

# TRES Tidyverse Tutorial

Raphael, Pratik and Theo

2020-06-05



# Contents

|    |   |           |
|----|---|-----------|
| 2  | <b>Outline</b>  | <b>5</b>  |
| 3  | About . . . . .   | 5         |
| 4  | Schedule . . . . .  | 5         |
| 5  | Possible extras . . . . .   | 5         |
| 6  | Join . . . . .  | 6         |
| 7  | <b>1 Reading files and string manipulation</b>  | <b>7</b>  |
| 8  | 1.1 Data import and export with <code>readr</code> . . . . .                            | 7         |
| 9  | 1.2 String manipulation with <code>stringr</code> . . . . .                             | 10        |
| 10 | 1.3 String interpolation with <code>glue</code> . . . . .                               | 18        |
| 11 | 1.4 Strings in <code>ggplot</code> . . . . .  | 19        |
| 12 | <b>2 Reshaping data tables in the tidyverse, and other things</b>                       | <b>21</b> |
| 13 | 2.1 The new data frame: <code>tibble</code> . . . . .                                   | 22        |
| 14 | 2.2 The concept of tidy data . . . . .  | 24        |
| 15 | 2.3 Reshaping with <code>tidyr</code> . . . . .   | 26        |
| 16 | 2.4 Extra: factors and the <code>forcats</code> package . . . . .                       | 32        |
| 17 | 2.5 External resources . . . . .  | 37        |
| 18 | <b>3 Data manipulation with <code>dplyr</code></b>                                      | <b>39</b> |
| 19 | 3.1 Introduction . . . . .  | 39        |
| 20 | 3.2 Example data of the day . . . . .   | 39        |
| 21 | 3.3 Select variables with <code>select()</code> . . . . .                               | 41        |
| 22 | 3.4 Select observations with <code>filter()</code> . . . . .                            | 41        |
| 23 | 3.5 Create new variables with <code>mutate()</code> . . . . .                           | 41        |
| 24 | 3.6 Grouped results with <code>group_by()</code> and <code>summarise()</code> . . . . . | 42        |
| 25 | 3.7 Scoped variables . . . . .  | 42        |
| 26 | 3.8 More ! . . . . .  | 42        |
| 27 | <b>4 Working with lists and iteration</b>   | <b>43</b> |
| 28 | 4.1 Iteration with <code>map</code> . . . . .   | 43        |
| 29 | 4.2 More <code>map</code> variants . . . . .  | 47        |
| 30 | 4.3 Working with lists . . . . .  | 51        |



## Outline

This is the readable version of the TRES tidyverse tutorial. A convenient PDF version can be downloaded by clicking the PDF document icon in the header bar.

## About

The TRES tidyverse tutorial is an online workshop on how to use the tidyverse, a set of packages in the R computing language designed at making data handling and plotting easier.

This tutorial will take the form of a one hour per week video stream via Google Meet, every Friday morning at 10.00 (Groningen time) starting from the 29th of May, 2020 and lasting for a couple of weeks (depending on the number of topics we want to cover, but there should be at least 5).

**PhD students from outside our department are welcome to attend.**

## Schedule

| Topic                                | Package              | Instructor | Date*    |
|--------------------------------------|----------------------|------------|----------|
| Reading data and string manipulation | readr, stringr, glue | Pratik     | 29/05/20 |
| Data and reshaping                   | tibble, tidyr        | Raphael    | 05/06/20 |
| Manipulating data                    | dplyr                | Theo       | 12/06/20 |
| Working with lists and iteration     | purrr                | Pratik     | 19/06/20 |
| Plotting                             | ggplot2              | Raphael    | 26/06/20 |
| Regular expressions                  | regex                | Richel     | 03/07/20 |
| Programming with the tidyverse       | rlang                | Pratik     | 10/07/20 |

## Possible extras

- Reproducibility and package-making (with e.g. usethis)

- Embedding C++ code with Rcpp

## Join

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

\*Tentative dates.

## 51 Chapter 1

# 52 Reading files and string 53 manipulation



Every use case is ridiculous  
until it happens to you.

54  
55 Load the packages for the day.

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

### 56 1.1 Data import and export with readr

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

### 1.1.1 Reading data

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

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

# read the file in
some_example = read_csv(some_example)

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

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

Other variants include:

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

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

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

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

### 1.1.2 Writing data

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

These functions are not covered here.



### 83 1.1.3 Reading and writing lines

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

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

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

90 The model summary can be written to file. When writing lines to file, BE AWARE OF THE  
 91 DIFFERENCES BETWEEN UNIX AND WINDOWS line separators. Usually, this causes no  
 92 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")
```

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

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

# use cat to show the model output as it would be on screen
cat(model_output, sep = "\n")
#>
#> Call:
#> lm(formula = mpg ~ wt, data = mtcars)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -4.543 -2.365 -0.125  1.410  6.873
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   37.285      1.878   19.86 < 2e-16 ***
#> wt           -5.344      0.559   -9.56 1.3e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.05 on 30 degrees of freedom
#> Multiple R-squared:  0.753, Adjusted R-squared:  0.745
```

```
#> F-statistic: 91.4 on 1 and 30 DF, p-value: 1.29e-10
```

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

## 99 Excel files

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

## 104 1.2 String manipulation with `stringr`

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

108 `stringr` functions begin with `str_`.

### 109 1.2.1 Putting strings together

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

```
# str_c works like paste(), choose a separator
str_c("this string", "this other string", sep = "_")
#> [1] "this string_this other string"
```

```
# str_dup works like rep
str_dup("this string", times = 3)
#> [1] "this stringthis stringthis string"
```

```
# str_flatten works on lists and vectors
str_flatten(string = as.list(letters), collapse = "_")
#> [1] "a_b_c_d_e_f_g_h_i_j_k_l_m_n_o_p_q_r_s_t_u_v_w_x_y_z"
str_flatten(string = letters, collapse = "-")
#> [1] "a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
```

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

```
# get the class of a tibble and display it as a single string
class_tibble = class(tibble::tibble(a = 1))
str_flatten(string = class_tibble, collapse = ", ")
#> [1] "tbl_df, tbl, data.frame"
```

114 **1.2.2 Detecting strings**

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

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

```
# there should be 5 a-s here
str_count(string = "ababababa", pattern = "a")
#> [1] 5

# vectorise over the input string
# should return a vector of length 2, with integers 5 and 3
str_count(string = c("ababbababa", "banana"), pattern = "a")
#> [1] 5 3

# vectorise over the pattern to count both a-s and b-s
str_count(string = "ababababa", pattern = c("a", "b"))
#> [1] 5 4
```

119 Vectorising over both string and pattern works as expected.

```
# vectorise over both string and pattern
# counts a-s in first input, and b-s in the second
str_count(string = c("ababababa", "banana"),
          pattern = c("a", "b"))
#> [1] 5 1

# provide a longer pattern vector to search for both a-s
# and b-s in both inputs
str_count(string = c("ababababa", "banana"),
          pattern = c("a", "b",
                     "b", "a"))
#> [1] 5 1 4 3
```

