TRES Tidyverse Tutorial

Raphael, Pratik and Theo

2020-06-11

Contents

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

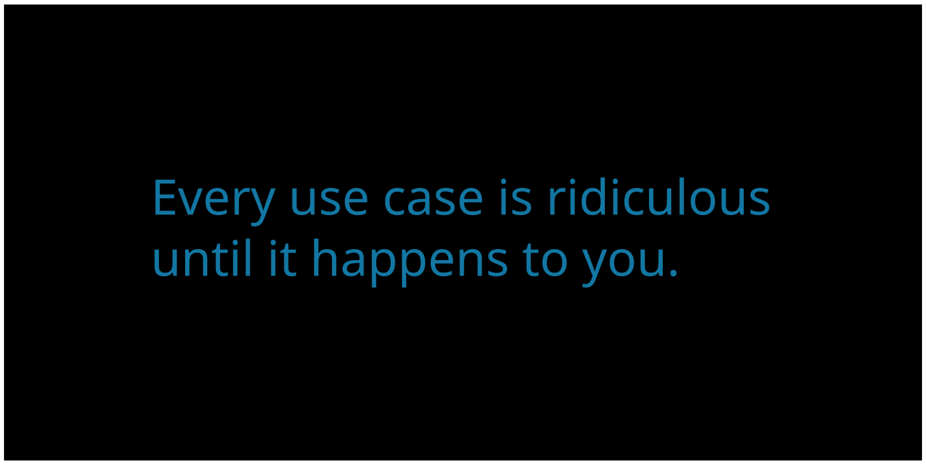
Join

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

*Tentative dates.

Chapter 1

Reading files and string manipulation



Every use case is ridiculous
until it happens to you.

Load the packages for the day.

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

1.1 Data import and export with `readr`

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

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.

```r
# 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.

1.1.3 Reading and writing lines

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

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

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

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

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

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

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

```r
# 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
```

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

Excel files

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

1.2 String manipulation with `stringr`

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

`stringr` functions begin with `str_`.

1.2.1 Putting strings together

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

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

```r
# 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"
```

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

```r
# 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"
```

1.2.2 Detecting strings

Count the frequency of a pattern in a string with `str_count`. Returns an inteegr. 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.

```r
# 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
```

Vectorising over both string and pattern works as expected.

```r
# 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("abababababa", "banana"),
          pattern = c("a", "b",
                      "b", "a"))
#> [1] 5 1 4 3
```

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

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

Detect whether a string starts or ends with a pattern. Also vectorised. Both have a `negate` 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
```

`str_subset` [WHICH IS NOT RELATED TO `str_sub`] helps with subsetting a character vector 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 elements.

```
# 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 patttern 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+`) seperated 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})[\\-\\/]([a-z]+)")
#> [[1]]
#> [1] "1970-somemonth"    "1990/anothermonth"
#>
#> [[2]]
#> [1] "1990-somemonth"    "2001/anothermonth"
```

1.2.5 Breaking strings apart

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

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"
```

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

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

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

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

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

1.2.6 Replacing string elements

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

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

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

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

1.2.7 Subsetting within strings

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

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"
```

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

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

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

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

1.2.8 Padding and truncating strings

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

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

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
```

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

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

1.4 Strings in ggplot

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

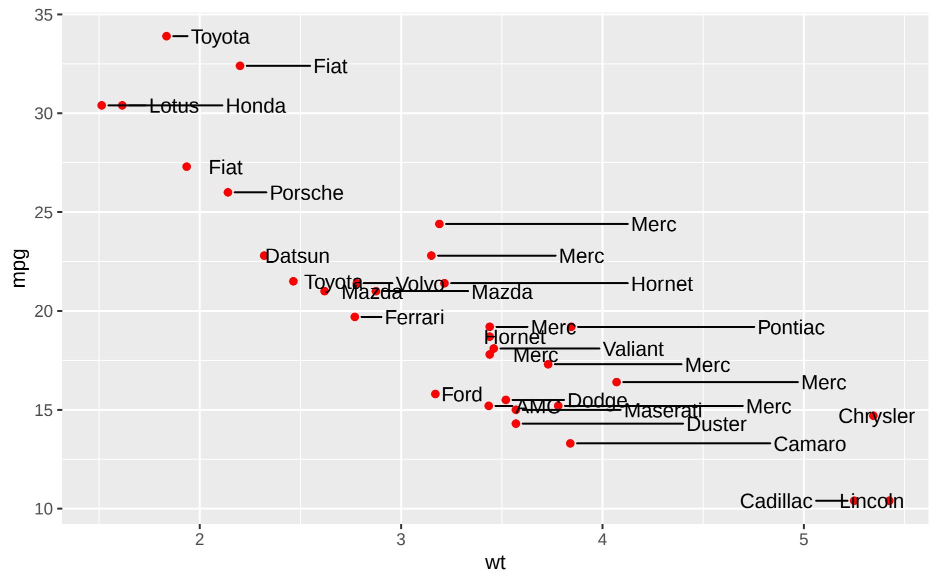
Often, these may overlap when the data are closely spaced. The package `ggrepel` offers another `geom`, `geom_text_repel` (and the related `geom_label_repel`) that help arrange text on a plot so it doesn't overlap with other features. This is *not perfect*, but it works more often than not.

More examples can be found on the ggrepl website.

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

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

```r
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)
```

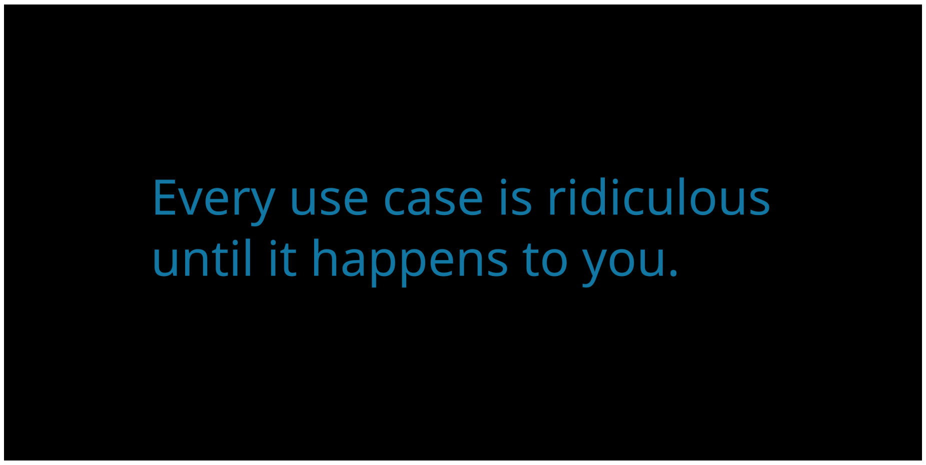


This is not a good looking plot, because it breaks other rules of plot design, such as whether this sort of plot should be made at all. Labels and text need to be applied sparingly, for example drawing attention or adding information to outliers.

Chapter 2

Reshaping data tables in the tidyverse, and other things

Raphael Scherrer



Every use case is ridiculous until it happens to you.

```r
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" "array"

# 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
```

Summary: Tidy or not tidy

To sum up, the definition of what is tidy and what is not is somewhat subjective. Tables can be in long or wide format, and depending on the complexity of a dataset, there may even be some intermediate states. To be clear, the tidyverse does not only accept long tables, and wide tables may sometimes be the way to go. This is very use-case specific. Have a clear idea of what you want to do with your data (what tidyverse tools you will use), and use that to figure which format makes more sense. And remember, `tidyr` is here to easily do the switching for you.

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 form 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
```

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

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

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

2.3.2 Handling missing values

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

```
# We replace 100 random observations by NAs
ii <- sample(nrow(data), 100)
data$value[ii] <- NA
data
#> # A tibble: 300 x 4
#>      id island  variable    value
#>   <int> <chr>   <chr>       <dbl>
#> 1      1 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
```

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
```

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
```

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

```
data <- unite(data, day, month, year, col = "date", sep = "-")
data
#> # A tibble: 300 x 5
#>      id island  variable    value date
#>   <int> <chr>   <chr>       <dbl> <chr>
#> 1     1 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
```

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

```
separate(data, date, into = c("day", "month", "year"))
#> # A tibble: 300 x 7
#>        id island  variable    value day   month year
#>     <int> <chr>   <chr>       <dbl> <chr> <chr> <chr>
#> 1       1 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
```

But note that the `day`, `month` and `year` columns are now of class `character` and not `integer` anymore. This is because they result from the splitting of `date`, which itself was a `character` column.

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

```
separate_rows(data, date)
#> # A tibble: 900 x 5
#>        id island  variable    value date
#>     <int> <chr>   <chr>       <dbl> <chr>
#> 1       1 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
```

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

Of course, if sites A and B are on Isabela, they cannot be on Santa Cruz, where we have sites C and D instead. It would not make sense to `expand` assuming that `island` and `site` are 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
```

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

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

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

which nicely keeps the rest of the columns in the tibble and just adds the missing combinations.

2.3.5 Nesting

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

2.3.6 What else can be tidied up?

2.3.6.1 Model output with `broom`

Check out the `broom` package and its `tidy` function to tidy up messy linear model output, e.g.

