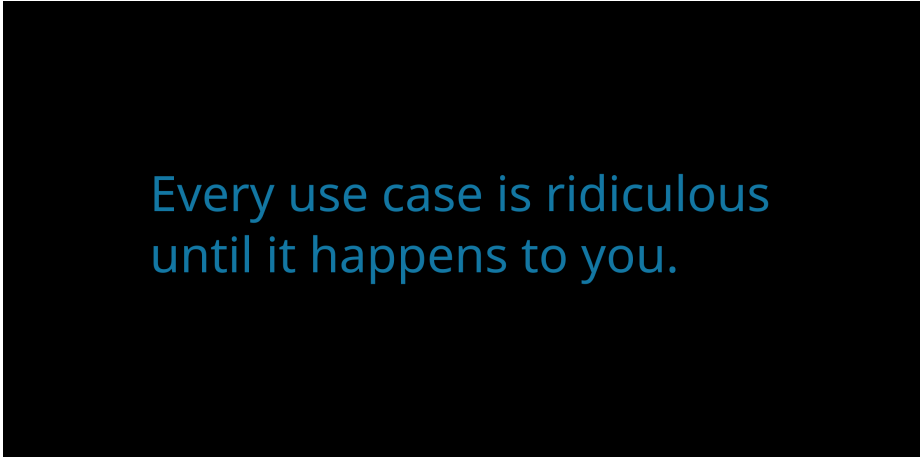
33 **Join**

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

35 *Tentative dates.

36 Chapter 1

37 Reading files and string 38 manipulation



Every use case is ridiculous
until it happens to you.

39

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

40 1.1 Data import and export with readr

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

1.1.1 Reading data

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

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

# read the file in
some_example = read_csv(some_example)

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

head(some_example)

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

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

Other variants include:

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

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

79 skipped before reading data.

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

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

84 1.1.2 Writing data

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

89 These functions are not covered here.

90 1.1.3 Reading and writing lines

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

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

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

97 The model summary can be written to file. When writing lines to file, BE AWARE OF THE
98 DIFFERENCES BETWEEN UNIX AND WINDOWS line separators. Usually, this causes no
99 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")
```

100 This model output can be read back in for display, and each line of the model output is an
101 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")
```

```

102 ##
103 ## Call:
104 ## lm(formula = mpg ~ wt, data = mtcars)
105 ##
106 ## Residuals:
107 ##      Min       1Q   Median       3Q      Max
108 ## -4.5432 -2.3647 -0.1252  1.4096  6.8727
109 ##
110 ## Coefficients:
111 ##              Estimate Std. Error t value Pr(>|t|)
112 ## (Intercept)  37.2851     1.8776  19.858 < 2e-16 ***
113 ## wt          -5.3445     0.5591  -9.559 1.29e-10 ***
114 ## ---
115 ## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
116 ##
117 ## Residual standard error: 3.046 on 30 degrees of freedom
118 ## Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
119 ## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10

```

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

124 Excel files

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

129 1.2 String manipulation with `stringr`

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

133 `stringr` functions begin with `str_`.

134 1.2.1 Putting strings together

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

```

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

```

```

137 ## [1] "this string_this other string"
    # str_dup works like rep
    str_dup("this string", times = 3)
138 ## [1] "this stringthis stringthis string"
    # str_flatten works on lists and vectors
    str_flatten(string = as.list(letters), collapse = "_")
139 ## [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 = "-")
140 ## [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"
141 str_flatten is especially useful when displaying the type of an object that returns a list
142 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 = ", ")
143 ## [1] "tbl_df, tbl, data.frame"

```

1.2.2 Detecting strings

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

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

```

    # there should be 5 a-s here
    str_count(string = "ababababa", pattern = "a")
149 ## [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")
150 ## [1] 5 3
    # vectorise over the pattern to count both a-s and b-s
    str_count(string = "ababababa", pattern = c("a", "b"))
151 ## [1] 5 4
152 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"))