120 `str_locate` locates the search pattern in a string, and returns the start and end as a two  
 121 column matrix.

```
# the behaviour of both str_locate and str_locate_all is
# to find the first match by default
str_locate(string = "banana", pattern = "ana")
#>      start end
#> [1,]     2   4

# str_detect detects a sequence in a string
str_detect(string = "Bananageddon is coming!",
           pattern = "na")
#> [1] TRUE
```

```
# str_detect is also vectorised and returns a two-element logical vector
str_detect(string = "Bananageddon is coming!",
            pattern = c("na", "don"))
#> [1] TRUE TRUE
```

```
# use any or all to convert a multi-element logical to a single logical
# here we ask if either of the patterns is detected
any(str_detect(string = "Bananageddon is coming!",
                pattern = c("na", "don")))
#> [1] TRUE
```

122 Detect whether a string starts or ends with a pattern. Also vectorised. Both have a negate  
123 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")
#> [1] FALSE FALSE TRUE TRUE
```

```
# str_ends looks at the last character
str_ends(fruit, "e")
#> [1] TRUE FALSE FALSE TRUE
```

```
# an example of negate = TRUE
str_ends(fruit, "e", negate = TRUE)
#> [1] FALSE TRUE TRUE FALSE
```

124 **str\_subset** [WHICH IS NOT RELATED TO **str\_sub**] helps with subsetting a character vec-  
125 tor based on a **str\_detect** predicate. In the example, all elements containing "banana"  
126 are subset.

127 **str\_which** has the same logic except that it returns the vector position and not the ele-  
128 ments.

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

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

```
#> [1] 1 2
```

### 1.2.3 Matching strings

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

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

```
str_match(string = c("banana",
                     "bananageddon",
                     "bananas are bad"),
          pattern = "banana")
#>      [,1]
#> [1,] "banana"
#> [2,] "banana"
#> [3,] "banana"
```

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

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

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

```
# first with [-.] treated simply as a separator
str_match(string = c("1970-somemonth-01",
                     "1990-anothermonth-01",
                     "2010-thismonth-01"),
          pattern = "(\\d{4})([-.])(\\w+)")
#>      [,1]      [,2]      [,3]
#> [1,] "1970-somemonth" "1970" "somemonth"
#> [2,] "1990-anothermonth" "1990" "anothermonth"
#> [3,] "2010-thismonth" "2010" "thismonth"

# then with [-.] actively searched for
str_match(string = c("1970-somemonth-01",
                     "1990-anothermonth-01",
                     "2010-thismonth-01"),
          pattern = "(\\d{4})([-.])(\\w+)")
#>      [,1]      [,2]      [,3]      [,4]
#> [1,] "1970-somemonth" "1970" "-" "somemonth"
```

```
#> [2,] "1990-anothermonth" "1990" "-" "anothermonth"
#> [3,] "2010-thismonth"      "2010" "-" "thismonth"
```

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

The return type is a list, with one element per input string. Each element is a character matrix, where each row is one possible match, and each column after the first (the full match) corresponds to the parts of the search pattern.

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

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

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

#> [[1]]
#>      [,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"
```

#### 1.2.4 Simpler pattern extraction

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

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

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

#> [1] "1970-somemonth" "1990-somemonth"
```

```
# extracting all full matches using str_extract_all
str_extract_all(string = c("1970-somemonth-01 or maybe 1990/anothermonth/01",
                           "1990-somemonth-01 or maybe 2001/anothermonth/01"),
                pattern = "(\\d{4})[\\-\\/](\\w+)" )

#> [[1]]
#> [1] "1970-somemonth"    "1990/anothermonth"
#>
#> [[2]]
#> [1] "1990-somemonth"    "2001/anothermonth"
```

### 1.2.5 Breaking strings apart

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

160 The return type is a list similar to `str_match`.

```
# split on either a hyphen or a forward slash
str_split(string = c("1970-somemonth-01",
                    "1990/anothermonth/01"),
          pattern = "[\\-\\/]" )

#> [[1]]
#> [1] "1970"      "somemonth" "01"
#>
#> [[2]]
#> [1] "1990"      "anothermonth" "01"
```

161 This can be useful in recovering simulation parameters from a filename, but may require  
 162 some knowledge of regex.

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

# not quite there
str_split(filename, pattern = "_")
#> [[1]]
#> [1] "sim"      "param1"   "0.01"     "param2"   "0.05"     "param3"   "0.01.ext"

# not really
str_split(filename,
          pattern = "sim_")
#> [[1]]
#> [1] ""
#> [2] "param1_0.01_param2_0.05_param3_0.01.ext"

# getting there but still needs work
```

```

str_split(filename,
           pattern = "(sim_)|_*param\\d{1}_|(.ext)")
#> [[1]]
#> [1] ""      ""      "0.01" "0.05" "0.01" ""

```

163 `str_split_fixed` split the string into as many pieces as specified, and can be especially  
 164 useful dealing with filepaths.

```

# split on either a hyphen or a forward slash
str_split_fixed(string = "dir_level_1/dir_level_2/file.ext",
                pattern = "/",
                n = 2)
#>      [,1]      [,2]
#> [1,] "dir_level_1" "dir_level_2/file.ext"

```

### 165 1.2.6 Replacing string elements

166 `str_replace` is intended to replace the search pattern, and can be co-opted into the  
 167 task of recovering simulation parameters or other data from regularly named files.  
 168 `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 = " ")
#> [1] " 0.01 0.05 0.01 "

```

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

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

```

# trim whitespaces from this filename after replacing unwanted text
filename = "sim_param1_0.01_param2_0.05_param3_0.01.ext"
filename_with_spaces = str_replace_all(filename,
                                       pattern = "(sim_)|_*param\\d{1}_|(.ext)",
                                       replacement = " ")
filename_without_spaces = str_trim(filename_with_spaces)
filename_without_spaces
#> [1] "0.01 0.05 0.01"

# the result can be split on whitespaces to return useful data
str_split(filename_without_spaces, " ")
#> [[1]]
#> [1] "0.01" "0.05" "0.01"

```



### 173 1.2.7 Subsetting within strings

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

175 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)
#> [1] "1970" "1990" "2010"
```

176 Similarly, it's possible to extract the last few characters using negative indices.

```
# get the day as characters -2 to -1
str_sub(string = c("1970-somemonth-01",
                  "1990-anothermonth-21",
                  "2010-thismonth-31"),
        start = -2, end = -1)
#> [1] "01" "21" "31"
```

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

178 requires using the assignment operator `<-`.

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

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

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

### 179 1.2.8 Padding and truncating strings

180 Strings included in filenames or plots are often of unequal lengths, especially when they

181 represent numbers. `str_pad` can pad strings with suitable characters to maintain equal

182 length filenames, with which it is easier to work.

```
# pad so all values have three digits
str_pad(string = c("1", "10", "100"),
        width = 3,
        side = "left",
        pad = "0")
#> [1] "001" "010" "100"
```

183 Strings can also be truncated if they are too long.

```

str_trunc(string = c("bananas are great and wonderful
                      and more stuff about bananas and
                      it really goes on about bananas"),
          width = 27,
          side = "right", ellipsis = "etc. etc.")
#> [1] "bananas are great etc. etc."