```
library(broom)
fit <- lm(mpg ~ cyl, mtcars)
summary(fit)
#>
#> Call:
#> lm(formula = mpg ~ cyl, data = mtcars)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -4.981 -2.119  0.222  1.072  7.519
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   37.885      2.074   18.27  < 2e-16 ***
#> cyl           -2.876      0.322   -8.92  6.1e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.21 on 30 degrees of freedom
#> Multiple R-squared:  0.726,  Adjusted R-squared:  0.717
#> F-statistic: 79.6 on 1 and 30 DF,  p-value: 6.11e-10
tidy(fit) # returns a tibble
#> # A tibble: 2 x 5
#>   term        estimate std.error statistic  p.value
#>   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
#> 1 (Intercept)    37.9      2.07      18.3  8.37e-18
#> 2 cyl            -2.88     0.322     -8.92 6.11e-10
```

The `broom` package is just one package among a series of packages together known as tidymodels that deal with statistical models according to the tidyverse philosophy, and those include machine learning models.

2.3.6.2 Graphs with `tidygraph`

For some datasets, sometimes there is no trivial and intuitive way to store them into a table. This is the case, for example, for data underlying graphs (as in networks), which contain information about relations between entities. What is the unit of observation in a network? A node? An edge between two nodes? Nodes and edges in a network may each have node- or edge-specific variables mapped to them, and both may be equally valid units of observation. The `tidygraph` package has tools to store graph-data in a tidyverse-friendly object, consisting of two tibbles: one for node-specific information, the other for edge-specific information. This package goes hand in hand with the `ggraph`, that makes plotting networks compatible with the grammar of graphics.

2.3.6.3 Trees with `tidytree`

Phylogenetic trees are a special type of graphs suffering from the same issue, i.e. of being non-trivial to store in a table. The `tidytree` package and its companion `treeio` offer an interface to convert tree-like objects (from most format used by other packages and software) into a tidyverse-friendly format. Again, the point is that the rest of the tidyverse can be used to wrangle or plot this type of data in the same way as one would do with regular tabular data. For plotting a `tidytree` with the grammar of graphics, see `ggtree`.

2.4 Extra: factors and the `forcats` package

```
library(forcats)
```

Categorical variables can be stored in R as character strings in `character` or `factor` objects. 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` of `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 implicitely 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.

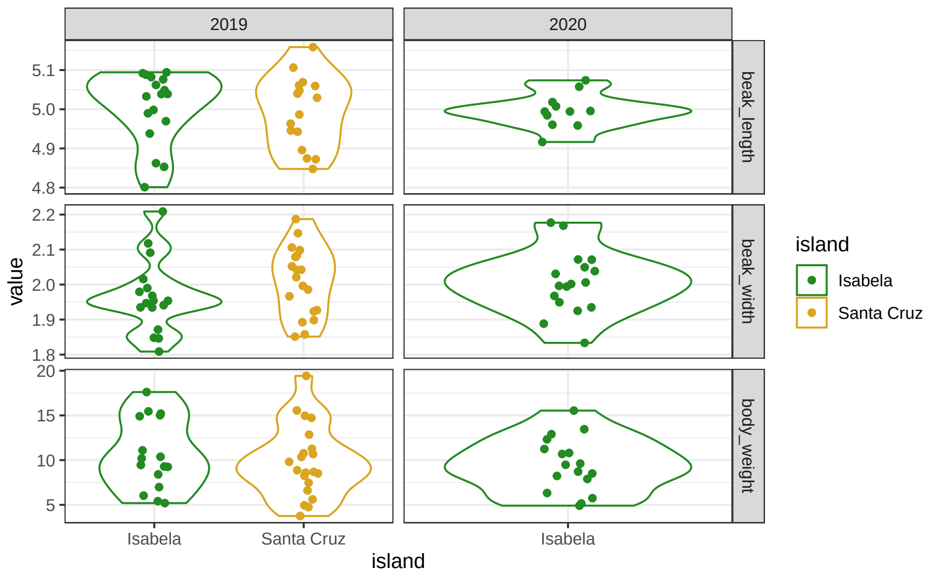
But this is a tidyverse tutorial, so I would like to introduce here the package `forcats`, which offers tools to manipulate factors. First of all, most tools from `stringr` *will work* on factors. The `forcats` functions expand the string manipulation toolbox with factor-specific utilities. Similar in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

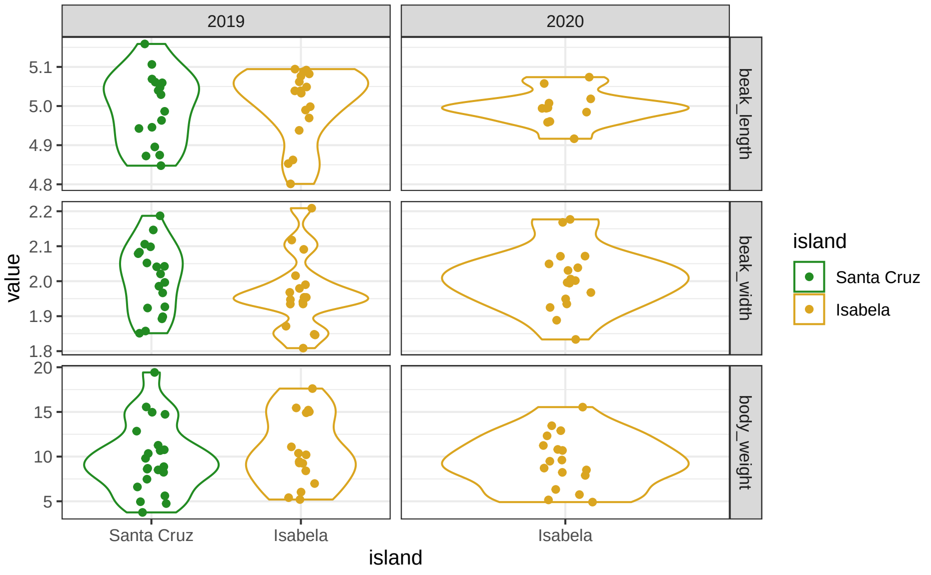
```
# Make the plotting code a function so we can re-use it without copying and pasting
my_plot <- function(data) {

  # 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
```



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

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

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

Note that this is equivalent to doing:

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

2.4.4 Other things

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

2.4.5 Take home message for forcats

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

2.5 External resources

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

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

Chapter 3

Data manipulation with `dplyr`

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

3.1 Introduction

3.1.1 Foreword on `dplyr`

`dplyr` is tasked with performing all sorts of transformations on a dataset.

The structure of `dplyr` revolves around a set of functions, the so-called **verbs**, that share a common syntax and logic, and are meant to work with one another in chained operations. Chained operations are performed with the pipe operator (`%>%`), that will be introduced in section 3.2.2.

The basic syntax is `verb(data, variable)`, where `data` is a data frame and `variable` is the name of one or more columns containing a set of values for each observation.

There are 5 main verbs, which names already hint at what they do: `rename()`, `select()`, `filter()`, `mutate()`, and `summarise()`. I'm going to introduce each of them (and a couple more) through the following sections.

3.1.2 Example data

Through this tutorial, we will be using mammal trait data from the Phylacine database. Let's have a peek at what it contains.

```
phylacine <- 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>
```

`readr` automatically loads the data in a `tibble`, as we have seen in chapter 1 and 2. Calling the tibble gives a nice preview of what it contains. We have data for 5,831 mammal species, and the variables contain information on taxonomy, (broad) habitat, mass, IUCN status, and diet.

If you remember Section 1.2 on tidy data, you may see that this data isn't exactly tidy. In fact, some columns are in wide (and messy) format, like the "habitat" (terrestrial, marine, etc.) and diet columns.

`dplyr` actually does not require your data to be strictly tidy. If you feel that your data satisfies the definition "one observation per row, one variable per column", that's probably good enough.

I use a `tibble` here, but `dplyr` works equally well on base data frames. In fact, `dplyr` is built for `data.frame` objects, and tibbles are data frames. Therefore, tibbles are mortal.

3.2 Working with existing variables

3.2.1 Renaming variables with `rename()`

The variable names in the phylacine dataset are descriptive, but quite unpractical. Typing `Binomial.1.2.` is cumbersome and subject to typos (in fact, I just made one). `binomial` would be much simpler to use.

Changing names is straightforward with `rename()`.

```
rename(.data = phylacine, "binomial" = Binomial.1.2)
#> # A tibble: 5,831 x 24
#>   binomial Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>    <chr>     <chr>      <chr>     <chr>             <dbl>  <dbl>
#> 1 Abditom~ Rodentia  Muridae    Abditomys latidens              1      0
#> 2 Abeomel~ Rodentia  Muridae    Abeomelo~ sevia                 1      0
#> 3 Abraway~ Rodentia  Cricetidae Abrawaya~ ruschii               1      0
#> 4 Abrocom~ Rodentia  Abrocomid~ Abrocoma  bennettii             1      0
#> 5 Abrocom~ Rodentia  Abrocomid~ Abrocoma  boliviensis           1      0
#> 6 Abrocom~ 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>
```

The first argument is always `.data`, the data table you want to apply change to. Note how columns are referred to. Once the data table as been passed as an argument, there is no need to refer to it directly anymore, `dplyr` understands that you're dealing with variables inside that data frame. So drop that `data$var`, `data[, "var"]`, and forget the very existence of `attach()` / `detach()`.

You can refer to variables names either with strings or directly as objects, whether you're reading or creating them:

```
rename(
  phylacine,
  # this works
  binomial = Binomial.1.2
)
rename(
  phylacine,
  # this works too!
  binomial = "Binomial.1.2"
)
rename(
  phylacine,
  # guess what
  "binomial" = "Binomial.1.2"
)
```

I have applied similar changes to all variables in the dataset. Here is what the new names look like:

```
#> # A tibble: 5,831 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~           1      0          0      0
#> 2 Abeomel~ Rode~ Murid~ Abeo~ sevia             1      0          0      0
#> 3 Abraway~ Rode~ Crice~ Abra~ ruschii           1      0          0      0
#> 4 Abrocom~ Rode~ Abroc~ Abro~ bennet~           1      0          0      0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ bolivi~           1      0          0      0
#> 6 Abrocom~ Rode~ Abroc~ Abro~ budini            1      0          0      0
#> # ... with 5,825 more rows, and 15 more variables: 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 <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