```

```

153 ## [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"))

154 ## [1] 5 1 4 3

155 str_locate locates the search pattern in a string, and returns the start and end as a two
156 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")

157 ##      start end
158 ## [1,]      2  4

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

159 ## [1] TRUE

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

160 ## [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")))

161 ## [1] TRUE

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

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

164 ## [1] FALSE FALSE  TRUE  TRUE

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

165 ## [1]  TRUE FALSE FALSE  TRUE

```

```

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

## [1] FALSE TRUE TRUE FALSE

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

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

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

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

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

## [1] 1 2

```

1.2.3 Matching strings

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

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

```

str_match(string = c("banana",
                     "bananageddon",
                     "bananas are bad"),
          pattern = "banana")

##      [,1]
## [1,] "banana"
## [2,] "banana"
## [3,] "banana"

```

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

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

187 The return type is a list, each element is a character matrix where the first column is
 188 the string subset matching the full search pattern, and then as many columns as there
 189 are parts to the search pattern. The parts of interest in the search pattern are indicated
 190 by wrapping them in parentheses. For example, in the case below, wrapping `[-.]` in
 191 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"
```

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

203 The return type is a list, with one element per input string. Each element is a character
 204 matrix, where each row is one possible match, and each column after the first (the full
 205 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]]
```

```

211 ##      [,1]      [,2] [,3]
212 ## [1,] "1970-somemonth" "1970" "somemonth"
213 ## [2,] "1990/anothermonth" "1990" "anothermonth"
214 ##
215 ## [[2]]
216 ##      [,1]      [,2] [,3]
217 ## [1,] "1990-somemonth" "1990" "somemonth"
218 ## [2,] "2001/anothermonth" "2001" "anothermonth"

```

219 1.2.4 Simpler pattern extraction

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

223 `str_extract` returns a character vector with the same length as the input string vector,
 224 while `str_extract_all` returns a list, with a character vector whose elements are the
 225 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]+)")

226 ## [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]+)")

227 ## [[1]]
228 ## [1] "1970-somemonth" "1990/anothermonth"
229 ##
230 ## [[2]]
231 ## [1] "1990-somemonth" "2001/anothermonth"

```

232 1.2.5 Breaking strings apart

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

236 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 = "[\\-|\\/]")

```

```

237 ## [[1]]
238 ## [1] "1970"      "somemonth" "01"
239 ##
240 ## [[2]]
241 ## [1] "1990"      "anothermonth" "01"

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

244 ## [[1]]
245 ## [1] "sim"      "param1"  "0.01"    "param2"  "0.05"    "param3"  "0.01.ext"

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

246 ## [[1]]
247 ## [1] ""
248 ## [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)")

249 ## [[1]]
250 ## [1] ""      ""      "0.01"  "0.05"  "0.01"  ""

251 str_split_fixed split the string into as many pieces as specified, and can be especially
252 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)

253 ##      [,1]      [,2]
254 ## [1,] "dir_level_1" "dir_level_2/file.ext"

```

255 1.2.6 Replacing string elements

256 str_replace is intended to replace the search pattern, and can be co-opted into the
 257 task of recovering simulation parameters or other data from regularly named files.
 258 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 = " ")
259 ## [1] " 0.01 0.05 0.01 "

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

262 Having replaced unwanted characters in the filename with spaces, str_trim offers a way
263 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

264 ## [1] "0.01 0.05 0.01"

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

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

267 1.2.7 Subsetting within strings

268 When strings are highly regular, useful data can be extracted from a string using str_sub.
269 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)

270 ## [1] "1970" "1990" "2010"

271 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)

272 ## [1] "01" "21" "31"

```

273 Finally, it's also possible to replace characters within a string based on the position. This
 274 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

275 ## [1] "1970-somemonth-01" "1990-anothermonth-01" "2010-thismonth-01"
```

276 1.2.8 Padding and truncating strings

277 Strings included in filenames or plots are often of unequal lengths, especially when they
 278 represent numbers. `str_pad` can pad strings with suitable characters to maintain equal
 279 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")

280 ## [1] "001" "010" "100"
```

281 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.")

282 ## [1] "bananas are great etc. etc."
```