```

### 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')
#> The Mazda RX4 is a car model
#> The Mazda RX4 Wag is a car model
#> The Datsun 710 is a car model
#> The Hornet 4 Drive is a car model
#> The Hornet Sportabout is a car model
#> The Valiant is a car model

```

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

The related glue\_data is even more useful in printing from a dataframe. In this example, it can quickly generate command line arguments or filenames.

```

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

```

```

# for command line arguments or to start multiple job scripts on the cluster
glue_data(parameter_combinations,
           'simulation-name {param1} {param2}')
#> simulation-name a 1
#> simulation-name b 2
#> simulation-name c 3
#> simulation-name d 4
#> simulation-name e 5

# for filenames
glue_data(parameter_combinations,
           'sim_data_param1_{param1}_param2_{param2}.ext')
#> sim_data_param1_a_param2_1.ext
#> sim_data_param1_b_param2_2.ext
#> sim_data_param1_c_param2_3.ext
#> sim_data_param1_d_param2_4.ext
#> sim_data_param1_e_param2_5.ext

```

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

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

## 205 1.4 Strings in ggplot

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

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

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

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

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

```

library(ggplot2)
library(ggrepel)

# prepare car labels using word function

```

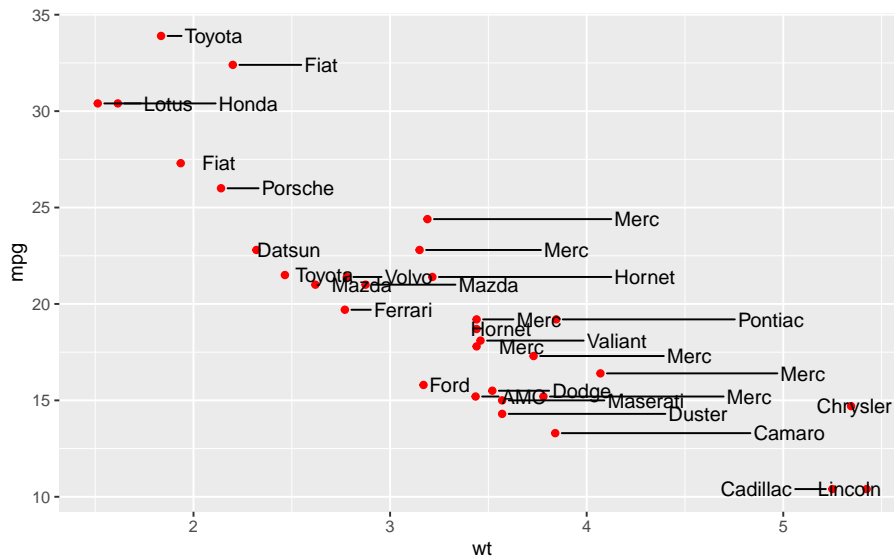
```

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)

```

218



219

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

## Chapter 2

# Reshaping data tables in the tidyverse, and other things

Raphael Scherrer

Every use case is ridiculous  
until it happens to you.

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

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

## 2.1 The new data frame: tibble

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

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

# Or an equivalent tibble
tibble(who = c("Pratik", "Theo", "Raph"), chapt = c("1, 4", "3", "2, 5"))
#> # A tibble: 3 x 2
#>   who   chapt
#>   <chr> <chr>
#> 1 Pratik 1, 4
#> 2 Theo   3
#> 3 Raph  2, 5
```

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

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

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

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

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

```
# Perform a PCA on mtcars
pca_scores <- prcomp(mtcars)$x
head(pca_scores) # looks like a data frame or a tibble...
#>           PC1  PC2  PC3  PC4  PC5  PC6  PC7  PC8
#> Mazda RX4      -79.60  2.13 -2.15 -2.707 -0.702 -0.3149 -0.09870 -0.0779
#> Mazda RX4 Wag  -79.60  2.15 -2.22 -2.178 -0.884 -0.4534 -0.00355 -0.0957
#> Datsun 710     -133.89 -5.06 -2.14  0.346  1.106  1.1730  0.00576  0.1362
#> Hornet 4 Drive   8.52 44.99  1.23  0.827  0.424 -0.0579 -0.02431  0.2212
#> Hornet Sportabout 128.69 30.82  3.34 -0.521  0.737 -0.3329  0.10630 -0.0530
#> Valiant        -23.22 35.11 -3.26  1.401  0.803 -0.0884  0.23895  0.4239
#>           PC9  PC10  PC11
#> Mazda RX4      -0.200 -0.2901  0.106
#> Mazda RX4 Wag  -0.353 -0.1928  0.107
#> Datsun 710     -0.198  0.0763  0.267
#> Hornet 4 Drive   0.356 -0.0906  0.209
#> Hornet Sportabout 0.153 -0.1886 -0.109
#> Valiant         0.101 -0.0377  0.276
class(pca_scores) # but is actually a matrix
#> [1] "matrix"

# Convert to tibble
as_tibble(pca_scores)
#> # A tibble: 32 x 11
#>       PC1  PC2  PC3  PC4  PC5  PC6  PC7  PC8  PC9  PC10
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  -79.6   2.13 -2.15 -2.71 -0.702 -0.315 -0.0987 -0.0779 -0.200 -0.290
#> 2  -79.6   2.15 -2.22 -2.18 -0.884 -0.453 -0.00355 -0.0957 -0.353 -0.193
#> 3 -134.   -5.06 -2.14  0.346  1.11  1.17  0.00576  0.136 -0.198  0.0763
#> 4   8.52 45.0  1.23  0.827  0.424 -0.0579 -0.0243  0.221  0.356 -0.0906
#> 5 129.   30.8  3.34 -0.521  0.737 -0.333  0.106 -0.0530  0.153 -0.189
#> 6  -23.2 35.1  -3.26  1.40  0.803 -0.0884  0.239  0.424  0.101 -0.0377
#> # ... with 26 more rows, and 1 more variable: PC11 <dbl>
```

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

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

## 2.2 The concept of tidy data

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

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

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

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

# Snapshot
data
#> # A tibble: 100 x 5
#>   id body_weight beak_length beak_width island
#>   <int>      <dbl>      <dbl>      <dbl> <chr>
#> 1     1      10.8        4.94      1.94 Isabela
#> 2     2      15.4        5.02      2.00 Isabela
#> 3     3      15.0        4.92      1.91 Isabela
#> 4     4       8.51        5.16      2.02 Isabela
#> 5     5      14.9        5.03      1.93 Isabela
#> 6     6       8.41        4.92      2.18 Isabela
#> # ... with 94 more rows
```

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

The tidy equivalent of this dataset would be:

```
data <- pivot_longer(
```



```

data,
  cols = c("body_weight", "beak_length", "beak_width"),
  names_to = "variable"
)
data
#> # A tibble: 300 x 4
#>   id island variable    value
#>   <int> <chr>   <chr>    <dbl>
#> 1     1  Isabela body_weight 10.8
#> 2     1  Isabela beak_length 4.94
#> 3     1  Isabela beak_width  1.94
#> 4     2  Isabela body_weight 15.4
#> 5     2  Isabela beak_length 5.02
#> 6     2  Isabela beak_width  2.00
#> # ... with 294 more rows

```

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

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

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

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

```

# Make a contingency table
ctg <- with(data, table(island, variable))
ctg
#>           variable
#> island  beak_length beak_width body_weight
#> Isabela           50          50          50
#> Santa Cruz         50          50          50

```

A variety of statistical tests can be used on contingency tables such as Fisher's 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)
#> # A tibble: 6 x 3
#>   island      variable      n
#>   <chr>      <chr>      <int>
#> 1 Isabela  beak_length    50
#> 2 Santa Cruz beak_length    50
#> 3 Isabela  beak_width     50
#> 4 Santa Cruz beak_width     50
#> 5 Isabela  body_weight    50
#> 6 Santa Cruz body_weight    50
```

## 2.3 Reshaping with `tidyr`

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

### 2.3.1 Pivoting

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

```
pivot_wider(
  data,
  names_from = "variable",
  values_from = "value",
  id_cols = c("id", "island")
)
#> # A tibble: 100 x 5
#>   id island body_weight beak_length beak_width
#>   <int> <chr>      <dbl>      <dbl>      <dbl>
#> 1     1 Isabela    10.8        4.94        1.94
#> 2     2 Isabela    15.4        5.02        2.00
#> 3     3 Isabela    15.0        4.92        1.91
```

```
#> 4      4 Isabela      8.51      5.16      2.02
#> 5      5 Isabela     14.9      5.03      1.93
#> 6      6 Isabela      8.41      4.92      2.18
#> # ... with 94 more rows
```

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

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

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

### 312 2.3.2 Handling missing values

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

```
# We replace 100 random observations by NAs
ii <- sample(nrow(data), 100)
data$value[ii] <- NA
data
#> # A tibble: 300 x 4
#>   id island variable  value
#>   <int> <chr>  <chr>    <dbl>
#> 1     1 Isabela body_weight 10.8
#> 2     1 Isabela beak_length NA
#> 3     1 Isabela beak_width NA
#> 4     2 Isabela body_weight NA
#> 5     2 Isabela beak_length 5.02
#> 6     2 Isabela beak_width NA
#> # ... with 294 more rows
```

314 We could get rid of the rows that have missing values using `drop_na`:

```
drop_na(data, value)
#> # A tibble: 200 x 4
#>   id island variable  value
#>   <int> <chr>  <chr>    <dbl>
#> 1     1 Isabela body_weight 10.8
#> 2     2 Isabela beak_length 5.02
#> 3     3 Isabela body_weight 15.0
#> 4     3 Isabela beak_length 4.92
#> 5     4 Isabela body_weight 8.51
#> 6     4 Isabela beak_width 2.02
#> # ... with 194 more rows
```

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

```

replace_na(data, replace = list(value = -999))
#> # A tibble: 300 x 4
#>   id island variable    value
#>   <int> <chr>   <chr>      <dbl>
#> 1     1  Isabela body_weight  10.8
#> 2     1  Isabela beak_length -999
#> 3     1  Isabela beak_width  -999
#> 4     2  Isabela body_weight -999
#> 5     2  Isabela beak_length   5.02
#> 6     2  Isabela beak_width  -999
#> # ... with 294 more rows

```

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

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

```

fill(data, value)
#> # A tibble: 300 x 4
#>   id island variable    value
#>   <int> <chr>   <chr>      <dbl>
#> 1     1  Isabela body_weight  10.8
#> 2     1  Isabela beak_length  10.8
#> 3     1  Isabela beak_width   10.8
#> 4     2  Isabela body_weight  10.8
#> 5     2  Isabela beak_length   5.02
#> 6     2  Isabela beak_width   5.02
#> # ... with 294 more rows

```

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

### 2.3.3 Splitting and combining cells

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

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

```

# Sample random dates for each observation
data$day <- sample(30, nrow(data), replace = TRUE)
data$month <- sample(12, nrow(data), replace = TRUE)
data$year <- sample(2019:2020, nrow(data), replace = TRUE)
data
#> # A tibble: 300 x 7

```

```
#>      id island variable    value  day month  year
#>   <int> <chr>  <chr>      <dbl> <int> <int> <int>
#> 1     1  Isabela body_weight 10.8    8     7  2020
#> 2     1  Isabela beak_length NA     19    7  2019
#> 3     1  Isabela beak_width  NA     17   12  2019
#> 4     2  Isabela body_weight NA     20   12  2020
#> 5     2  Isabela beak_length 5.02   21   10  2020
#> 6     2  Isabela beak_width  NA     23    2  2020
#> # ... with 294 more rows
```

329 We could combine the day, month and year columns into a single date column, with a  
 330 dash as a separator, using `unite`:

```
data <- unite(data, day, month, year, col = "date", sep = "-")
data
#> # A tibble: 300 x 5
#>      id island variable    value date
#>   <int> <chr>  <chr>      <dbl> <chr>
#> 1     1  Isabela body_weight 10.8 8-7-2020
#> 2     1  Isabela beak_length NA    19-7-2019
#> 3     1  Isabela beak_width  NA    17-12-2019
#> 4     2  Isabela body_weight NA    20-12-2020
#> 5     2  Isabela beak_length 5.02 21-10-2020
#> 6     2  Isabela beak_width  NA    23-2-2020
#> # ... with 294 more rows
```

331 Of course, we can revert back to the previous dataset by splitting the date column with  
 332 `separate`.

```
separate(data, date, into = c("day", "month", "year"))
#> # A tibble: 300 x 7
#>      id island variable    value day  month year
#>   <int> <chr>  <chr>      <dbl> <chr> <chr> <chr>
#> 1     1  Isabela body_weight 10.8    8     7  2020
#> 2     1  Isabela beak_length NA     19    7  2019
#> 3     1  Isabela beak_width  NA     17   12  2019
#> 4     2  Isabela body_weight NA     20   12  2020
#> 5     2  Isabela beak_length 5.02   21   10  2020
#> 6     2  Isabela beak_width  NA     23    2  2020
#> # ... with 294 more rows
```

333 But note that the day, month and year columns are now of class character and not in-  
 334 teger anymore. This is because they result from the splitting of date, which itself was a  
 335 character column.