3.2.2 The pipe operator %>%

If you have already come across pieces of code using the tidyverse, chances are that you have seen this odd symbol. While the pipe is not strictly-speaking a part of the tidyverse (it comes from its own package, `magrittr`), it is imported along with each package and widely used in conjunction with its functions. What does it do? Consider the following example with `rename()`:

```
phylacine2 <- readr::read_csv("data/phylacine_traits.csv")
# regular syntax
rename(phylacine2, "binomial" = "Binomial.1.2")
#> # A tibble: 5,831 x 24
#>   binomial Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>    <chr>     <chr>      <chr>     <chr>             <dbl>  <dbl>
#> 1 Abditom~ Rodentia  Muridae    Abditomys latidens              1      0
#> 2 Abeomel~ Rodentia  Muridae    Abeomelo~ sevia                 1      0
#> 3 Abraway~ Rodentia  Cricetidae Abrawaya~ ruschii               1      0
#> 4 Abrocom~ Rodentia  Abrocomid~ Abrocoma  bennettii             1      0
#> 5 Abrocom~ Rodentia  Abrocomid~ Abrocoma  boliviensis           1      0
#> 6 Abrocom~ 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>
# alternative syntax with the pipe operator
phylacine2 %>% rename("binomial" = "Binomial.1.2")
#> # A tibble: 5,831 x 24
#>   binomial Order.1.2 Family.1.2 Genus.1.2 Species.1.2 Terrestrial Marine
#>   <chr>    <chr>     <chr>       <chr>      <chr>              <dbl>  <dbl>
#> 1 Abditom~ Rodentia  Muridae     Abditomys  latidens               1      0
#> 2 Abeomel~ Rodentia  Muridae     Abeomelo~  sevia                  1      0
#> 3 Abraway~ Rodentia  Cricetidae  Abrawaya~  ruschii                1      0
#> 4 Abrocom~ Rodentia  Abrocomid~  Abrocoma   bennettii              1      0
#> 5 Abrocom~ Rodentia  Abrocomid~  Abrocoma   boliviensis            1      0
#> 6 Abrocom~ 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>
```

Got it? The pipe takes the object on its left-side and silently feeds it to the *first* argument of the function on its right-side. It could be read as “take x, then do…”. The reason for using the pipe is because it makes code syntax closer to the syntax of a sentence, and therefore, easier and faster for your brain to process (and write!) the code. In particular, the pipe enables easy chains of operations, where you apply something to an object, then apply something else to the outcome, and so on… Through the later sections, you will see some examples of chained operations with `dplyr` functions, but for that I first need to introduce a couple more verbs.

Using the pipe can be quite unsettling at first, because you are not used to think in this way. But if you push a bit for it, I promise it will make things a lot easier (and it’s quite addictive!). To avoid typing the tedious symbols, `magrittr` installs a shortcut for you in RStudio. Use `Ctrl + Shift + M` on Windows, and `Cmd + Shift + M` on MacOS.

Finally I should emphasize that the use of the pipe isn’t limited to the tidyverse, but extends to almost all R functions. Remember that by default the piped value is always matched to the first argument of the following function

```
5 %>% rep(3)
#> [1] 5 5 5
"meow" %>% cat()
#> meow
```

If you need to pass the left-hand side to an argument other than the first, you can use the dot place-holder `.`.

```
"meow" %>% cat("cats", "go")
#> meow cats go
```

Because of its syntax, most base R operators are not compatible with the pipe (but this is very rarely needed). If needed, `magrittr` introduces alternative functions for operators.

Subsetting operators can be piped, with the dot place-holder.

```
# 5 %>% * 3 # no, won't work
# 5 %>% .* 3 # neither
5 %>% magrittr::multiply_by(3) # yes
#> [1] 15

# subsetting
list("monkey see", "monkey_do") %>% .[[2]]
#> [1] "monkey_do"
phylacine %>% .$binomial %>% head()
#> [1] "Abditomys_latidens"   "Abeomelomys_sevia"    "Abrawayaomys_ruschii"
#> [4] "Abrocoma_bennettii"   "Abrocoma_boliviensis" "Abrocoma_budini"
```

Because subsetting in this way is particularly hideous, `dplyr` delivers a function to extract values from a single variable. In only works on tables, though.

```
phylacine %>% pull(binomial) %>% head()
#> [1] "Abditomys_latidens"   "Abeomelomys_sevia"    "Abrawayaomys_ruschii"
#> [4] "Abrocoma_bennettii"   "Abrocoma_boliviensis" "Abrocoma_budini"
```

3.2.3 Select variables with `select()`

To extract a set of variables (i.e. columns), use the conveniently-named `select()`. The basic syntax is the same as `rename()`: pass your data as the first argument, then call the variables to select, quoted or not.

```
# Single variable
phylacine %>% select(binomial)
#> # A tibble: 5,831 x 1
#>   binomial
#>   <chr>
#> 1 Abditomys_latidens
#> 2 Abeomelomys_sevia
#> 3 Abrawayaomys_ruschii
#> 4 Abrocoma_bennettii
#> 5 Abrocoma_boliviensis
#> 6 Abrocoma_budini
#> # ... with 5,825 more rows
# A set of variables
phylacine %>% select(genus, "species", mass_g)
#> # A tibble: 5,831 x 3
#>   genus        species     mass_g
#>   <chr>        <chr>        <dbl>
#> 1 Abditomys    latidens       269
#> 2 Abeomelomys  sevia           52
#> 3 Abrawayaomys ruschii         63
#> 4 Abrocoma     bennettii     250
#> 5 Abrocoma     boliviensis   158
#> 6 Abrocoma     budini        361.
#> # ... with 5,825 more rows
# A range of contiguous variables
phylacine %>% select(family:terrestrial)
#> # A tibble: 5,831 x 4
#>   family      genus         species     terrestrial
#>   <chr>       <chr>         <chr>             <dbl>
#> 1 Muridae     Abditomys     latidens              1
#> 2 Muridae     Abeomelomys   sevia                 1
#> 3 Cricetidae  Abrawayaomys ruschii                1
#> 4 Abrocomidae Abrocoma      bennettii             1
#> 5 Abrocomidae Abrocoma      boliviensis           1
#> 6 Abrocomidae Abrocoma      budini                1
#> # ... with 5,825 more rows
```

610 You can select by variable numbers. This is not recommended, as prone to
611 errors, especially if you change the variable order.

```
phylacine %>% select(2)
#> # A tibble: 5,831 x 1
#>   order
#>   <chr>
#> 1 Rodentia
#> 2 Rodentia
#> 3 Rodentia
#> 4 Rodentia
#> 5 Rodentia
#> 6 Rodentia
#> # ... with 5,825 more rows
```

612 `select()` can also be used to *exclude* variables:

```
phylacine %>% select(-binomial)
#> # A tibble: 5,831 x 23
#>   order family genus species terrestrial marine freshwater aerial
#>   <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Rode~ Murid~ Abdi~ latide~           1      0          0      0
#> 2 Rode~ Murid~ Abeo~ sevia             1      0          0      0
#> 3 Rode~ Crice~ Abra~ ruschii           1      0          0      0
#> 4 Rode~ Abroc~ Abro~ bennet~           1      0          0      0
#> 5 Rode~ Abroc~ Abro~ bolivi~           1      0          0      0
#> 6 Rode~ Abroc~ Abro~ budini            1      0          0      0
#> # ... with 5,825 more rows, and 15 more variables: 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 <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
phylacine %>% select(-(binomial:species))
#> # A tibble: 5,831 x 19
#>   terrestrial marine freshwater aerial life_habit_meth~ life_habit_sour~ mass_g
#>         <dbl>  <dbl>      <dbl>  <dbl> <chr>            <chr>             <dbl>
#> 1           1      0          0      0 Reported         IUCN. 2016. IUC~   269
#> 2           1      0          0      0 Reported         IUCN. 2016. IUC~    52
#> 3           1      0          0      0 Reported         IUCN. 2016. IUC~    63
#> 4           1      0          0      0 Reported         IUCN. 2016. IUC~   250
#> 5           1      0          0      0 Reported         IUCN. 2016. IUC~   158
#> 6           1      0          0      0 Reported         IUCN. 2016. IUC~   361.
#> # ... with 5,825 more rows, and 12 more variables: mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

`select()` and `rename()` are pretty similar, and in fact, `select()` can also rename variables along the way:

```
phylacine %>% select("linnaeus" = binomial)
#> # A tibble: 5,831 x 1
#>   linnaeus
#>   <chr>
#> 1 Abditomys_latidens
#> 2 Abeomelomys_sevia
#> 3 Abrawayaomys_ruschii
#> 4 Abrocoma_bennettii
#> 5 Abrocoma_boliviensis
#> 6 Abrocoma_budini
#> # ... with 5,825 more rows
```

And you can mix all of that at once:

```
phylacine %>% select(
  "fam" = family,
  genus:freshwater,
  -terrestrial
)
#> # A tibble: 5,831 x 5
#>   fam          genus        species     marine freshwater
#>   <chr>        <chr>        <chr>        <dbl>      <dbl>
#> 1 Muridae      Abditomys    latidens         0          0
#> 2 Muridae      Abeomelomys  sevia            0          0
#> 3 Cricetidae   Abrawayaomys ruschii          0          0
#> 4 Abrocomidae  Abrocoma     bennettii        0          0
#> 5 Abrocomidae  Abrocoma     boliviensis      0          0
#> 6 Abrocomidae Abrocoma     budini            0           0
#> # ... with 5,825 more rows
```

3.2.4 Select variables with helpers

The Rstudio team just released `dplyr 1.0.0`, and along with it, some nice helper functions to ease the selection of a set of variables. I give three examples here, and encourage you to look at the documentation (`?select()`) to find out more.