283 1.2.9 Stringr aspects not covered here

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

- 285 • `str_wrap` (of dubious use),
- 286 • `str_interp`, `str_glue*` (better to use `glue`; see below),
- 287 • `str_sort`, `str_order` (used in sorting a character vector),
- 288 • `str_to_case*` (case conversion), and

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

292 1.3 String interpolation with glue

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

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

```
296 # print that each car name is a car model
297 cars = rownames(head(mtcars))
298 glue('The {cars} is a car model')
299 ## The Mazda RX4 is a car model
300 ## The Mazda RX4 Wag is a car model
301 ## The Datsun 710 is a car model
302 ## The Hornet 4 Drive is a car model
303 ## The Hornet Sportabout is a car model
304 ## The Valiant is a car model
```

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

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

```
305 # use dataframes for now
306 parameter_combinations = data.frame(param1 = letters[1:5],
307                                     param2 = 1:5)
308
309 # for command line arguments or to start multiple job scripts on the cluster
310 glue_data(parameter_combinations,
311           'simulation-name {param1} {param2}')
312
313 ## simulation-name a 1
314 ## simulation-name b 2
315 ## simulation-name c 3
316 ## simulation-name d 4
317 ## simulation-name e 5
318
319 # for filenames
320 glue_data(parameter_combinations,
321           'sim_data_param1_{param1}_param2_{param2}.ext')
322
323 ## sim_data_param1_a_param2_1.ext
324 ## sim_data_param1_b_param2_2.ext
325 ## sim_data_param1_c_param2_3.ext
```

```

313 ## sim_data_param1_d_param2_4.ext
314 ## sim_data_param1_e_param2_5.ext

```

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

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

320 1.4 Strings in ggplot

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

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

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

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

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

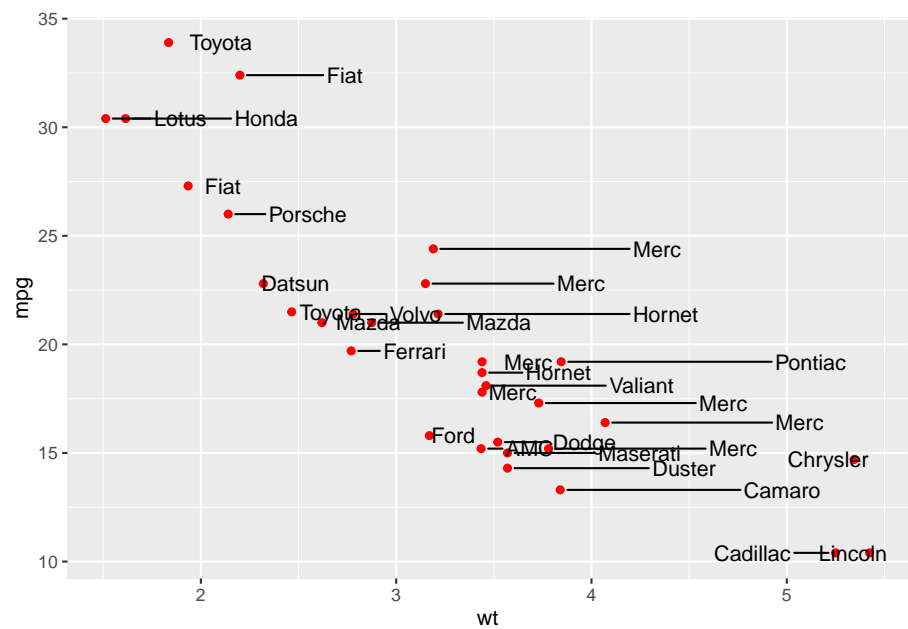
```

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)

```



333

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

Chapter 2

Working with lists and iteration

Every use case is ridiculous
until it happens to you.

```
# load the tidyverse
library(tidyverse)

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

## v tibble 3.0.1    v dplyr  0.8.5
## v tidyr  1.0.2    v forcats 0.5.0
## v purrr   0.3.4

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

350 2.1 Basic iteration with map

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

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

357 A good reason to use `purrr` functions instead of base R functions is their consistent and
358 clear naming, which always indicates how they should be used. This is explained in the
359 examples below.

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

362 2.1.1 map basic use

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

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

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
##
## [[7]]
## [1] 2.645751
##
```



```

387 ## [[8]]
388 ## [1] 2.828427
389 ##
390 ## [[9]]
391 ## [1] 3
392 ##
393 ## [[10]]
394 ## [1] 3.162278