336 You can also separate a single column into multiple rows using `separate_rows`:

```
separate_rows(data, date)
#> # A tibble: 900 x 5
```

```
#>      id island variable    value date
#>   <int> <chr>   <chr>      <dbl> <chr>
#> 1     1  Isabela body_weight  10.8  8
#> 2     1  Isabela body_weight  10.8  7
#> 3     1  Isabela body_weight  10.8 2020
#> 4     1  Isabela beak_length   NA   19
#> 5     1  Isabela beak_length   NA    7
#> 6     1  Isabela beak_length   NA  2019
#> # ... with 894 more rows
```

### 2.3.4 Expanding tables using combinations

Instead of getting rid of rows with NAs, we may want to add rows with NAs, for example, for combinations of parameters that we did not measure.

```
data <- separate(data, date, into = c("day", "month", "year"))
to_rm <- with(data, island == "Santa Cruz" & year == "2020")
data <- data[!to_rm,]
tail(data)
#> # A tibble: 6 x 7
#>      id island variable    value day month year
#>   <int> <chr>   <chr>      <dbl> <chr> <chr> <chr>
#> 1    98 Santa Cruz beak_length  4.94  22   12   2019
#> 2    98 Santa Cruz beak_width   1.90   9    1   2019
#> 3    99 Santa Cruz body_weight  15.0  16    7   2019
#> 4    99 Santa Cruz beak_length   NA   26   10   2019
#> 5    99 Santa Cruz beak_width   2.04  30    7   2019
#> 6   100 Santa Cruz beak_width   NA   23    3   2019
```

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

```
expand_grid(
  island = c("Isabela", "Santa Cruz"),
  year = c("2019", "2020")
)
#> # A tibble: 4 x 2
#>   island year
#>   <chr>   <chr>
#> 1 Isabela 2019
#> 2 Isabela 2020
#> 3 Santa Cruz 2019
#> 4 Santa Cruz 2020
```

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

```
expand(data, island, year)
```

```
#> # A tibble: 4 x 2
#>   island   year
#>   <chr>   <chr>
#> 1 Isabela 2019
#> 2 Isabela 2020
#> 3 Santa Cruz 2019
#> 4 Santa Cruz 2020
```

344 As you can see, we get all the combinations of the variables of interest, even those that are  
 345 missing. But sometimes you might be interested in variables that are *nested* within each  
 346 other and not *crossed*. For example, say we have measured birds at different locations  
 347 within each island:

```
nrow_Isabela <- with(data, length(which(island == "Isabela")))
nrow_SantaCruz <- with(data, length(which(island == "Santa Cruz")))
sites_Isabela <- sample(c("A", "B"), size = nrow_Isabela, replace = TRUE)
sites_SantaCruz <- sample(c("C", "D"), size = nrow_SantaCruz, replace = TRUE)
sites <- c(sites_Isabela, sites_SantaCruz)
data$site <- sites
data
#> # A tibble: 232 x 8
#>   id island variable    value day month year site
#>   <int> <chr>   <chr>    <dbl> <chr> <chr> <chr> <chr>
#> 1     1 Isabela body_weight 10.8   8     7    2020 A
#> 2     1 Isabela beak_length NA     19    7    2019 B
#> 3     1 Isabela beak_width NA     17   12    2019 B
#> 4     2 Isabela body_weight NA     20   12    2020 A
#> 5     2 Isabela beak_length 5.02  21   10    2020 A
#> 6     2 Isabela beak_width NA     23    2    2020 A
#> # ... with 226 more rows
```

348 Of course, if sites A and B are on Isabela, they cannot be on Santa Cruz, where we have sites  
 349 C and D instead. It would not make sense to expand assuming that island and site are  
 350 crossed, instead, they are nested. We can therefore expand using the `nesting` function:

```
expand(data, nesting(island, site, year))
#> # A tibble: 6 x 3
#>   island   site year
#>   <chr>   <chr> <chr>
#> 1 Isabela A    2019
#> 2 Isabela A    2020
#> 3 Isabela B    2019
#> 4 Isabela B    2020
#> 5 Santa Cruz C    2019
#> 6 Santa Cruz D    2019
```

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

353 crossing, the complement of nesting:

```

expand(data, crossing(nesting(island, site), year)) # both can be used together
#> # A tibble: 8 x 3
#>   island    site year
#>   <chr>    <chr> <chr>
#> 1 Isabela  A      2019
#> 2 Isabela  A      2020
#> 3 Isabela  B      2019
#> 4 Isabela  B      2020
#> 5 Santa Cruz C      2019
#> 6 Santa Cruz C      2020
#> # ... with 2 more rows

```

354 Here, we specify that `site` is nested within `island` and these two are crossed with `site`.  
 355 Easy!

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

```

tail(complete(data, crossing(nesting(island, site), year)))
#> # A tibble: 6 x 8
#>   island    site year   id variable    value day  month
#>   <chr>    <chr> <chr> <int> <chr>    <dbl> <chr> <chr>
#> 1 Santa Cruz D    2019    95 beak_width  NA    13    10
#> 2 Santa Cruz D    2019    98 beak_length 4.94  22    12
#> 3 Santa Cruz D    2019    99 body_weight 15.0  16     7
#> 4 Santa Cruz D    2019    99 beak_length  NA    26    10
#> 5 Santa Cruz D    2019    99 beak_width  2.04  30     7
#> 6 Santa Cruz D    2020    NA <NA>      NA    <NA> <NA>
# the last row has been added, full of NAs

```

359 which nicely keeps the rest of the columns in the tibble and just adds the missing combi-  
 360 nations.

### 361 2.3.5 Nesting

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

## 365 2.4 Extra: factors and the `forcats` package

```
library(forcats)
```

366 Categorical variables can be stored in R as character strings in character or factor ob-  
 367 jects. A factor looks like a character, but it actually is an integer vector, where each



integer is mapped to a character label. With this respect it is sort of an enhanced version of character. For example,

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

is a character vector, recognizable to its double quotes, while

```
my_fact_vec <- factor(my_char_vec) # as.factor would work too
my_fact_vec
#> [1] Pratik Theo  Raph
#> Levels: Pratik Raph Theo
```

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

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

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 Change the order of the levels

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

```
# Make the plotting code a function so we can re-use it without copying and pasting
my_plot <- function(data) {
```

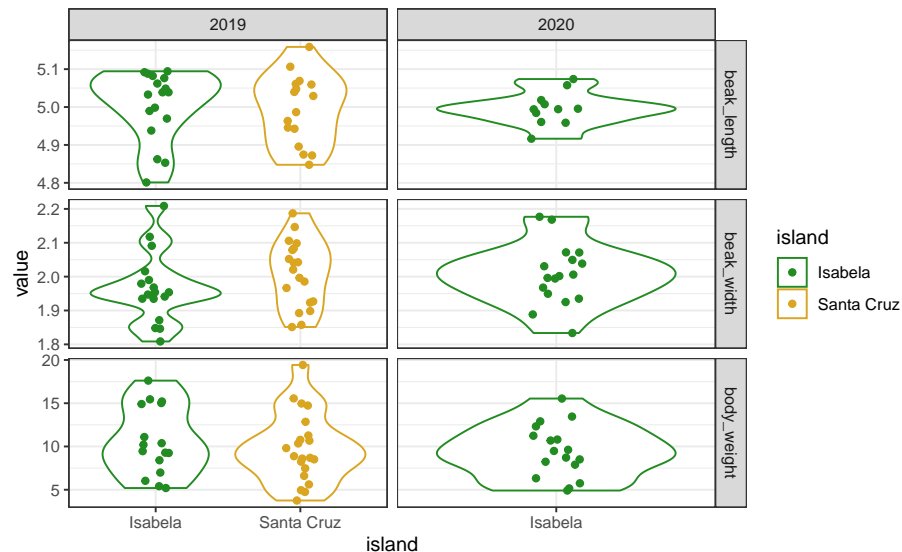
```
# We do not cover the ggplot functions in this chapter, this is just to
# illustrate our use-case, wait until chapter 5!
```