```
phylacine %>% select(where(is.numeric))
#> # A tibble: 5,831 x 8
#>   terrestrial marine freshwater aerial mass_g diet_plant diet_vertebrate
#>         <dbl>  <dbl>      <dbl>  <dbl>  <dbl>      <dbl>           <dbl>
#> 1           1      0          0      0   269         100               0
#> 2           1      0          0      0    52          78               3
#> 3           1      0          0      0    63          88               1
#> 4           1      0          0      0   250         100               0
#> 5           1      0          0      0   158         100               0
#> 6           1      0          0      0   361.        100               0
#> # ... with 5,825 more rows, and 1 more variable: diet_invertebrate <dbl>
phylacine %>% select(contains("mass") | contains("diet"))
#> # A tibble: 5,831 x 10
#>   mass_g mass_method mass_source mass_comparison mass_comparison~ diet_plant
#>    <dbl> <chr>       <chr>       <chr>           <chr>                 <dbl>
#> 1   269  Reported    Smith, F. ~ <NA>            <NA>                    100
#> 2    52  Reported    Smith, F. ~ <NA>            <NA>                     78
#> 3    63  Reported    Smith, F. ~ <NA>            <NA>                     88
#> 4   250  Reported    Smith, F. ~ <NA>            <NA>                    100
#> 5   158  Reported    Smith, F. ~ <NA>            <NA>                    100
#> 6   361. Assumed is~ Journal of~ Abrocoma_ciner~ Journal of Mamm~        100
#> # ... with 5,825 more rows, and 4 more variables: diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>

habitats <- c("terrestrial", "marine", "arboreal", "fossorial", "freshwater")
phylacine %>% select(any_of(habitats))
#> # A tibble: 5,831 x 3
#>   terrestrial marine freshwater
#>         <dbl>  <dbl>      <dbl>
#> 1           1      0          0
#> 2           1      0          0
#> 3           1      0          0
#> 4           1      0          0
#> 5           1      0          0
#> 6           1      0          0
#> # ... with 5,825 more rows
```

3.2.5 Rearranging variable order with `relocate()`

The order of variables seldom matters in `dplyr`, but due to popular demand, `dplyr` now has a dedicated verb to rearrange the order of variables. The syntax is identical to `rename()`, `select()`.

```
phylacine %>% relocate(mass_g, .before = binomial)
#> # A tibble: 5,831 x 24
#>   mass_g binomial order family genus species terrestrial marine freshwater
#>    <dbl> <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>
#> 1   269  Abditom~ Rode~ Murid~ Abdi~ latide~           1      0          0
#> 2    52  Abeomel~ Rode~ Murid~ Abeo~ sevia             1      0          0
#> 3    63  Abraway~ Rode~ Crice~ Abra~ ruschii           1      0          0
#> 4   250  Abrocom~ Rode~ Abroc~ Abro~ bennet~           1      0          0
#> 5   158  Abrocom~ Rode~ Abroc~ Abro~ bolivi~           1      0          0
#> 6   361. Abrocom~ Rode~ Abroc~ Abro~ budini            1      0          0
#> # ... with 5,825 more rows, and 15 more variables: aerial <dbl>,
#> #   life_habit_method <chr>, life_habit_source <chr>, mass_method <chr>,
#> #   mass_source <chr>, mass_comparison <chr>, mass_comparison_source <chr>,
#> #   island_endemicity <chr>, iucn_status <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
phylacine %>% relocate(starts_with("diet"), .after = species)
#> # A tibble: 5,831 x 24
#>   binomial order family genus species diet_plant diet_vertebrate
#>   <chr>    <chr> <chr>  <chr> <chr>        <dbl>           <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~        100               0
#> 2 Abeomel~ Rode~ Murid~ Abeo~ sevia           78               3
#> 3 Abraway~ Rode~ Crice~ Abra~ ruschii         88               1
#> 4 Abrocom~ Rode~ Abroc~ Abro~ bennet~        100               0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ bolivi~        100               0
#> 6 Abrocom~ Rode~ Abroc~ Abro~ budini         100               0
#> # ... with 5,825 more rows, and 17 more variables: diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>, terrestrial <dbl>, marine <dbl>,
#> #   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 <chr>, added_iucn_status <chr>
```

3.3 Working with observations

3.3.1 Ordering rows by value - `arrange()`

`arrange()` sorts rows in the data by **ascending** value for a given variable. Use the wrapper `desc()` to sort by descending values instead.

```
# Smallest mammals
phylacine %>%
  arrange(mass_g) %>%
  select(binomial, mass_g)
#> # A tibble: 5,831 x 2
#>   binomial            mass_g
#>   <chr>                <dbl>
#> 1 Sorex_yukonicus        1.6
#> 2 Crocidura_levicula     1.8
#> 3 Suncus_remyi           1.8
#> 4 Crocidura_lusitania    2
#> 5 Kerivoula_minuta       2.1
#> 6 Suncus_etruscus        2.1
#> # ... with 5,825 more rows

# Largest mammals
phylacine %>%
  arrange(desc(mass_g)) %>%
  select(binomial, mass_g)
#> # A tibble: 5,831 x 2
#>   binomial                  mass_g
#>   <chr>                      <dbl>
#> 1 Balaenoptera_musculus  190000000
#> 2 Balaena_mysticetus     100000000
#> 3 Balaenoptera_physalus   70000000
#> 4 Caperea_marginata       32000000
#> 5 Megaptera_novaeangliae  30000000
#> 6 Eschrichtius_robustus   28500000
#> # ... with 5,825 more rows

# Extra variables are used to sort ties in the first variable
phylacine %>%
  arrange(mass_g, desc(binomial)) %>%
  select(binomial, mass_g)
#> # A tibble: 5,831 x 2
#>   binomial            mass_g
#>   <chr>                <dbl>
#> 1 Sorex_yukonicus        1.6
#> 2 Suncus_remyi           1.8
#> 3 Crocidura_levicula     1.8
#> 4 Crocidura_lusitania    2
#> 5 Suncus_etruscus        2.1
#> 6 Kerivoula_minuta       2.1
#> # ... with 5,825 more rows
```

Important: `NA` values, if present, are always ordered at the end!

3.3.2 Subset rows by position - `slice()`

Use `slice()` and its variants to extract particular rows.

```
phylacine %>% slice(3) # third row
#> # A tibble: 1 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abraway~ Rode~ Crice~ Abra~ ruschii           1      0          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
phylacine %>% slice(5, 1, 2) # fifth, first and second row
#> # A tibble: 3 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abrocom~ Rode~ Abroc~ Abro~ bolivi~           1      0          0      0
#> 2 Abditom~ Rode~ Murid~ Abdi~ latide~           1      0          0      0
#> 3 Abeomel~ Rode~ Murid~ Abeo~ sevia             1      0          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
phylacine %>% slice(rep(3, 2)) # duplicate the third row
#> # A tibble: 2 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abraway~ Rode~ Crice~ Abra~ ruschii           1      0          0      0
#> 2 Abraway~ Rode~ Crice~ Abra~ ruschii           1      0          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
```

```
phylacine %>% slice(-c(2:5830)) # exclude all but first and last row
#> # A tibble: 2 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~           1      0          0      0
#> 2 Zyzomys~ Rode~ Murid~ Zyzo~ woodwa~           1      0          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>

phylacine %>% slice_tail(n = 3) # last three rows
#> # A tibble: 3 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Zyzomys~ Rode~ Murid~ Zyzo~ palata~           1      0          0      0
#> 2 Zyzomys~ Rode~ Murid~ Zyzo~ pedunc~           1      0          0      0
#> 3 Zyzomys~ Rode~ Murid~ Zyzo~ woodwa~           1      0          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
phylacine %>% slice_max(mass_g) # largest mammal
#> # A tibble: 1 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Balaeno~ Ceta~ Balae~ Bala~ muscul~           0      1          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
```

You can also sample random rows in the data:

```
phylacine %>% slice_sample() # a random row
#> # A tibble: 1 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Crocidu~ Euli~ Soric~ Croc~ levicu~           1      0          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
```

```
# bootstrap
phylacine %>% slice_sample(n = 5831, replace = TRUE)
#> # A tibble: 5,831 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Rhinolo~ Chir~ Rhino~ Rhin~ adami             0      0          0      1
#> 2 Hylomys~ Euli~ Erina~ Hylo~ megalo~           1      0          0      0
#> 3 Sciurus~ Rode~ Sciur~ Sciu~ yucata~           1      0          0      0
#> 4 Emballo~ Chir~ Embal~ Emba~ alecto            0      0          0      1
#> 5 Pteralo~ Chir~ Ptero~ Pter~ taki              0      0          0      1
#> 6 Lasiorh~ Dipr~ Vomba~ Lasi~ latifr~           1      0          0      0
#> # ... with 5,825 more rows, and 15 more variables: 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 <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

3.3.3 Subsetting rows by value with `filter()`

`filter()` does a similar job as `slice()`, but extract rows that satisfy a set of conditions. The conditions are supplied much the same way as you would do for an `if` statement.

Along with `mutate()` (next section), this is probably the function you are going to use the most.

For example, I might want to extract mammals above a given mass:

```
# megafauna
phylacine %>%
  filter(mass_g > 1e5) %>% # 100 kg
  select(binomial, mass_g)
#> # A tibble: 302 x 2
#>   binomial                      mass_g
#>   <chr>                          <dbl>
#> 1 Ailuropoda_melanoleuca       108400
#> 2 Alcelaphus_buselaphus        171002.
#> 3 Alces_alces                  356998
#> 4 Archaeoindris_fontoynonti    160000
#> 5 Arctocephalus_forsteri       101250
#> 6 Arctocephalus_pusillus       178500
#> # ... with 296 more rows

# non-extinct megafauna
```

```
phylacine %>%
  filter(mass_g > 1e5, iucn_status != "EP") %>%
  select(binomial, mass_g, iucn_status)
#> # A tibble: 178 x 3
#>   binomial                  mass_g iucn_status
#>   <chr>                      <dbl> <chr>
#> 1 Ailuropoda_melanoleuca  108400  VU
#> 2 Alcelaphus_buselaphus   171002. LC
#> 3 Alces_alces             356998  LC
#> 4 Arctocephalus_forsteri  101250  LC
#> 5 Arctocephalus_pusillus  178500  LC
#> 6 Arctocephalus_townsendi 105000  LC
#> # ... with 172 more rows
```

640 Are there any flying mammals that aren't bats?

```
phylacine %>%
  filter(aerial == 1, order != "Chiroptera")
#> # A tibble: 0 x 24
#> # ... with 24 variables: binomial <chr>, order <chr>, family <chr>,
#> #   genus <chr>, species <chr>, terrestrial <dbl>, marine <dbl>,
#> #   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 <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
# no :(
```

