Programming in the tidyverse

Kirill Müller, Tobias Schieferdecker

29 May 2019, 21:54 CEST

Contents

Preface **5**
Links . . . 5
Package versions used . . . 5
License . . . 7

1 Introduction **9**
1.1 Overview . . . 9
1.2 Review of visualization and data transformation . . . 11

2 Function basics **21**
2.1 Definition and execution . . . 21
2.2 Arguments . . . 24
2.3 Use case: Intermediate variables . . . 25
2.4 Default values . . . 27
2.5 Multiple arguments . . . 30
2.6 Argument matching . . . 33

3 Simple iteration **39**
3.1 Vectors and columns . . . 39
3.2 Named vectors and two-column tibbles . . . 41
3.3 Indexing/subsetting . . . 45
3.4 Construction . . . 47
3.5 Processing multiple files . . . 50
3.6 Manipulating all datasets . . . 57
3.7 Typed output . . . 66

4 Pairwise iteration and nesting **69**
4.1 Manipulating pairwise . . . 70
4.2 Moving to tibble-land . . . 73
4.3 Nesting and unnesting . . . 76

5 Scoping and flow control **81**
5.1 Scope . . . 81
5.2 Pure functions and side effects . . . 82

5.3 Control flow 84
5.4 Closures 88

6 Non-rectangular data **97**
6.1 Traversing 97
6.2 Iterating and traversing 106
6.3 Plucking multiple locations 110
6.4 Flattening 113
6.5 Transposing 114
6.6 Rectangling 116
6.7 Accessing APIs 117

7 Tidy evaluation **125**
7.1 A custom plotting function 125
7.2 Do you need tidy evaluation? 131
7.3 Explicit quote-unquote of ellipsis 132
7.4 Names 133
7.5 Debugging 134
7.6 Argument names 135
7.7 purrr-style mappers 136

8 Best practices **139**
8.1 DESCRIPTION 139
8.2 R 140
8.3 roxygen2 141
8.4 testthat 141

9 **143**

Preface

Material for the zhRcourse workshop "Programming in the tidyverse" on May 10, 2019.

See the controls at the top of the website for searching, font size, editing, and a link to the PDF version of the material.

Links

- This website: https://bit.ly/tidyprog
  - Longer URL: https://krlmlr.github.io/tidyprog/
- Scripts and installation instructions: https://github.com/krlmlr/tidyprog-proj/tree/2019-05-zhr
  - Prepared scripts: https://github.com/krlmlr/tidyprog-proj/tree/2019-05-zhr/script
  - Live code: https://github.com/krlmlr/tidyprog-proj/tree/2019-05-zhr/live
  - The code **will be updated live** with a delay of a few seconds during the workshop, it is **not necessary** to repeat the instructor's typing
- rstudio.cloud server: https://rstudio.cloud/project/329883
  - Sign up, or log in with Google or GitHub
  - Click the "Save a private copy" link next to the **TEMPORARY** label in red in the header
  - All necessary packages are preinstalled
- The source project for this material: https://github.com/krlmlr/tidyprog

Package versions used

Click to expand

```
withr::with_options(list(width = 80), print(sessioninfo::session_info()))
```

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 3.6.0 (2017-01-27)
##  os       Ubuntu 16.04.6 LTS
##  system   x86_64, linux-gnu
##  ui       X11
##  language en_US.UTF-8
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       UTC
##  date     2019-05-29
##
## - Packages -------------------------------------------------------------------
##  package     * version     date       lib source
##  assertthat    0.2.1       2019-03-21 [1] CRAN (R 3.6.0)
##  backports     1.1.4       2019-04-10 [1] CRAN (R 3.6.0)
##  bookdown      0.9         2018-12-21 [1] CRAN (R 3.6.0)
##  broom         0.5.2       2019-04-07 [1] CRAN (R 3.6.0)
##  cellranger    1.1.0       2016-07-27 [1] CRAN (R 3.6.0)
##  cli           1.1.0       2019-03-19 [1] CRAN (R 3.6.0)
##  codetools     0.2-16      2018-12-24 [3] CRAN (R 3.6.0)
##  colorspace    1.4-1       2019-03-18 [1] CRAN (R 3.6.0)
##  crayon        1.3.4       2017-09-16 [1] CRAN (R 3.6.0)
##  digest        0.6.19      2019-05-20 [1] CRAN (R 3.6.0)
##  dplyr       * 0.8.1       2019-05-14 [1] CRAN (R 3.6.0)
##  evaluate      0.14        2019-05-28 [1] CRAN (R 3.6.0)
##  forcats     * 0.4.0       2019-02-17 [1] CRAN (R 3.6.0)
##  generics      0.0.2       2018-11-29 [1] CRAN (R 3.6.0)
##  ggplot2     * 3.1.1       2019-04-07 [1] CRAN (R 3.6.0)
##  glue          1.3.1       2019-03-12 [1] CRAN (R 3.6.0)
##  gtable        0.3.0       2019-03-25 [1] CRAN (R 3.6.0)
##  haven         2.1.0       2019-02-19 [1] CRAN (R 3.6.0)
##  here        * 0.1         2017-05-28 [1] CRAN (R 3.6.0)
##  hms           0.4.2       2018-03-10 [1] CRAN (R 3.6.0)
##  htmltools     0.3.6       2017-04-28 [1] CRAN (R 3.6.0)
##  httr          1.4.0       2018-12-11 [1] CRAN (R 3.6.0)
##  jsonlite      1.6         2018-12-07 [1] CRAN (R 3.6.0)
##  knitr         1.23        2019-05-18 [1] CRAN (R 3.6.0)
##  lattice       0.20-38     2018-11-04 [3] CRAN (R 3.6.0)
##  lazyeval      0.2.2       2019-03-15 [1] CRAN (R 3.6.0)
##  lubridate     1.7.4       2018-04-11 [1] CRAN (R 3.6.0)
##  magrittr      1.5         2014-11-22 [1] CRAN (R 3.6.0)
##  memoise       1.1.0       2017-04-21 [1] CRAN (R 3.6.0)
##  modelr        0.1.4       2019-02-18 [1] CRAN (R 3.6.0)
##  munsell       0.5.0       2018-06-12 [1] CRAN (R 3.6.0)
##  nlme          3.1-139     2019-04-09 [3] CRAN (R 3.6.0)
```