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

```
my_plot(data)
```

```
# Remember that data are missing from Santa Cruz in 2020
```

394

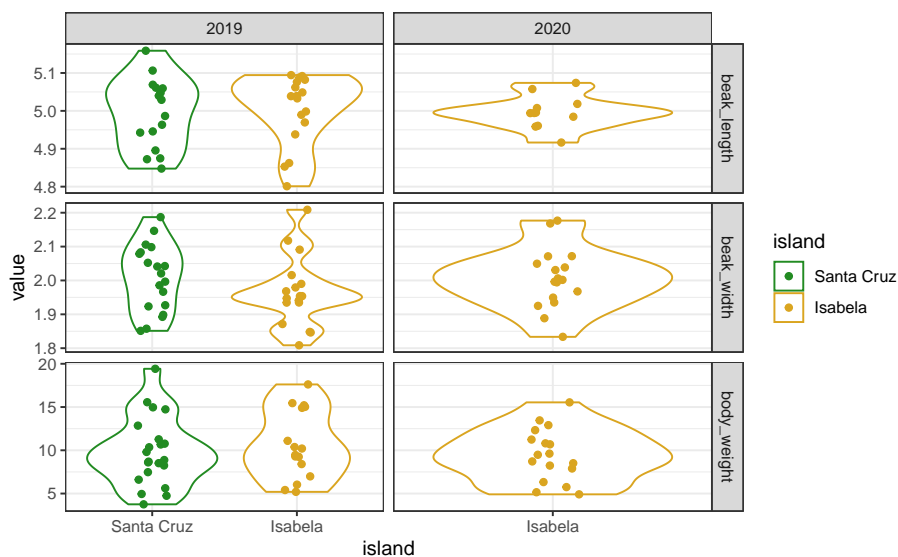


395

396 Here, the islands (horizontal axis) and the variables (the facets) are displayed in alphabet-  
 397 ical order. When making a figure you may want to customize these orders in such a way  
 398 that your message is optimally conveyed by your figure, and this may involve playing with  
 399 the order of levels.

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

```
data$island <- as.factor(data$island) # turn this column into a factor
data$island <- fct_relevel(data$island, c("Santa Cruz", "Isabela"))
my_plot(data) # order of islands has changed!
```



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

```
data$island[1:10]
#> [1] Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela
#> [10] Isabela
#> Levels: Santa Cruz Isabela
fct_relevel(data$island, c("Isabela", "Santa Cruz"))[1:10] # same thing, different levels
#> [1] Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela Isabela
#> [10] Isabela
#> Levels: Isabela Santa Cruz
```

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

```
data$variable <- as.factor(data$variable)
levels(data$variable)
#> [1] "beak_length" "beak_width" "body_weight"
levels(fct_inorder(data$variable))
#> [1] "body_weight" "beak_length" "beak_width"
```

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

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

Other variants exist to do more complex reordering, all present in the `forcats` cheatsheet,

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

### 2.4.2 Change the levels themselves

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

```
fct_recode(
  my_fact_vec,
  "Pratik Gupte" = "Pratik",
  "Theo Pannetier" = "Theo",
  "Raphael Scherrer" = "Raph"
)
#> [1] Pratik Gupte    Theo Pannetier    Raphael Scherrer
#> Levels: Pratik Gupte Raphael Scherrer Theo Pannetier
```

or collapse factor levels together using `fct_collapse`:

```
fct_collapse(my_fact_vec, EU = c("Theo", "Raph"), NonEU = "Pratik")
#> [1] NonEU EU      EU
#> Levels: NonEU EU
```

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

### 2.4.3 Dropping levels

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

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

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

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

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

436 Note that this is equivalent to doing:

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

#### 437 2.4.4 Other things

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

#### 440 2.4.5 Take home message for forcats

441 Use this package to manipulate your factors. Do you need factors? Or are character vec-  
442 tors enough? That is your call, and may depend on the kind of analyses you want to do  
443 and what they require. We saw here that for plotting, having factors can allow you to do  
444 quite some tweaking of the display. If you encounter a situation where the order of encod-  
445 ing of your character vector starts to matter, then maybe converting into a factor would  
446 make your life easier. And if you do so, remember that lots of tools to perform all kinds of  
447 manipulation are available to you with both `stringr` and `forcats`.

### 448 2.5 External resources

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

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



## Chapter 3

# Data manipulation with dplyr

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

### 3.1 Introduction

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

Why dplyr ? dplyr vs base R

### 3.2 Example data of the day

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

```
phylacine <- readr::read_csv("data/phylacine_traits.csv")
phylacine
#> # A tibble: 5,831 x 24
#>   Binomial.1.2 Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>          <chr>      <chr>      <chr>      <chr>          <dbl> <dbl>
#> 1 Abditomys_l~ Rodentia Muridae  Abditomys latidens          1      0
#> 2 Abeomelomys~ Rodentia Muridae  Abeomelo~ sevia            1      0
#> 3 Abrawayaomy~ Rodentia Cricetidae Abrawaya~ ruschii          1      0
#> 4 Abrocoma_be~ Rodentia Abrocomid~ Abrocoma bennettii          1      0
#> 5 Abrocoma_bo~ Rodentia Abrocomid~ Abrocoma boliviensis          1      0
#> 6 Abrocoma_bu~ Rodentia Abrocomid~ Abrocoma budini            1      0
#> # ... with 5,825 more rows, and 17 more variables: Freshwater <dbl>,
#> #   Aerial <dbl>, Life.Habit.Method <chr>, Life.Habit.Source <chr>,
#> #   Mass.g <dbl>, Mass.Method <chr>, Mass.Source <chr>, Mass.Comparison <chr>,
```

```
#> #   Mass.Comparison.Source <chr>, Island.Endemicity <chr>,
#> #   IUCN.Status.1.2 <chr>, Added.IUCN.Status.1.2 <chr>, Diet.Plant <dbl>,
#> #   Diet.Vertebrate <dbl>, Diet.Invertebrate <dbl>, Diet.Method <chr>,
#> #   Diet.Source <chr>
```