```

2.1.2 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)

399 ## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
400 ## [9] 3.000000 3.162278

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

401 ## [1] "1.000000" "1.414214" "1.732051" "2.000000" "2.236068" "2.449490"
402 ## [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)

403 ## [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
407 ## # A tibble: 3 x 5
408 ## # Groups:   cyl [3]
409 ##   cyl data                n_rows mean_mpg first_car
410 ##   <dbl> <list>              <int>   <dbl> <chr>
411 ## 1     6 <tibble [7 x 11]>         7     19.7 Mazda RX4
412 ## 2     4 <tibble [11 x 11]>        11     26.7 Datsun 710
413 ## 3     8 <tibble [14 x 11]>        14     15.1 Hornet Sportabout
414 map accepts multiple functions that are applied in sequence to the input list-like object,
415 but this is confusing to the reader and ill advised.

```

416 2.1.3 map variants returning dataframes

417 `map_df` returns data frames, and by default binds dataframes by rows, while `map_dfr` does
 418 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)

```

```

419 ##   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
420 ## 1 22.8   4 108.0  93 3.85 2.320 18.61  1  1   4    1
421 ## 2 24.4   4 146.7  62 3.69 3.190 20.00  1  0   4    2
422 ## 3 21.0   6 160.0 110 3.90 2.620 16.46  0  1   4    4
423 ## 4 21.0   6 160.0 110 3.90 2.875 17.02  0  1   4    4
424 ## 5 18.7   8 360.0 175 3.15 3.440 17.02  0  0   3    2
425 ## 6 14.3   8 360.0 245 3.21 3.570 15.84  0  0   3    4

```

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

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

```

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

429 ##   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
430 ## 1 22.8   4 108.0  93 3.85 2.320 18.61  1  1   4    1
431 ## 2 24.4   4 146.7  62 3.69 3.190 20.00  1  0   4    2
432 ## 3 21.0   6 160.0 110 3.90 2.620 16.46  0  1   4    4

```

```

433 ## 4 21.0    6 160.0 110 3.90 2.875 17.02 0 1 4 4
434 ## 5 18.7    8 360.0 175 3.15 3.440 17.02 0 0 3 2
435 ## 6 14.3    8 360.0 245 3.21 3.570 15.84 0 0 3 4

```

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

```

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

438 ##   mpg cyl  disp hp drat   wt  qsec vs am gear carb mpg1 cyl1 disp1 hp1 drat1
439 ## 1 22.8   4 108.0 93 3.85 2.32 18.61 1 1   4   1 21   6 160 110 3.9
440 ## 2 24.4   4 146.7 62 3.69 3.19 20.00 1 0   4   2 21   6 160 110 3.9
441 ##   wt1 qsec1 vs1 am1 gear1 carb1 mpg2 cyl2 disp2 hp2 drat2 wt2 qsec2 vs2 am2
442 ## 1 2.620 16.46 0 1   4   4 18.7   8 360 175 3.15 3.44 17.02 0 0
443 ## 2 2.875 17.02 0 1   4   4 14.3   8 360 245 3.21 3.57 15.84 0 0
444 ##   gear2 carb2
445 ## 1     3     2
446 ## 2     3     4

```

447 2.1.4 Selective mapping

448 • `map_at` and `map_if`

449 2.2 More map variants

450 2.2.1 `map2`

451 `imap` here

452 2.2.2 `pmap`

453 2.2.3 `walk`

454 `walk2` and `pwalk`

455 2.3 Modification in place

456 `modify`

457 **2.4 Working with lists**

458 **2.4.1 Filtering lists**

459 **2.4.2 Summarising lists**

460 **2.4.3 Reduction and accumulation**

461 **2.4.4 Miscellaneous operation**