```
##  pillar        1.4.1       2019-05-28 [1] CRAN (R 3.6.0)
##  pkgconfig     2.0.2       2018-08-16 [1] CRAN (R 3.6.0)
##  plyr          1.8.4       2016-06-08 [1] CRAN (R 3.6.0)
##  purrr       * 0.3.2       2019-03-15 [1] CRAN (R 3.6.0)
##  R6            2.4.0       2019-02-14 [1] CRAN (R 3.6.0)
##  Rcpp          1.0.1       2019-03-17 [1] CRAN (R 3.6.0)
##  readr       * 1.3.1       2018-12-21 [1] CRAN (R 3.6.0)
##  readxl        1.3.1       2019-03-13 [1] CRAN (R 3.6.0)
##  rlang         0.3.4       2019-04-07 [1] CRAN (R 3.6.0)
##  rmarkdown     1.13        2019-05-22 [1] CRAN (R 3.6.0)
##  rprojroot     1.3-2       2018-01-03 [1] CRAN (R 3.6.0)
##  rstudioapi    0.10        2019-03-19 [1] CRAN (R 3.6.0)
##  rvest         0.3.4       2019-05-15 [1] CRAN (R 3.6.0)
##  scales        1.0.0       2018-08-09 [1] CRAN (R 3.6.0)
##  sessioninfo   1.1.1       2018-11-05 [1] CRAN (R 3.6.0)
##  stringi       1.4.3       2019-03-12 [1] CRAN (R 3.6.0)
##  stringr     * 1.4.0       2019-02-10 [1] CRAN (R 3.6.0)
##  tibble      * 2.1.1       2019-03-16 [1] CRAN (R 3.6.0)
##  tic           0.2.13.9016 2019-05-29 [1] Github (ropenscilabs/tic@31cd3db)
##  tidyr       * 0.8.3       2019-03-01 [1] CRAN (R 3.6.0)
##  tidyselect    0.2.5       2018-10-11 [1] CRAN (R 3.6.0)
##  tidyverse   * 1.2.1       2017-11-14 [1] CRAN (R 3.6.0)
##  withr         2.1.2       2018-03-15 [1] CRAN (R 3.6.0)
##  xfun          0.7         2019-05-14 [1] CRAN (R 3.6.0)
##  xml2          1.2.0       2018-01-24 [1] CRAN (R 3.6.0)
##  yaml          2.2.0       2018-07-25 [1] CRAN (R 3.6.0)
##
## [1] /home/travis/R/Library
## [2] /usr/local/lib/R/site-library
## [3] /home/travis/R-bin/lib/R/library
```

License

Chapter 1

Introduction

The `tidyverse` has quickly developed over the last years. Its first implementation as a collection of partly older packages was in the second half of 2016. All its packages "share an underlying design philosophy, grammar, and data structures."[1] It is for sure difficult to tell, if "learning the `tidyverse`" is a hard task, since the result of this assessment might differ from person to person. We do believe though, that there are concepts in its approach, which – when grasped – have the potential to increase one's productivity, since code creation will seem more natural. While this might be true for all languages (once you speak it well enough, things go smoothly), in our opinion the `tidyverse` worth exploring in depth, since it is

1. consistent: an especially well designed framework that aims at making data analysis and programming intuitive,
2. evolving: constantly deepened understanding for challenges arising in modern data analysis leads to improving ergonomic user interfaces.

This section gives a brief overview, introduces the data used for the course, and offers a refresher for tidy data manipulation and visualization.

1.1 Overview

This course covers several topics, which everyone working more intently with the `tidyverse` almost inevitably needs to deal with at some point or another. The topics are organized in chapters that contain mostly R code with output and text. In each section, exercises are provided.

Each subsection corresponds to an R script in the `script` directory in the sister repository on GitHub. For example, the code from the next section 1.2

[1] citation from tidyverse homepage

can be found in `12-intro.R`. Clone or download the repository and open the `R-workshop.Rproj` file to run the script. (It is important to open the `.Rproj` file and not only the `.R` scripts.)

1. Function basics

   > structuring the code to avoid too much copy-pasting

   Using functions to structure code. This part is independent of the subsequent section.

   - We begin with how to define and execute a function
   - Discussion of a function's arguments (from both the developers' and the users' perspective)
   - A few words on function design

2. Simple iteration

   > processing multiple files that contain different parts of the same dataset

   This part introduces iteration and is independent of the previous section.

   - How to get from a list or a vector to a tibble and vice-versa
   - Indexing for vectors and lists
   - Applying a function to each element of a list or a vector

3. Pairwise iteration and nesting

   More advanced iteration.

   - Simultaneously feed two or more separate lists of inputs into a function working with those two arguments
   - Iterate rowwise through columns in a tibble
   - Nested tibbles, a very powerful concept

4. Scoping and flow control

   More advanced functional concepts.

   - Data lifecycle
   - Purity
   - Control flow
   - Metaprogramming

5. Non-rectangular data

   > working with raw data from online services (JSON)

   Processing hierarchical lists as commonly returned from web APIs.

   - Data lifecycle
   - Purity
   - Control flow
   - Metaprogramming

6. Tidy evaluation

   writing functions that work with datasets of different shape

   TBD

1.2 Review of visualization and data transformation

This section is a refresher for visualization and data transformation in the tidyverse. Readers familiar with the first half or R for data science will recognize the concepts repeated here. The data used throughout this course is presented, plotted and briefly analyzed.

The code in each chapter is self-contained. The code in each section is also self-contained, but the necessary setup code is hidden and can be expanded with a click. We will always load the following packages:

```
library(tidyverse)
library(here)
```

Functions from other packages may be used with the `::` notation.

1.2.1 Data

We will be working with hourly measurements of weather data (link to data documentation) in four cities (Berlin, Toronto, Tel Aviv and Zurich) between 2019-04-28, 3pm and 2019-04-30, 3pm. Thus we have 49 observations in each city. Variables are:

- `time`
- `summary` (how to describe the weather in one word)
- `icon` (mix of description of weather plus time of day)
- `precipIntensity` (intensity of precipitation [mm/h])
- `precipProbability`
- `temperature`
- `apparentTemperature`
- `dewPoint`
- `humidity`
- `pressure`
- `windSpeed`
- `windGust`
- `windBearing` (direction in degrees)
- `cloudCover`
- `uvIndex`

- visibility
- ozone
- precipType

Reading in the data, which is stored in MS Excel-Files:

```
berlin <- readxl::read_excel(here("data/weather", "berlin.xlsx"))
toronto <- readxl::read_excel(here("data/weather", "toronto.xlsx"))
tel_aviv <- readxl::read_excel(here("data/weather", "tel_aviv.xlsx"))
zurich <- readxl::read_excel(here("data/weather", "zurich.xlsx"))
```

Create one larger tibble from the four smaller ones:

```
weather_data <- bind_rows(
  berlin = berlin,
  toronto = toronto,
  tel_aviv = tel_aviv,
  zurich = zurich,
  .id = "city_code"
)
```

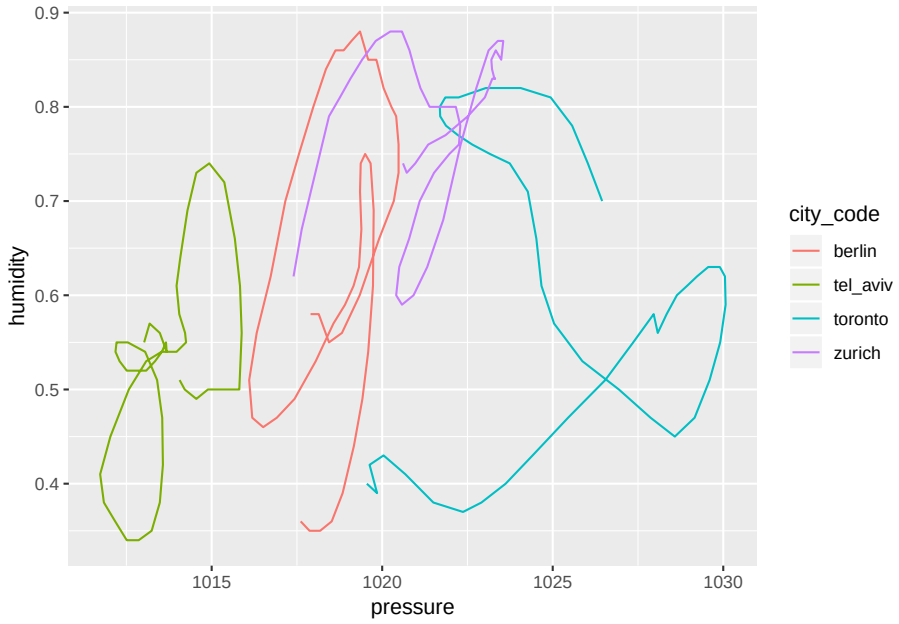
1.2.2 Exploration