641 Are humans included in the table?

```
phylacine %>% filter(binomial == "Homo_sapiens")
#> # A tibble: 1 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Homo_sa~ Prim~ Homin~ Homo  sapiens           1      0          0      0
#> # ... with 15 more variables: 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 <chr>,
#> #   added_iucn_status <chr>, diet_plant <dbl>, diet_vertebrate <dbl>,
#> #   diet_invertebrate <dbl>, diet_method <chr>, diet_source <chr>
```

642 `filter()` can be used to deal with NAs:

```
phylacine %>%
  filter(!is.na(mass_comparison))
#> # A tibble: 754 x 24
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abrocom~ Rode~ Abroc~ Abro~ budini            1      0          0      0
#> 2 Abrocom~ Rode~ Abroc~ Abro~ famati~           1      0          0      0
#> 3 Abrocom~ Rode~ Abroc~ Abro~ shista~           1      0          0      0
#> 4 Abrocom~ Rode~ Abroc~ Abro~ uspall~           1      0          0      0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ vaccar~           1      0          0      0
#> 6 Acerodo~ Chir~ Ptero~ Acer~ humilis           0      0          0      1
#> # ... with 748 more rows, and 15 more variables: 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 <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

643 Tip: `dplyr` introduces the useful function `between()` that does exactly what
644 the name implies

```
between(1:5, 2, 4)
#> [1] FALSE  TRUE  TRUE  TRUE FALSE

# Mesofauna
phylacine %>%
  filter(mass_g > 1e3, mass_g < 1e5) %>%
  select(binomial, mass_g)
#> # A tibble: 1,126 x 2
#>   binomial                 mass_g
#>   <chr>                     <dbl>
#> 1 Acerodon_jubatus          1075
#> 2 Acinonyx_jubatus         46700
#> 3 Acratocnus_odontrigonus  22990
#> 4 Acratocnus_ye            21310
#> 5 Addax_nasomaculatus      70000.
#> 6 Aepyceros_melampus       52500.
#> # ... with 1,120 more rows

# same thing
phylacine %>%
  filter(mass_g %>% between(1e3, 1e5)) %>%
  select(binomial, mass_g)
#> # A tibble: 1,148 x 2
#>   binomial                 mass_g
#>   <chr>                     <dbl>
#> 1 Acerodon_jubatus          1075
#> 2 Acinonyx_jubatus         46700
#> 3 Acratocnus_odontrigonus  22990
#> 4 Acratocnus_ye            21310
#> 5 Addax_nasomaculatus      70000.
#> 6 Aepyceros_melampus       52500.
#> # ... with 1,142 more rows
```

Note that you can pipe operations inside function arguments as in the last line above (arguments are expressions, after all!).

3.4 Making new variables

3.4.1 Create new variables with `mutate()`

Very often in data analysis, you will want to create new variables, or edit existing ones. This is done easily through `mutate()`. For example, consider the diet data:

```
diet <- phylacine %>%
  select(
    binomial,
    contains("diet") & !contains(c("method", "source"))
  )
diet
#> # A tibble: 5,831 x 4
#>   binomial              diet_plant diet_vertebrate diet_invertebrate
#>   <chr>                      <dbl>           <dbl>             <dbl>
#> 1 Abditomys_latidens           100               0                 0
#> 2 Abeomelomys_sevia             78               3                19
#> 3 Abrawayaomys_ruschii          88               1                11
#> 4 Abrocoma_bennettii           100               0                 0
#> 5 Abrocoma_boliviensis         100               0                 0
#> 6 Abrocoma_budini              100               0                 0
#> # ... with 5,825 more rows
```

These three variables show the percentage of each category of food that make the diet of that species. They should sum to 100, unless the authors made a typo or other entry error. To assert this, I'm going to create a new variable, `total_diet`.

```
diet <- diet %>% mutate(
  "total_diet" = diet_vertebrate + diet_invertebrate + diet_plant
)
diet
#> # A tibble: 5,831 x 5
#>   binomial              diet_plant diet_vertebrate diet_invertebrate total_diet
#>   <chr>                      <dbl>           <dbl>             <dbl>      <dbl>
#> 1 Abditomys_latidens           100               0                 0        100
#> 2 Abeomelomys_sevia             78               3                19        100
#> 3 Abrawayaomys_ruschii          88               1                11        100
#> 4 Abrocoma_bennettii           100               0                 0        100
#> 5 Abrocoma_boliviensis         100               0                 0        100
#> 6 Abrocoma_budini              100                 0                   0         100
#> # ... with 5,825 more rows

all(diet$total_diet == 100)
#> [1] TRUE
# cool and good
```

`mutate()` adds a variable to the table, and keeps all other variables. Sometimes you may want to just keep the new variable, and drop the other ones. That's the job of `mutate()`'s twin sibling, `transmute()`. For example, I want to combine `diet_invertebrate` and `diet_vertebrate` together:

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertebrate
  )
#> # A tibble: 5,831 x 1
#>   diet_animal
#>         <dbl>
#> 1           0
#> 2          22
#> 3          12
#> 4           0
#> 5           0
#> 6           0
#> # ... with 5,825 more rows
```

You may want to keep some variables and drop others. You could pipe `mutate()` and `select()` to do so, or you could just pass the variables to keep to `transmute()`.

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertebrate,
    diet_plant
  )
#> # A tibble: 5,831 x 2
#>   diet_animal diet_plant
#>         <dbl>      <dbl>
#> 1           0        100
#> 2          22         78
#> 3          12         88
#> 4           0        100
#> 5           0        100
#> 6           0        100
#> # ... with 5,825 more rows
```

You can also refer to variables you're creating to derive new variables from them as part of the same operation, this is not an issue.

```
diet %>%
  transmute(
    "diet_animal" = diet_invertebrate + diet_vertebrate,
    diet_plant,
    "total_diet" = diet_animal + diet_plant
  )
#> # A tibble: 5,831 x 3
#>   diet_animal diet_plant total_diet
#>         <dbl>      <dbl>      <dbl>
#> 1           0        100        100
#> 2          22         78        100
#> 3          12         88        100
#> 4           0        100        100
#> 5           0        100        100
#> 6           0        100        100
#> # ... with 5,825 more rows
```

Sometimes, you may need to perform an operation based on the row number (I don't have a good example in mind). `tibble` has a built-in function to do just that:

```
phylacine %>%
  select(binomial) %>%
  tibble::rownames_to_column(var = "row_nb")
#> # A tibble: 5,831 x 2
#>   row_nb binomial
#>   <chr>  <chr>
#> 1 1      Abditomys_latidens
#> 2 2      Abeomelomys_sevia
#> 3 3      Abrawayaomys_ruschii
#> 4 4      Abrocoma_bennettii
#> 5 5      Abrocoma_boliviensis
#> 6 6      Abrocoma_budini
#> # ... with 5,825 more rows
```

3.4.2 Summarise observations with `summarise()`

`mutate()` applies operations to all observations in a table. By contrast, `summarise()` applies operations to *groups* of observations, and returns, er, summaries. The default grouping unit is the entire table:

```
phylacine %>%
  summarise(
    "nb_species" = n(), # counts observations
    "nb_terrestrial" = sum(terrestrial),
    "nb_marine" = sum(marine),
    "nb_freshwater" = sum(freshwater),
    "nb_aerial" = sum(aerial),
    "mean_mass_g" = mean(mass_g)
  )
#> # A tibble: 1 x 6
#>   nb_species nb_terrestrial nb_marine nb_freshwater nb_aerial mean_mass_g
#>        <int>          <dbl>     <dbl>         <dbl>     <dbl>       <dbl>
#> 1       5831           4575       135           156      1162     156882.
```

Above you can see that bats account for a large portion of mammal species diversity (`nb_aerial`). How much exactly? Just as with `mutate()`, you can perform operations on the variables you just created, in the same statement:

```
phylacine %>%
  summarise(
    "nb_species" = n(),
    "nb_aerial" = sum(aerial), # bats
    "prop_aerial" = nb_aerial / nb_species
  )
#> # A tibble: 1 x 3
#>   nb_species nb_aerial prop_aerial
#>        <int>     <dbl>       <dbl>
#> 1       5831      1162       0.199
```

One fifth!

If the british spelling bothers you, `summarize()` exists and is strictly equivalent.

Here's a simple trick with logical (TRUE / FALSE) variables. Their sum is the count of observations that evaluate to `TRUE` (because `TRUE` is taken as 1 and `FALSE` as 0) and their mean is the proportion of `TRUE` observations. This can be exploited to count the number of observations that satisfy a condition:

```
phylacine %>%
  summarise(
    "nb_species" = n(),
    "nb_megafauna" = sum(mass_g > 100000),
    "p_megafauna" = mean(mass_g > 100000)
  )
#> # A tibble: 1 x 3
#>   nb_species nb_megafauna p_megafauna
#>        <int>        <int>       <dbl>
#> 1       5831          302      0.0518
```

There are more summaries that just means and counts (see `?summarise()` for some helpful functions). In fact, summarise can use any function or expression that evaluates to a single value or a *vector* of values. This includes base R `max()`, `quantiles`, etc.

`mutate()` and `transmute()` can compute summaries as well, but they will return the summary once for each observation, in a new column.