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

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

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

```
phylacine <- phylacine %>%
  dplyr::rename(
    "binomial" = Binomial.1.2,
    "order" = Order.1.2,
    "family" = Family.1.2,
    "genus" = Genus.1.2,
    "species" = Species.1.2,
    "terrestrial" = Terrestrial,
    "marine" = Marine,
    "freshwater" = Freshwater,
    "aerial" = Aerial,
    "life_habit_method" = Life.Habit.Method,
    "life_habit_source" = Life.Habit.Source,
    "mass_g" = Mass.g,
    "mass_method" = Mass.Method,
    "mass_source" = Mass.Source,
    "mass_comparison" = Mass.Comparison,
    "mass_comparison_source" = Mass.Comparison.Source,
    "island_endemicity" = Island.Endemicity,
    "iucn_status" = IUCN.Status.1.2, # not even for acronyms
    "added_iucn_status" = Added.IUCN.Status.1.2,
    "diet_plant" = Diet.Plant,
    "diet_vertibrate" = Diet.Vertebrate,
    "diet_invertebrate" = Diet.Invertebrate,
    "diet_method" = Diet.Method,
    "diet_source" = Diet.Source
  )
```



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

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

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

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

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

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

### 483 3.3 Select variables with select( )

### 484 3.4 Select observations with filter( )

### 485 3.5 Create new variables with mutate( )

486 can also edit existing ones

487 drop existing variables with transmute( )

### 488 3.6 Grouped results with `group_by()` and `summarise()`

### 489 3.7 Scoped variables

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

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

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

### 490 3.8 More !

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

## 493 Chapter 4

# 494 Working with lists and iteration



Every use case is ridiculous  
until it happens to you.

495  
`# load the tidyverse`  
`library(tidyverse)`

## 496 4.1 Iteration with map

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

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

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

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

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

#### 508 4.1.1 Basic use of `map`

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

```
# get the square root of each integer 1 - 10
some_numbers = 1:10
map(some_numbers, sqrt)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 1.41
#>
#> [[3]]
#> [1] 1.73
#>
#> [[4]]
#> [1] 2
#>
#> [[5]]
#> [1] 2.24
#>
#> [[6]]
#> [1] 2.45
#>
#> [[7]]
#> [1] 2.65
#>
#> [[8]]
#> [1] 2.83
#>
#> [[9]]
#> [1] 3
#>
#> [[10]]
#> [1] 3.16
```

### 512 4.1.2 map variants returning vectors

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

```
# use map_dbl to get a vector of square roots
some_numbers = 1:10
map_dbl(some_numbers, sqrt)
#> [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16

# map_chr will convert the output to a character
map_chr(some_numbers, sqrt)
#> [1] "1.000000" "1.414214" "1.732051" "2.000000" "2.236068" "2.449490"
#> [7] "2.645751" "2.828427" "3.000000" "3.162278"

# map_int will NOT round the output to an integer

# map_lgl returns TRUE/FALSE values
some_numbers = c(NA, 1:3, NA, NaN, Inf, -Inf)
map_lgl(some_numbers, is.na)
#> [1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
```

### 516 4.1.3 Integrating map and tidyr::nest

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

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

# get the number of rows per dataframe
# the mean mileage
# and the first car
some_data = some_data %>%
  mutate(n_rows = map_int(data, nrow),
         mean_mpg = map_dbl(data, ~mean(.$mpg)),
         first_car = map_chr(data, ~first(.$car_name)))

some_data
#> # A tibble: 3 x 5
#> # Groups:   cyl [3]
#>   cyl data                                n_rows mean_mpg first_car
#>   <dbl> <list>                                <int>   <dbl> <chr>
#> 1     6 <tibble [7 x 11]>                        7      19.7 Mazda RX4
```

```
#> 2      4 <tibble [11 x 11]>      11      26.7 Datsun 710
#> 3      8 <tibble [14 x 11]>      14      15.1 Hornet Sportabout
```

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

#### 521 4.1.4 `map` variants returning dataframes

522 `map_df` returns data frames, and by default binds dataframes by rows, while `map_dfr` does  
 523 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)
#>   mpg cyl disp  hp drat   wt  qsec vs am gear carb
#> 1 22.8   4  108  93 3.85 2.32 18.6  1  1   4    1
#> 2 24.4   4  147  62 3.69 3.19 20.0  1  0   4    2
#> 3 21.0   6  160 110 3.90 2.62 16.5  0  1   4    4
#> 4 21.0   6  160 110 3.90 2.88 17.0  0  1   4    4
#> 5 18.7   8  360 175 3.15 3.44 17.0  0  0   3    2
#> 6 14.3   8  360 245 3.21 3.57 15.8  0  0   3    4
```

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

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

```
# the same as above but with a pipe
some_list %>%
  map_dfr(head, n = 2)
#>   mpg cyl disp  hp drat   wt  qsec vs am gear carb
#> 1 22.8   4  108  93 3.85 2.32 18.6  1  1   4    1
#> 2 24.4   4  147  62 3.69 3.19 20.0  1  0   4    2
#> 3 21.0   6  160 110 3.90 2.62 16.5  0  1   4    4
#> 4 21.0   6  160 110 3.90 2.88 17.0  0  1   4    4
#> 5 18.7   8  360 175 3.15 3.44 17.0  0  0   3    2
#> 6 14.3   8  360 245 3.21 3.57 15.8  0  0   3    4
```

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

```
some_list %>%
  map_dfc(head, n = 2)
#>   mpg cyl disp  hp drat   wt  qsec vs am gear carb mpg1 cyl1 disp1 hp1 drat1
#> 1 22.8   4  108  93 3.85 2.32 18.6  1  1   4    1  21    6  160 110   3.9
#> 2 24.4   4  147  62 3.69 3.19 20.0  1  0   4    2  21    6  160 110   3.9
#>   wt1 qsec1 vs1 am1 gear1 carb1 mpg2 cyl2 disp2 hp2 drat2  wt2 qsec2 vs2 am2
```

```
#> 1 2.62 16.5 0 1 4 4 18.7 8 360 175 3.15 3.44 17.0 0 0
#> 2 2.88 17.0 0 1 4 4 14.3 8 360 245 3.21 3.57 15.8 0 0
#> gear2 carb2
#> 1 3 2
#> 2 3 4
```

### 529 4.1.5 Selective mapping

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

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

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

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

# check the data structure
data
#> # A tibble: 3 x 3
#>   cyl data          model
#>   <dbl> <list>         <list>
#> 1     6 <tibble [7 x 10]> <tibble [7 x 10]>
#> 2     4 <tibble [11 x 10]> <lm>
#> 3     8 <tibble [14 x 10]> <lm>
```

533 `map_at` works on specific elements of a list or vector. Come back to this, it's not particu-  
534 larly useful.

## 535 4.2 More map variants

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

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

```
#> [1] 7
#>
#> [[2]]
#> [1] 9
#>
#> [[3]]
#> [1] 11
#>
#> [[4]]
#> [1] 13
#>
#> [[5]]
#> [1] 15
```

#### 539 4.2.1 Mapping over two inputs with map2

540 map2 has the same variants as map, allowing for different return types. Here map2\_int  
541 returns an integer vector.