```
weather_data
```

```
## # A tibble: 196 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 193 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

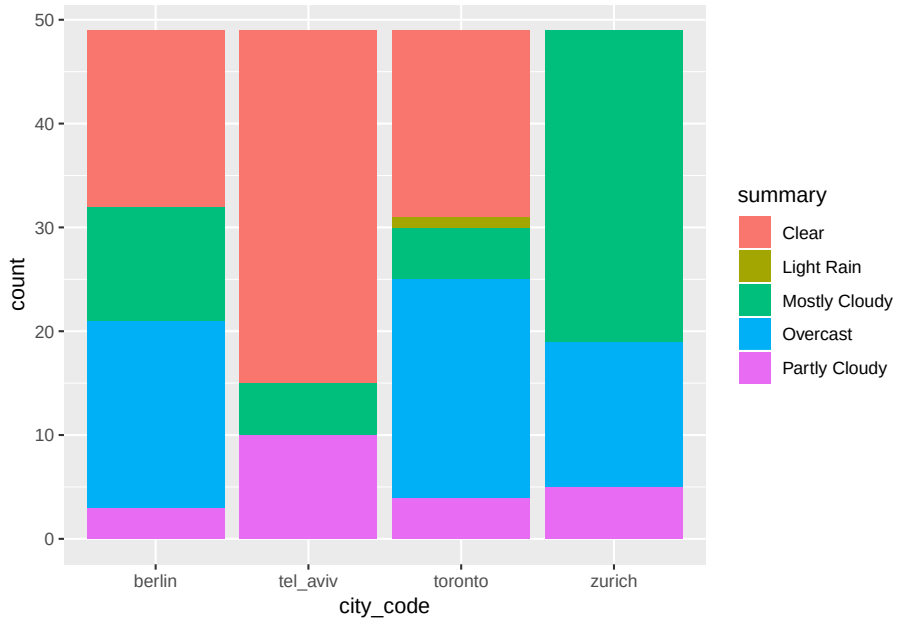
Example plot of humidity vs. pressure (`geom_path()` ensures that points are connected according to their order in the tibble):

```
weather_data %>%
  ggplot(aes(x = pressure, y = humidity, color = city_code)) +
  geom_path()
```

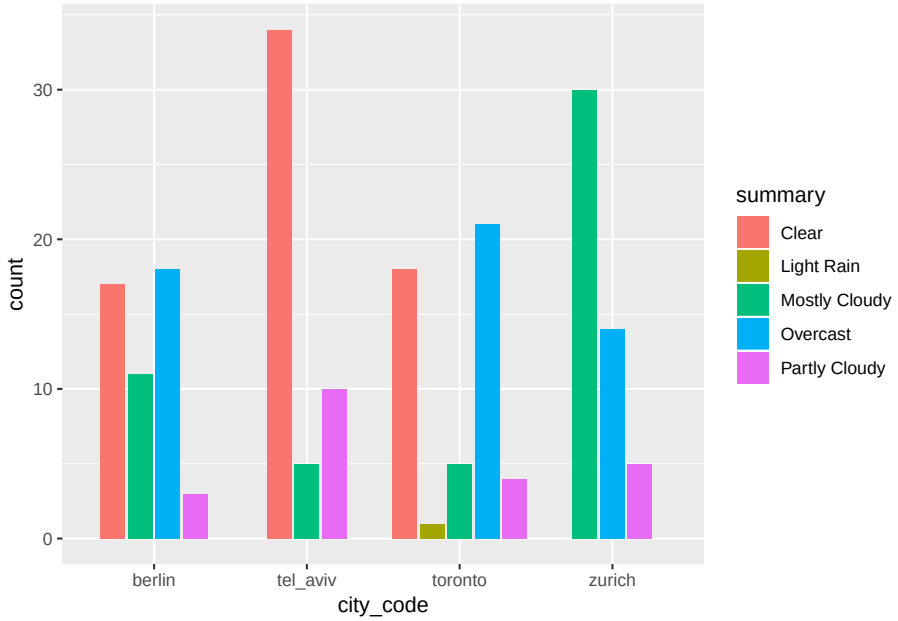


Barplots of number of occurences of each kind of weather per city:

```
weather_data %>%
  ggplot(aes(x = city_code)) +
  geom_bar(aes(fill = summary))
```



```
weather_data %>%
  ggplot(aes(x = city_code)) +
  geom_bar(aes(fill = summary), position = position_dodge2("dodge", preserve = "single"
```