```
phylacine %>%
  mutate("nb_species" = n()) %>%
  select(binomial, nb_species)
#> # A tibble: 5,831 x 2
#>   binomial               nb_species
#>   <chr>                       <int>
#> 1 Abditomys_latidens           5831
#> 2 Abeomelomys_sevia            5831
#> 3 Abrawayaomys_ruschii         5831
#> 4 Abrocoma_bennettii           5831
#> 5 Abrocoma_boliviensis         5831
#> 6 Abrocoma_budini              5831
#> # ... with 5,825 more rows
```

3.4.3 Grouping observations by variables

In most cases you don't want to run summary operations on the entire set of observations, but instead on observations that share a common value, i.e. groups. For example, I want to run the summary displayed above, but for each Order of mammals.

`distinct()` extracts all the unique values of a variable

```
phylacine %>% distinct(order)
#> # A tibble: 29 x 1
#>   order
#>   <chr>
#> 1 Rodentia
#> 2 Chiroptera
#> 3 Carnivora
#> 4 Pilosa
#> 5 Diprotodontia
#> 6 Cetartiodactyla
#> # ... with 23 more rows
```

I could work my way with what we have already seen, filtering observations (`filter(order == "Rodentia")`) and then pipeing the output to `summarise()`, and do it again for each Order. But that would be tedious.

Instead, I can use `group_by()` to pool observations by `order`.

```
phylacine %>%
  group_by(order)
#> # A tibble: 5,831 x 24
#> # Groups:   order [29]
#>   binomial order family genus species terrestrial marine freshwater aerial
#>   <chr>    <chr> <chr>  <chr> <chr>         <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abditom~ Rode~ Murid~ Abdi~ latide~           1      0          0      0
#> 2 Abeomel~ Rode~ Murid~ Abeo~ sevia             1      0          0      0
#> 3 Abraway~ Rode~ Crice~ Abra~ ruschii           1      0          0      0
#> 4 Abrocom~ Rode~ Abroc~ Abro~ bennet~           1      0          0      0
#> 5 Abrocom~ Rode~ Abroc~ Abro~ bolivi~           1      0          0      0
#> 6 Abrocom~ Rode~ Abroc~ Abro~ budini            1      0          0      0
#> # ... with 5,825 more rows, and 15 more variables: 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 <chr>, added_iucn_status <chr>,
#> #   diet_plant <dbl>, diet_vertebrate <dbl>, diet_invertebrate <dbl>,
#> #   diet_method <chr>, diet_source <chr>
```

At first glance, nothing has changed, apart from an extra line of information in the output that tells me the observations have been grouped. But now here's what happen if I run the same `summarise()` statement on an ungrouped and a grouped table

```
phylacine %>%
  summarise(
    "n_species" = n(),
    "mean_mass_g" = mean(mass_g)
  )
#> # A tibble: 1 x 2
#>   n_species mean_mass_g
#>       <int>       <dbl>
#> 1      5831     156882.

phylacine %>%
  group_by(order) %>%
  summarise(
    "n_species" = n(),
    "mean_mass_g" = mean(mass_g)
  )
#> # A tibble: 29 x 3
#>   order           n_species mean_mass_g
#>   <chr>               <int>       <dbl>
#> 1 Afrosoricida           57        306.
#> 2 Carnivora             313      47905.
#> 3 Cetartiodactyla       392    1854811.
#> 4 Chiroptera           1162         49.1
#> 5 Cingulata              39     235529.
#> 6 Dasyuromorphia         74        748.
#> # ... with 23 more rows
```

I get one value for each group.

Observations can be grouped by multiple variables, which will output a summary for every unique combination of groups.

```
phylacine %>%
  group_by(order, iucn_status) %>%
  summarise(
    "n_species" = n()
  )
#> # A tibble: 138 x 3
#> # Groups:   order [29]
#>   order        iucn_status n_species
#>   <chr>        <chr>           <int>
#> 1 Afrosoricida CR                  1
#> 2 Afrosoricida DD                  4
#> 3 Afrosoricida EN                  7
#> 4 Afrosoricida EP                  2
#> 5 Afrosoricida LC                 32
#> 6 Afrosoricida NT                  3
#> # ... with 132 more rows
```

Whenever you call `summarise()`, the last level of grouping is dropped. Note how in the output table above, observations are still grouped by order, and no longer by IUCN status. If I summarise observations again:

```
phylacine %>%
  group_by(order, iucn_status) %>%
  summarise(
    "n_species" = n()
  ) %>%
  summarise(
    "n_species_2" = n()
  )
#> # A tibble: 29 x 2
#>   order           n_species_2
#>   <chr>                 <int>
#> 1 Afrosoricida              7
#> 2 Carnivora                 8
#> 3 Cetartiodactyla           9
#> 4 Chiroptera                8
#> 5 Cingulata                 5
#> 6 Dasyuromorphia            7
#> # ... with 23 more rows
```

I get the summary across orders, and the table is no longer grouped at all. This is useful to consider if you need to work on summaries across different levels of the data.

For example, I would like to know how the species in each order are distributed between the different levels of threat in the IUCN classification. To get these proportions, I need to first get the count of each number of species in a level of threat inside an order, and divide that by the number of species in that order.

```
phylacine %>%
  group_by(order, iucn_status) %>%
  summarise("n_order_iucn" = n()) %>%
  # grouping by iucn_status silently dropped
  mutate(
    "n_order" = sum(n_order_iucn),
    "p_iucn" = n_order_iucn / n_order
  )
#> # A tibble: 138 x 5
#> # Groups:   order [29]
#>   order        iucn_status n_order_iucn n_order p_iucn
#>   <chr>        <chr>              <int>   <int>  <dbl>
#> 1 Afrosoricida CR                     1      57 0.0175
#> 2 Afrosoricida DD                     4      57 0.0702
#> 3 Afrosoricida EN                     7      57 0.123
#> 4 Afrosoricida EP                     2      57 0.0351
#> 5 Afrosoricida LC                    32      57 0.561
#> 6 Afrosoricida NT                     3      57 0.0526
#> # ... with 132 more rows
```

10.2% of Carnivores are Endangered (“EN”).

3.4.4 Grouped data and other `dplyr` verbs

Grouping does not only affect the behaviour of `summarise`, but under circumstances, other verbs can (and will!) perform operations by groups.

```
# Species with a higher mass than the mammal mean
phylacine %>%
  select("binomial", "mass_g") %>%
  filter(mass_g > mean(mass_g, na.rm = TRUE))
#> # A tibble: 234 x 2
#>   binomial                      mass_g
#>   <chr>                          <dbl>
#> 1 Alcelaphus_buselaphus        171002.
#> 2 Alces_alces                  356998
#> 3 Archaeoindris_fontoynonti    160000
#> 4 Arctocephalus_pusillus       178500
#> 5 Arctodus_simus               709500
#> 6 Balaena_mysticetus        100000000
#> # ... with 228 more rows
```

```
# Species with a higher mass than the mean in their order
phylacine %>%
  group_by(order) %>%
  select("binomial", "mass_g") %>%
  filter(mass_g > mean(mass_g, na.rm = TRUE))
#> # A tibble: 890 x 3
#> # Groups:   order [27]
#>   order      binomial              mass_g
#>   <chr>      <chr>                  <dbl>
#> 1 Chiroptera Acerodon_celebensis     390
#> 2 Chiroptera Acerodon_humilis        600.
#> 3 Chiroptera Acerodon_jubatus       1075
#> 4 Chiroptera Acerodon_leucotis       513.
#> 5 Chiroptera Acerodon_mackloti       470.
#> 6 Rodentia   Aeretes_melanopterus    732.
#> # ... with 884 more rows


# Largest mammal
phylacine %>%
  select(binomial, mass_g) %>%
  slice_max(mass_g)
#> # A tibble: 1 x 2
#>   binomial                   mass_g
#>   <chr>                       <dbl>
#> 1 Balaenoptera_musculus 190000000
# Largest species in each order
phylacine %>%
  group_by(order) %>%
  select(binomial, mass_g) %>%
  slice_max(mass_g)
#> # A tibble: 30 x 3
#> # Groups:   order [29]
#>   order           binomial                             mass_g
#>   <chr>           <chr>                                 <dbl>
#> 1 Afrosoricida    Plesiorycteropus_madagascariensis     13220
#> 2 Carnivora       Mirounga_leonina                    1600000
#> 3 Cetartiodactyla Balaenoptera_musculus             190000000
#> 4 Chiroptera      Acerodon_jubatus                       1075
#> 5 Cingulata       Glyptodon_clavipes                  2000000
#> 6 Dasyuromorphia  Thylacinus_cynocephalus               30000
#> # ... with 24 more rows
```

To avoid grouped operations, you can simply drop grouping with `ungroup()`.

3.5 Working with multiple tables

3.5.1 Binding tables

`dplyr` introduces `bind_rows()` and `bind_cols()`, which are equivalent to base R `rbind()` and `cbind()`, with a few extra feature. They are faster, and can bind many tables at once, and bind data frames with vectors or lists.

`bind_rows()` has an option to pass a variable specifying which dataset each observation originates from.

```
porpoises <- phylacine %>%
  filter(family == "Phocoenidae") %>%
  select(binomial, iucn_status)
echidnas <- phylacine %>%
  filter(family == "Tachyglossidae") %>%
  select(binomial, iucn_status)

bind_rows(
  "porpoise" = porpoises,
  "echidna" = echidnas,
  .id = "kind"
)
#> # A tibble: 13 x 3
#>   kind     binomial                     iucn_status
#>   <chr>    <chr>                        <chr>
#> 1 porpoise Neophocaena_asiaeorientalis VU
#> 2 porpoise Neophocaena_phocaenoides    VU
#> 3 porpoise Phocoena_dioptrica          DD
#> 4 porpoise Phocoena_phocoena           LC
#> 5 porpoise Phocoena_sinus              CR
#> 6 porpoise Phocoena_spinipinnis        DD
#> # ... with 7 more rows
```

3.5.2 Combining variables of two tables with mutating joins

Mutating joins are tailored to combine tables that share a set of observations but have different variables.

As an example, let's split the `phylacine` dataset in two smaller datasets, one containing information on diet and one on the dominant habitat.