```
# consider 2 vectors and replicate the simple vector addition using map2
map2_int(.x = 1:5,
        .y = 6:10,
        .f = sum)
#> [1] 7 9 11 13 15
```

542 map2 doesn't have \_at and \_if variants.

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

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

# a not particularly useful example
map2(this_list, names(this_list),
     function(x, y) {
       glue::glue('{x} : {y}')
     })
#> $a
#> first letter : a
#>
#> $b
#> second letter : b

# imap can also do this
imap(this_list,
```



```

    function(x, .y){
      glue::glue('{x} : {.y}')
    })
#> $a
#> first letter : a
#>
#> $b
#> second letter : b

```

#### 546 4.2.2 Mapping over multiple inputs with pmap

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

```

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

# print a few statements
pmap_chr(list(list_01, list_02, list_03),
  function(l1, l2, l3){
    glue::glue('number {l1}, letter {l2}, colour {l3}')
  })
#> [1] "number 1, letter a, colour #FF0000FF"
#> [2] "number 2, letter b, colour #00FF00FF"
#> [3] "number 3, letter c, colour #0000FFFF"

```

#### 549 4.2.3 Mapping at depth

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

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

```

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

# map over the second level to count the number of
# cars with N gears in the set of cars with M cylinders
# display only for cyl = 4
map_depth(this_list[1], 2, nrow)
#> $`4`
#> $`4`$`3`
#> [1] 1

```

```
#>
#> `$`4`$`4`
#> [1] 8
#>
#> `$`4`$`5`
#> [1] 2
```

#### 554 4.2.4 Iteration without a return

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

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

```
this_list = split(mtcars, mtcars$cyl)

iwalk(this_list,
      function(df, .y){
        message(glue::glue('{nrow(df)} cars with {.y} cylinders'))
      })
```

#### 561 4.2.5 Modify rather than map

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

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

```
vec = as.integer(1:10)

tryCatch(
  expr = {

    # this is what we want you to look at

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

  },

  # do not pay attention to this
  error = function(e){
    print(toString(e))
  }
)
```

```
)
#> [1] "Error: Can't coerce element 1 from a double to a integer\n"
```

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

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

#### 568 A note on invoke

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

## 572 4.3 Working with lists

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

### 575 4.3.1 Filtering lists

576 Lists can be filtered on any predicate using `keep`, while the special case `compact` is applied  
 577 when the empty elements of a list are to be filtered out. `discard` is the opposite of `keep`,  
 578 and keeps only elements not satisfying a condition. Again, the predicate is specified by  
 579 `.p`.

```
# a list containing numbers
this_list = list(a = 1, b = -1, c = 2, d = NULL, e = NA)

# remove the empty element
# this must be done before using keep on the list
this_list = compact(this_list)

# use discard to remove the NA
this_list = discard(this_list, .p = is.na)

# keep list elements which are positive
keep(this_list, .p = function(x){ x > 0 })
#> $a
#> [1] 1
#>
#> $c
#> [1] 2
```

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

```

1:10 %>%
  head_while(.p = function(x) x < 5)
#> [1] 1 2 3 4

```

### 582 4.3.2 Summarising lists

583 The purrr functions every, some, has\_element, detect, detect\_index, and vec\_depth  
 584 help determine whether a list passes a certain logical test or not. These are seldom used  
 585 and are not discussed here.

### 586 4.3.3 Reduction and accumulation

587 reduce helps combine elements along a list using a specific function. Consider the exam-  
 588 ple below where list elements are concatenated into a single vector.

```

this_list = list(a = 1:3, b = 3:4, c = 5:10)

reduce(this_list, c)
#> [1] 1 2 3 3 4 5 6 7 8 9 10

```

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

```

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

reduce(this_list, intersect)
#> [1] 3

```

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

```

# make a list
this_list = list(a = 1:3, b = 3:6, c = 5:10, d = c(1,2,5,10,12))

# a multiple accumulate can help
accumulate(this_list, union, .dir = "forward")
#> $a
#> [1] 1 2 3
#>
#> $b
#> [1] 1 2 3 4 5 6
#>
#> $c
#> [1] 1 2 3 4 5 6 7 8 9 10
#>

```

```
#> $d
#> [1] 1 2 3 4 5 6 7 8 9 10 12
```

#### 595 4.3.4 Miscellaneous operation

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

```
# use prepend to add values to the head of a list
prepend(x = list("a", "b"), values = list("1", "2"))
#> [[1]]
#> [1] "1"
#>
#> [[2]]
#> [1] "2"
#>
#> [[3]]
#> [1] "a"
#>
#> [[4]]
#> [1] "b"

# use splice to add multiple elements together
splice(list("a", "b"), list("1", "2"), "something else")
#> [[1]]
#> [1] "a"
#>
#> [[2]]
#> [1] "b"
#>
#> [[3]]
#> [1] "1"
#>
#> [[4]]
#> [1] "2"
#>
#> [[5]]
#> [1] "something else"
```

599 `flatten` has a similar behaviour, and converts a list of vectors or list of lists to a single  
 600 list-like object. `flatten_*` options allow the output type to be specified.

```
this_list = list(a = rep("a", 3),
                  b = rep("b", 4))

this_list
```

```

#> $a
#> [1] "a" "a" "a"
#>
#> $b
#> [1] "b" "b" "b" "b"

# use flatten chr to get a character vector
flatten_chr(this_list)
#> [1] "a" "a" "a" "b" "b" "b" "b"

```

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

```

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

# from a list of lists where cars are divided by cylinders and then
# gears, this is now a list of lists where cars are divided by
# gears and then cylinders
transpose(this_list[1])
#> $`3`
#> $`3`$`4`
#>
#>      mpg cyl disp hp drat   wt  qsec vs am gear carb
#> Toyota Corona 21.5   4 120 97  3.7 2.46   20   1   0    3    1
#>
#>
#> $`4`
#> $`4`$`4`
#>
#>      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
#> Datsun 710   22.8   4 108.0  93 3.85 2.32 18.6   1   1    4    1
#> Merc 240D    24.4   4 146.7  62 3.69 3.19 20.0   1   0    4    2
#> Merc 230     22.8   4 140.8  95 3.92 3.15 22.9   1   0    4    2
#> Fiat 128     32.4   4  78.7  66 4.08 2.20 19.5   1   1    4    1
#> Honda Civic  30.4   4  75.7  52 4.93 1.61 18.5   1   1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.83 19.9   1   1    4    1
#> Fiat X1-9    27.3   4  79.0  66 4.08 1.94 18.9   1   1    4    1
#> Volvo 142E   21.4   4 121.0 109 4.11 2.78 18.6   1   1    4    2
#>
#>
#> $`5`
#> $`5`$`4`
#>
#>      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
#> Porsche 914-2 26.0   4 120.3  91 4.43 2.14 16.7   0   1    5    2
#> Lotus Europa  30.4   4  95.1 113 3.77 1.51 16.9   1   1    5    2

```