Lineplot with different line types and an additional visualisation of the line range (here, difference between apparent and actual temperature):

```
weather_data %>%
  select(city_code, time, temperature, apparentTemperature) %>%
  gather(kind, temperature, -city_code, -time)
```

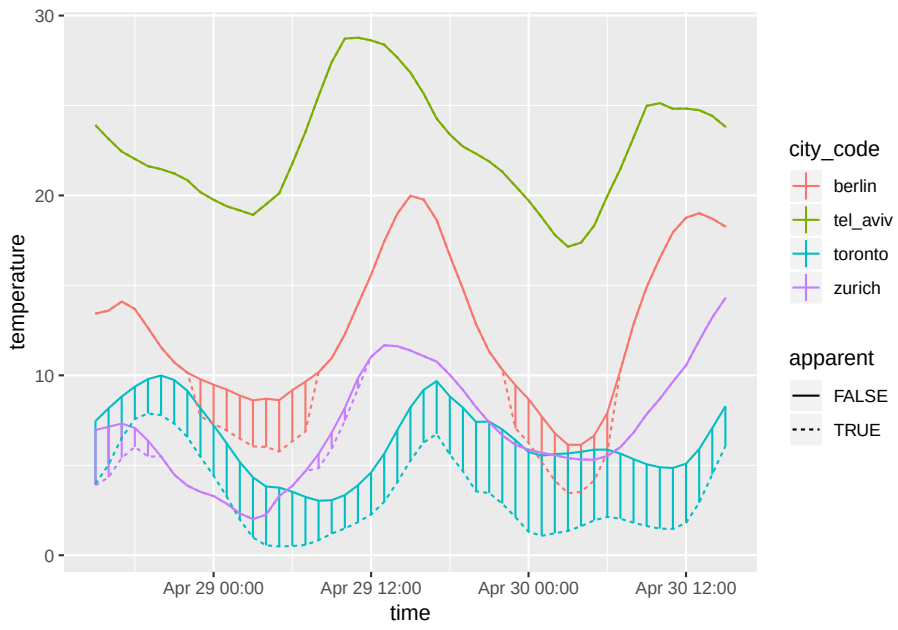
```
## # A tibble: 392 x 4
##   city_code time                kind        temperature
##   <chr>     <dttm>              <chr>             <dbl>
## 1 berlin    2019-04-28 15:00:00 temperature        13.4
## 2 berlin    2019-04-28 16:00:00 temperature        13.6
## 3 berlin    2019-04-28 17:00:00 temperature        14.1
## # ... with 389 more rows
```

```
temperature_data <-
  weather_data %>%
  select(city_code, time, temperature, apparentTemperature) %>%
  gather(kind, temperature, -city_code, -time) %>%
  mutate(apparent = (kind == "apparentTemperature")) %>%
  select(-kind)
```

```
temperature_data
```

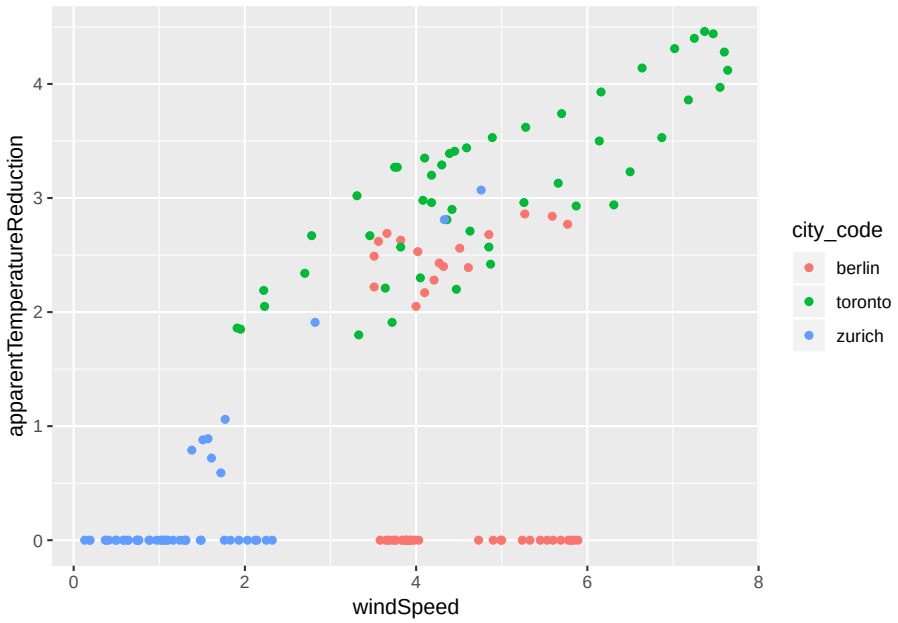
```
## # A tibble: 392 x 4
##   city_code time                temperature apparent
##   <chr>     <dttm>                    <dbl> <lgl>
## 1 berlin    2019-04-28 15:00:00        13.4 FALSE
## 2 berlin    2019-04-28 16:00:00        13.6 FALSE
## 3 berlin    2019-04-28 17:00:00        14.1 FALSE
## # ... with 389 more rows
```

```
temperature_data %>%
  ggplot(aes(x = time, color = city_code)) +
  geom_linerange(data = weather_data, aes(ymin = temperature, ymax = apparentTemperature)) +
  geom_line(aes(linetype = apparent, y = temperature))
```