```
diet <- phylacine %>%
  select(binomial, diet_plant:diet_invertebrate) %>%
  slice(1:5)
```

```
diet
#> # A tibble: 5 x 4
#>   binomial               diet_plant diet_vertebrate diet_invertebrate
#>   <chr>                       <dbl>           <dbl>             <dbl>
#> 1 Abditomys_latidens            100               0                 0
#> 2 Abeomelomys_sevia              78               3                19
#> 3 Abrawayaomys_ruschii           88               1                11
#> 4 Abrocoma_bennettii            100               0                 0
#> 5 Abrocoma_boliviensis          100               0                 0

life_habit <- phylacine %>% select(binomial, terrestrial:aerial) %>%
  slice(1:3, 6:7)
life_habit
#> # A tibble: 5 x 5
#>   binomial               terrestrial marine freshwater aerial
#>   <chr>                        <dbl>  <dbl>      <dbl>  <dbl>
#> 1 Abditomys_latidens               1      0          0      0
#> 2 Abeomelomys_sevia                1      0          0      0
#> 3 Abrawayaomys_ruschii             1      0          0      0
#> 4 Abrocoma_budini                  1      0          0      0
#> 5 Abrocoma_cinerea                 1      0          0      0
```

The two datasets each contain 5 species, the first three are shared, and the two last differ between the two.

```
intersect(diet$binomial, life_habit$binomial)
#> [1] "Abditomys_latidens"   "Abeomelomys_sevia"    "Abrawayaomys_ruschii"
setdiff(diet$binomial, life_habit$binomial)
#> [1] "Abrocoma_bennettii"   "Abrocoma_boliviensis"
```

To use mutate-joins, both tables need to have a **key**, a variable that identifies each observation. Here, that would be `binomial`, the sepcies names. If your table doesn't have such a key and the rows between the tables match one another, remember you can create a row number variable easily with `tibble::column_to_rownames()`.

```
inner_join(diet, life_habit, by = "binomial")
#> # A tibble: 3 x 8
#>   binomial diet_plant diet_vertebrate diet_invertebra~ terrestrial marine
#>   <chr>         <dbl>           <dbl>            <dbl>       <dbl>  <dbl>
#> 1 Abditom~        100               0                0           1      0
#> 2 Abeomel~         78               3               19           1      0
#> 3 Abraway~         88               1               11           1      0
#> # ... with 2 more variables: freshwater <dbl>, aerial <dbl>
```

`inner_join` combined the variables, and dropped the observations that weren't matched between the two tables. There are three other variations of mutating joins, differing in what they do with unmatching variables.

```
left_join(diet, life_habit, by = "binomial")
#> # A tibble: 5 x 8
#>   binomial diet_plant diet_vertebrate diet_invertebra~ terrestrial marine
#>   <chr>         <dbl>           <dbl>            <dbl>       <dbl>  <dbl>
#> 1 Abditom~        100               0                0           1      0
#> 2 Abeomel~         78               3               19           1      0
#> 3 Abraway~         88               1               11           1      0
#> 4 Abrocom~        100               0                0          NA     NA
#> 5 Abrocom~        100               0                0          NA     NA
#> # ... with 2 more variables: freshwater <dbl>, aerial <dbl>
right_join(diet, life_habit, by = "binomial")
#> # A tibble: 5 x 8
#>   binomial diet_plant diet_vertebrate diet_invertebra~ terrestrial marine
#>   <chr>         <dbl>           <dbl>            <dbl>       <dbl>  <dbl>
#> 1 Abditom~        100               0                0           1      0
#> 2 Abeomel~         78               3               19           1      0
#> 3 Abraway~         88               1               11           1      0
#> 4 Abrocom~         NA              NA               NA           1      0
#> 5 Abrocom~         NA              NA               NA           1      0
#> # ... with 2 more variables: freshwater <dbl>, aerial <dbl>
full_join(diet, life_habit, by = "binomial")
#> # A tibble: 7 x 8
#>   binomial diet_plant diet_vertebrate diet_invertebra~ terrestrial marine
#>   <chr>         <dbl>           <dbl>            <dbl>       <dbl>  <dbl>
#> 1 Abditom~        100               0                0           1      0
#> 2 Abeomel~         78               3               19           1      0
#> 3 Abraway~         88               1               11           1      0
#> 4 Abrocom~        100               0                0          NA     NA
#> 5 Abrocom~        100               0                0          NA     NA
#> 6 Abrocom~         NA              NA               NA           1      0
#> # ... with 1 more row, and 2 more variables: freshwater <dbl>, aerial <dbl>

semi_join(diet, life_habit, by  = "binomial")
#> # A tibble: 3 x 4
#>   binomial             diet_plant diet_vertebrate diet_invertebrate
#>   <chr>                     <dbl>           <dbl>             <dbl>
#> 1 Abditomys_latidens          100               0                 0
#> 2 Abeomelomys_sevia            78               3                19
#> 3 Abrawayaomys_ruschii         88               1                11
anti_join(diet, life_habit, by  = "binomial")
#> # A tibble: 2 x 4
#>   binomial             diet_plant diet_vertebrate diet_invertebrate
#>   <chr>                     <dbl>           <dbl>             <dbl>
#> 1 Abrocoma_bennettii          100               0                 0
#> 2 Abrocoma_boliviensis        100               0                 0
```

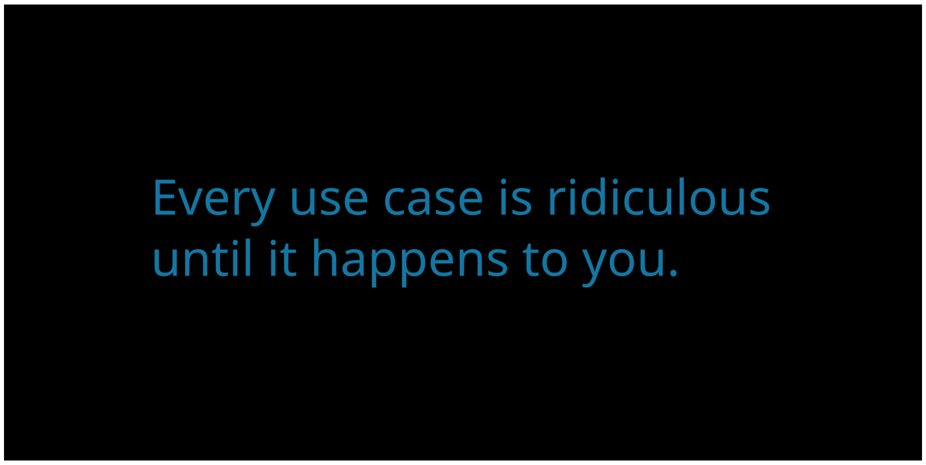
3.5.3 Filtering matching observations between two tables wiht filtering joins

So-called filtering joins return row from the first table that are matched (or not, for `anti_join()`) in the second.

```
semi_join(diet, life_habit, by  = "binomial")
#> # A tibble: 3 x 4
#>   binomial              diet_plant diet_vertebrate diet_invertebrate
#>   <chr>                      <dbl>           <dbl>             <dbl>
#> 1 Abditomys_latidens           100               0                 0
#> 2 Abeomelomys_sevia             78               3                19
#> 3 Abrawayaomys_ruschii          88               1                11
anti_join(diet, life_habit, by  = "binomial")
#> # A tibble: 2 x 4
#>   binomial              diet_plant diet_vertebrate diet_invertebrate
#>   <chr>                      <dbl>           <dbl>             <dbl>
#> 1 Abrocoma_bennettii           100               0                 0
#> 2 Abrocoma_boliviensis         100               0                 0
```

Chapter 4

Working with lists and iteration



Every use case is ridiculous until it happens to you.

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

4.1 List columns with `tidyr`

4.1.1 Nesting data

It may become necessary to indicate the groups of a tibble in a somewhat more explicit way than simply using `dplyr::group_by`. `tidyr` offers the option to create nested tibbles, that is, to store complex objects in the columns of a tibble. This includes other tibbles, as well as model objects and plots.

NB: Nesting data is done using `tidyr::nest`, which is different from the similarly named `tidyr::nesting`.

The example below shows how `mtcars` can be converted into a nested tibble.

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

nested_cars
#> # A tibble: 3 x 2
#> # Groups:   cyl [3]
#>     cyl data
#>   <dbl> <list>
#> 1     6 <tibble [7 x 11]>
#> 2     4 <tibble [11 x 11]>
#> 3     8 <tibble [14 x 11]>

# get column class
sapply(nested_cars, class)
#>       cyl      data
#> "numeric"    "list"
```

`mtcars` is now a nested data frame. The class of each of its columns is respectively, a numeric (number of cylinders) and a list (the data of all cars with as many cylinders as in the corresponding row).

While `nest` can be used without first grouping the tibble, it's just much easier to group first.

4.1.2 Unnesting data

A nested tibble can be converted back into the original, or into a processed form, using `tidyr::unnest`. The original groups are retained.

```
# use unnest to recover the original data frame
unnest(nested_cars, cols = "data")
#> # A tibble: 32 x 12
#> # Groups:   cyl [3]
#>     cyl car_name        mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <dbl> <chr>         <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     6 Mazda RX4      21     160   110  3.9   2.62  16.5     0     1     4     4
#> 2     6 Mazda RX4 W~   21     160   110  3.9   2.88  17.0     0     1     4     4
#> 3      6 Hornet 4 Dr~  21.4  258    110  3.08  3.22  19.4      1     0     3     1
#> 4      6 Valiant       18.1  225    105  2.76  3.46  20.2      1     0     3     1
#> 5      6 Merc 280      19.2  168.   123  3.92  3.44  18.3      1     0     4     4
#> 6      6 Merc 280C     17.8  168.   123  3.92  3.44  18.9      1     0     4     4
#> # ... with 26 more rows

# unnesting preserves groups
groups(unnest(nested_cars, cols = "data"))
#> [[1]]
#> cyl
```

The `unnest_longer` and `unnest_wider` variants of `unnest` are maturing functions, that is, not in their final form. They allow interesting variations on unnesting — these are shown here but advised against.