Relation of temperature difference between actual and apparent temperature (cf. line range in last plot) with wind speed, shown as scatter plot.

```
weather_data %>%
  mutate(apparentTemperatureReduction = temperature - apparentTemperature) %>%
  filter(city_code != "tel_aviv") %>%
  ggplot(aes(x = windSpeed, y = apparentTemperatureReduction)) +
  geom_point(aes(color = city_code))
```



1.2.3 Further dplyr transformations

If you want to compare measurements of the same observable at two different points in time, maybe the most straightforward way to do so is to create a new column with an appropriate lag:

```
weather_data %>%
  group_by(city_code) %>%
  mutate_at(vars(temperature, pressure, humidity), list(lag = lag)) %>%
  ungroup()
```

```
## # A tibble: 196 x 22
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 193 more rows, and 17 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>, temperature_lag <dbl>,
## #   pressure_lag <dbl>, humidity_lag <dbl>
```

Count observations per category or combinations of categories:

```
weather_data %>%
  count(city_code)
```

```
## # A tibble: 4 x 2
##   city_code     n
##   <chr>     <int>
## 1 berlin       49
## 2 tel_aviv     49
## 3 toronto      49
## 4 zurich       49
```

```
weather_data %>%
  count(city_code, summary)
```

```
## # A tibble: 15 x 3
##   city_code summary           n
##   <chr>     <chr>         <int>
## 1 berlin    Clear            17
## 2 berlin    Mostly Cloudy    11
## 3 berlin    Overcast         18
## # ... with 12 more rows
```

Use `summarize()` to create a tibble with mean and maximum temperature for each city:

```
weather_data %>%
  group_by(city_code) %>%
  summarize(temperature_mean = mean(temperature), temperature_max = max(temperature)) %>%
  ungroup()
```

```
## # A tibble: 4 x 3
##   city_code temperature_mean temperature_max
##   <chr>                <dbl>           <dbl>
## 1 berlin               12.5            20.0
## 2 tel_aviv             22.6            28.8
## 3 toronto               6.39            9.99
## 4 zurich                7.15           14.3
```