Unnest the data first, and then convert it to the form needed.

```
unnest_longer(nested_cars, col = "data") %>%
  head()
#> # A tibble: 6 x 2
#> # Groups:   cyl [1]
#>      cyl data$car_name  $mpg $disp   $hp $drat   $wt $qsec   $vs   $am $gear
#>    <dbl> <chr>         <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1      6 Mazda RX4      21    160    110  3.9   2.62  16.5     0     1     4
#> 2      6 Mazda RX4 Wag  21    160    110  3.9   2.88  17.0     0     1     4
#> 3      6 Hornet 4 Dri~  21.4  258    110  3.08  3.22  19.4     1     0     3
#> 4      6 Valiant        18.1  225    105  2.76  3.46  20.2     1     0     3
#> 5      6 Merc 280       19.2  168.   123  3.92  3.44  18.3     1     0     4
#> 6      6 Merc 280C      17.8  168.   123  3.92  3.44  18.9     1     0     4
#> # ... with 1 more variable: $carb <dbl>

unnest_wider(nested_cars, col = "data")
#> # A tibble: 3 x 12
#> # Groups:   cyl [3]
#>      cyl car_name  mpg    disp   hp     drat  wt    qsec  vs    am    gear  carb
#>    <dbl> <list>    <list> <list> <list> <lis> <lis> <lis> <lis> <lis> <lis> <lis>
#> 1      6 <chr [7]> <dbl ~ <dbl ~ <dbl ~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~
#> 2      4 <chr [11~ <dbl ~ <dbl ~ <dbl ~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~
#> 3      8 <chr [14~ <dbl ~ <dbl ~ <dbl ~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~ <dbl~
```

4.1.3 Working with list columns

The class of a list column is `list`, and working with list columns (and lists, and list-like objects such as vectors) makes iteration necessary, since this is one of the only ways to operate on lists.

Two examples are shown below when getting the class and number of rows of the nested tibbles in the list column.

```
# how many rows in each nested tibble?
for (i in seq_along(nested_cars$data)) {
  print(nrow(nested_cars$data[[i]]))
}
#> [1] 7
#> [1] 11
#> [1] 14

# what is the class of each element?
lapply(X = nested_cars$data, FUN = class)
#> [[1]]
#> [1] "tbl_df"     "tbl"        "data.frame"
#>
#> [[2]]
#> [1] "tbl_df"     "tbl"        "data.frame"
#>
#> [[3]]
#> [1] "tbl_df"     "tbl"        "data.frame"
```

Functionals

The second example uses `lapply`, and this is a *functional*. *Functionals* are functions that take another function as one of their arguments. Base `R` functionals include the `*apply` family of functions: `apply`, `lapply`, `vapply` and so on.

4.2 Iteration with `map`

The `tidyverse` replaces traditional loop-based iteration with *functionals* from the `purrr` package. A good reason to use `purrr` functionals instead of base `R` functionals is their consistent and clear naming, which always indicates how they should be used. This is explained in the examples below.

How `map` is different from `for` and `lapply` are best explained in the **Advanced R Book**.

4.2.1 Basic use of `map`

`map` works very similarly to `lapply`, where `.x` is object on whose elements to apply the function `.f`.

```
# get the number of rows in data
map(.x = nested_cars$data, .f = nrow)
#> [[1]]
#> [1] 7
#>
#> [[2]]
#> [1] 11
#>
#> [[3]]
#> [1] 14
```

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

```
# get the square root of each integer 1 - 10
some_numbers = 1:3
map(some_numbers, sqrt)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 1.41
#>
#> [[3]]
#> [1] 1.73
```

4.2.2 `map` variants returning vectors

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

```
# use map_dbl to get a vector of square roots
some_numbers = 1:10
map_dbl(some_numbers, sqrt)
#>  [1] 1.00 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_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
```

4.2.3 `map` variants returning data frames

`map_df` returns data frames, and by default binds dataframes by rows, while `map_dfr` does 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
#> Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4    1
#> Merc 240D         24.4   4  147  62 3.69 3.19 20.0  1  0    4    2
#> Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4    4
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3    2
#> Duster 360        14.3   8  360 245 3.21 3.57 15.8  0  0    3    4
```

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

`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
#> Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4    1
#> Merc 240D         24.4   4  147  62 3.69 3.19 20.0  1  0    4    2
#> Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4    4
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3    2
#> Duster 360        14.3   8  360 245 3.21 3.57 15.8  0  0    3    4
```

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

```
some_list %>%
  map_dfc(head, n = 2)
#>            mpg...1 cyl...2 disp...3 hp...4 drat...5 wt...6 qsec...7 vs...8
#> Datsun 710    22.8       4      108     93     3.85   2.32     18.6      1
#> Merc 240D     24.4       4      147     62     3.69   3.19     20.0      1
#>            am...9 gear...10 carb...11 mpg...12 cyl...13 disp...14 hp...15
#> Datsun 710      1         4         1       21        6       160     110
#> Merc 240D       0         4         2       21        6       160     110
#>             drat...16 wt...17 qsec...18 vs...19 am...20 gear...21 carb...22
#> Datsun 710        3.9    2.62      16.5       0       1         4         4
#> Merc 240D         3.9    2.88      17.0       0       1         4         4
#>             mpg...23 cyl...24 disp...25 hp...26 drat...27 wt...28 qsec...29
#> Datsun 710      18.7        8       360     175      3.15    3.44      17.0
#> Merc 240D       14.3        8       360     245      3.21    3.57      15.8
#>             vs...30 am...31 gear...32 carb...33
#> Datsun 710        0       0         3         2
#> Merc 240D         0       0         3         4
```

806 4.2.4 Working with list columns using `map`

The various `map` versions integrate well with list columns to make synthetic/summary data. In the example, the `dplyr::mutate` function is used to add three columns to the nested tibble: the number of rows, the mean mileage, and the name of the first car.

In each of these cases, the vectors added are generated using `purrr` functions.

```
# get the number of rows per dataframe, the mean mileage, and the first car
nested_cars = nested_cars %>%
  mutate(
    # use the int return to get the number of rows
    n_rows = map_int(data, nrow),

    # double return for mean mileage
    mean_mpg = map_dbl(data, function(df) {mean(df$mpg)}),

    # character return to get first car
    first_car = map_chr(data, function(df) {first(df$car_name)}
    )
  )

# examine the output
nested_cars
#> # 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
```

4.2.5 Selective mapping using `map` variants

`map_at` and `map_if` work like other `*_at` and `*_if` functions. Here, `map_if` is used to run a linear model only on those tibbles which have sufficient data. The predicate is specified by `.p`.

In this example, the nested tibble is given a new column using `dplyr::mutate`, where the data to be added is a mixed list.

```r
# 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>
```

The first element is a tibble of the corresponding element in `mtcars$cars`, which has not been operated on because it has fewer than 10 rows. The remaining elements are `lm` objects.

4.3 More `map` variants

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

4.3.1 Mapping over two inputs with map2

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

map2 doesn't have _at and _if variants.

One use case for map2 is to deal with both a list element and its index, as shown in the example. This may be necessary when the list index is removed in a split or nest. This 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
```

4.3.2 Mapping over multiple inputs with pmap

pmap instead operates on a list of multiple list-like objects, and also comes with the same 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 #FF0000" "number 2, letter b, colour #00FF00"
#> [3] "number 3, letter c, colour #0000FF"
```

4.3.3 Mapping at depth

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

In the example, `mtcars` is split by cylinders, and then by gears, creating a two-level list, 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
```

4.3.4 Iteration without a return

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

`walk` is the function for this task. It has only the variants `walk2`, `iwalk`, and `pwalk`, whose logic is similar to `map2`, `imap`, and `pmap`. In the example, the function applied to each list 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'))
     })
```

4.3.5 Modify rather than map

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

In the example, simply adding 2 to each vector element produces an error, because the 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"
```

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
```

A note on `invoke`

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

4.4 Other functions for working with lists

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

4.4.1 Filtering lists

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

`head_while` is bit of an odd case, which returns all elements of a list-like object in sequence 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
```

4.4.2 Summarising lists

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

4.4.3 Reduction and accumulation

`reduce` helps combine elements along a list using a specific function. Consider the example 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
```

This can also be applied to data frames. Consider some random samples of `mtcars`, each with only 5 cars removed. The objective is to find the cars present in all 10 samples.

The way `reduce` works in the example below is to take the first element and find its intersection with the second, and to take the result and find its intersection with the third and so on.

```
# sample mtcars
mtcars = as_tibble(mtcars, rownames = "car")
sampled_data = map(1:10, function(x){sample_n(mtcars, nrow(mtcars)-5)})

# get cars which appear in all samples
sampled_data = reduce(sampled_data, dplyr::inner_join)
```

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

4.4.4 Miscellaneous operation

`purrr` offers a few more functions to work with lists (or list like objects). `prepend` works very similarly to `append`, except it adds to the head of a list. `splice` adds multiple objects 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"
```

`flatten` has a similar behaviour, and converts a list of vectors or list of lists to a single 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"
```

`transpose` shifts the index order in multi-level lists. This is seen in the example, where 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`
#> # A tibble: 1 x 12
#>   car              mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <chr>          <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Toyota Coro~    21.5     4  120.    97   3.7  2.46  20.0     1     0     3     1
#>
#>
#> $`4`
#> $`4`$`4`
#> # A tibble: 8 x 12
#>   car              mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <chr>          <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Datsun 710      22.8     4 108      93  3.85  2.32  18.6     1     1     4     1
#> 2 Merc 240D       24.4     4 147.     62  3.69  3.19  20       1     0     4     2
#> 3 Merc 230        22.8     4 141.     95  3.92  3.15  22.9     1     0     4     2
#> 4 Fiat 128        32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
#> 5 Honda Civic     30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
#> 6 Toyota Coro~    33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1
#> # ... with 2 more rows
#>
#>
```

```
#> $`5`
#> $`5`$`4`
#> # A tibble: 2 x 12
#>   car              mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <chr>          <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Porsche 914~  26        4 120.      91  4.43  2.14  16.7     0     1     5     2
#> 2 Lotus Europa  30.4      4  95.1    113  3.77  1.51  16.9     1     1     5     2
```

4.5 To add: patchwork

4.5.0.1 Final words

In general, an iteration based problem can usually be solved with `purrr`.