Compute and display summary data for all numeric variables:

```
weather_data %>%
  group_by(city_code) %>%
  summarize_if(is.numeric, list(mean = mean, sd = sd, min = min, max = max)) %>%
  ungroup() %>%
  gather(key, value, -city_code) %>%
  separate(key, into = c("indicator", "fun")) %>%
  xtabs(value ~ city_code + indicator + fun, .) %>%
  ftable()
```

```
##                                  fun          max          mean          min
## city_code indicator
## berlin    apparentTemperature       19.98000000   11.62836735    3.45000000    5.073
##           cloudCover                 1.00000000    0.59734694    0.00000000    0.423
##           dewPoint                  10.18000000    5.23632653    1.88000000    2.458
##           humidity                   0.88000000    0.63448980    0.35000000    0.155
##           ozone                    378.04000000  343.34571429  319.68000000   21.608
##           precipIntensity            0.34800000    0.03084286    0.00000000    0.079
##           precipProbability          0.54000000    0.06000000    0.00000000    0.139
##           pressure                1020.48000000 1018.71714286 1016.10000000    1.192
##           temperature               19.98000000   12.49795918    6.14000000    4.100
##           uvIndex                    5.00000000    1.24489796    0.00000000    1.614
##           visibility                16.09000000   15.77102041   10.01000000    1.133
##           windBearing              358.00000000  151.59183673    4.00000000  152.792
##           windGust                  11.14000000    7.59591837    3.67000000    2.187
##           windSpeed                  5.89000000    4.49326531    3.51000000    0.813
## tel_aviv  apparentTemperature       28.77000000   22.64591837   17.15000000    3.152
##           cloudCover                 0.81000000    0.19693878    0.00000000    0.249
##           dewPoint                  14.43000000   12.18244898    9.51000000    1.292
##           humidity                   0.74000000    0.52612245    0.34000000    0.093
##           ozone                    339.37000000  318.31836735  307.16000000   10.058
##           precipIntensity            0.00000000    0.00000000    0.00000000    0.000
##           precipProbability          0.00000000    0.00000000    0.00000000    0.000
##           pressure                1015.88000000 1013.66265306 1011.73000000    1.125
##           temperature               28.77000000   22.64591837   17.15000000    3.152
##           uvIndex                   10.00000000    2.40816327    0.00000000    3.564
##           visibility                16.09000000   15.87163265   10.01000000    1.079
##           windBearing              355.00000000  188.36734694    0.00000000  123.849
##           windGust                   5.53000000    3.47775510    1.66000000    1.194
##           windSpeed                  4.90000000    2.49285714    0.57000000    1.048
## toronto   apparentTemperature        7.88000000    3.27306122    0.49000000    2.248
##           cloudCover                 1.00000000    0.59510204    0.00000000    0.431
##           dewPoint                   3.05000000   -1.26653061   -5.17000000    2.691
##           humidity                   0.82000000    0.59734694    0.37000000    0.146
##           ozone                    401.89000000  362.02632653  327.57000000   23.424
##           precipIntensity            0.84070000    0.08387551    0.00000000    0.166
##           precipProbability          0.51000000    0.06653061    0.00000000    0.117
##           pressure                1030.07000000 1025.14918367 1019.55000000    3.220
##           temperature                9.99000000    6.38795918    3.03000000    2.023
##           uvIndex                    6.00000000    1.40816327    0.00000000    1.847
##           visibility                16.09000000   15.14673469    5.13000000    2.838
##           windBearing              357.00000000  140.32653061    2.00000000  129.308
##           windGust                  11.51000000    7.51020408    2.66000000    2.337
##           windSpeed                  7.64000000    4.87510204    1.91000000    1.623
## zurich    apparentTemperature       14.30000000    6.88551020    2.01000000    3.144
##           cloudCover                 1.00000000    0.80877551    0.37000000    0.157
```

```
##           dewPoint                7.23000000    3.38367347   -0.27000000    1.90397030
##           humidity                0.88000000    0.77551020    0.59000000    0.08304269
##           ozone                 377.57000000  359.81510204  340.69000000   11.33226737
##           precipIntensity         0.26670000    0.07106939    0.00000000    0.05976032
##           precipProbability       0.29000000    0.13326531    0.00000000    0.07816616
##           pressure             1023.55000000 1021.37612245 1017.40000000    1.62120174
##           temperature            14.30000000    7.14510204    2.01000000    3.07049475
##           uvIndex                 4.00000000    1.10204082    0.00000000    1.44690615
##           visibility             16.09000000   12.90938776    3.89000000    4.47872769
##           windBearing           357.00000000  147.61224490   20.00000000  102.66182679
##           windGust                4.76000000    1.98428571    1.07000000    0.95327506
##           windSpeed               4.76000000    1.31244898    0.13000000    0.91823774
```

Chapter 2

Function basics

Structuring the code to avoid too much copy-pasting

This chapter discusses functions as building blocks for more expressive and more powerful data analysis code.

2.1 Definition and execution

The following packages are used for this chapter.

```
library(tidyverse)
library(here)
```

Create functions for tasks that need to be executed repeatedly, or to hide implementation details.

```
read_weather_data <- function() {
  # Read all files
  berlin <- readxl::read_excel(here("data/weather", "berlin.xlsx"))
  toronto <- readxl::read_excel(here("data/weather", "toronto.xlsx"))
  tel_aviv <- readxl::read_excel(here("data/weather", "tel_aviv.xlsx"))
  zurich <- readxl::read_excel(here("data/weather", "zurich.xlsx"))

  # Create ensemble dataset
  weather_data <- bind_rows(
    berlin = berlin,
    toronto = toronto,
    tel_aviv = tel_aviv,
    zurich = zurich,
    .id = "city_code"
```

```
  )

  # Return it
  weather_data
}
```

Display the code of any function by writing its name without the subsequent parentheses:

```
read_weather_data
```

```
## function() {
##   # Read all files
##   berlin <- readxl::read_excel(here("data/weather", "berlin.xlsx"))
##   toronto <- readxl::read_excel(here("data/weather", "toronto.xlsx"))
##   tel_aviv <- readxl::read_excel(here("data/weather", "tel_aviv.xlsx"))
##   zurich <- readxl::read_excel(here("data/weather", "zurich.xlsx"))
##
##   # Create ensemble dataset
##   weather_data <- bind_rows(
##     berlin = berlin,
##     toronto = toronto,
##     tel_aviv = tel_aviv,
##     zurich = zurich,
##     .id = "city_code"
##   )
##
##   # Return it
##   weather_data
## }
## <environment: 0x36a96e8>
```

Call the function by adding the parentheses:

```
read_weather_data()
```

```
## # A tibble: 196 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 193 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

Execution of the function does not create new variables in the global environment. The only object in the global environment is the function itself:

```
ls()
```

```
## [1] "read_weather_data"
```

A function can also be used as input for a pipe:

```
read_weather_data() %>%
  count(city_code)
```

```
## # A tibble: 4 x 2
##   city_code     n
##   <chr>     <int>
## 1 berlin       49
## 2 tel_aviv     49
## 3 toronto      49
## 4 zurich       49
```

To reuse a function value, assign it to a variable:

```
weather_data <- read_weather_data()
```

2.1.1 Exercises

1. Create a modified version of the function to return only data for Toronto and Tel Aviv. Call it.

   ```
   read_weather_data_non_europe <- function() {
     -------
   }


   -------
   ```

   ```
   ## # A tibble: 98 x 19
   ##   city_code time                summary icon  precipIntensity
   ##   <chr>     <dttm>              <chr>   <chr>           <dbl>
   ## 1 toronto   2019-04-28 15:00:00 Partly~ part~               0
   ## 2 toronto   2019-04-28 16:00:00 Clear   clea~               0
   ## 3 toronto   2019-04-28 17:00:00 Clear   clea~               0
   ## # ... with 95 more rows, and 14 more variables: precipProbability <dbl>,
   ## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
   ## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
   ## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
   ## #   ozone <dbl>, precipType <chr>
   ```

2. Compute number of rows for Europe, count observations to validate:

```
nrow(_____) - nrow(_____)
```

```
## [1] 98
```

2.2 Arguments

Click here to show setup code.

```
library(tidyverse)
library(here)
```

By adding arguments to your functions, you can turn them into tools for a wide range of applications. But it is advisable to be conservative here: try to minimise the number of arguments to the necessary ones, so the user has a clear and intuitive interface to deal with.

Functions with arguments:

```
weather_path <- function(filename) {
  # Returned value
  here("data/weather", filename)
}

weather_path("milan.xlsx")
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/milan.xlsx"
```

Call functions from within functions:

```
read_weather_data <- function() {
  # Read all files
  berlin <- readxl::read_excel(weather_path("berlin.xlsx"))
  toronto <- readxl::read_excel(weather_path("toronto.xlsx"))
  tel_aviv <- readxl::read_excel(weather_path("tel_aviv.xlsx"))
  zurich <- readxl::read_excel(weather_path("zurich.xlsx"))

  # Create ensemble dataset
  weather_data <- bind_rows(
    berlin = berlin,
    toronto = toronto,
    tel_aviv = tel_aviv,
    zurich = zurich,
    .id = "city_code"
  )

  # Return it
```

```
  weather_data
}
```

The function still needs to be called for testing it. It is a good practice to always immediately test a the newly created or updated function by running it:

```
read_weather_data()
```

```
## # A tibble: 196 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 193 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

2.2.1 Exercises

1. How does the behavior of `read_weather_data()` change if we update the definition of the `read_weather()` function as follows:

   ```
   weather_path <- function(filename) {
     # Returned value
     here("data", "weather", filename)
   }
   ```

   Hint: Define this function with a different name and check its output values, before running `read_weather_data()` again.

2.3 Use case: Intermediate variables

Click here to show setup code.

```
library(tidyverse)
library(here)

weather_path <- function(filename) {
  # Returned value
  here("data/weather", filename)
}
```

We start with the function `weather_path()` from section “Arguments”.

Functions can help to avoid having to use intermediate variables:

```
read_weather_file <- function(filename) {
  readxl::read_excel(weather_path(filename))
}

read_weather_data <- function() {
  # Create ensemble dataset from files on disk
  weather_data <- bind_rows(
    berlin = read_weather_file("berlin.xlsx"),
    toronto = read_weather_file("toronto.xlsx"),
    tel_aviv = read_weather_file("tel_aviv.xlsx"),
    zurich = read_weather_file("zurich.xlsx"),
    .id = "city_code"
  )

  # Return it
  weather_data
}

read_weather_data()
```

```
## # A tibble: 196 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 193 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

2.3.1 Exercises

1. Implement a helper function `get_weather_file_for()` that takes a city code as input and returns the file name for the corresponding Excel file. Intended usage: `get_weather_file_for("berlin")`. Test this function on a few example inputs.

   ```
   get_weather_file_for <- _____ {
     paste0(city_code, ".xlsx")
   }
   ```

```
get_weather_file_for("munich")
```

```
## [1] "munich.xlsx"
```

```
get_weather_file_for("san_diego")
```

```
## [1] "san_diego.xlsx"
```

2. Implement a helper function `get_weather_data_for()` that takes a city code as input (as opposed to a file name). Intended usage: `get_weather_data_for("berlin")`. Update `read_weather_data()` to use `get_weather_data_for()`.

```
get_weather_data_for <- _____ {
  read_weather_file(_____)
}
```

```
get_weather_data_for("toronto")
```

```
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
```

2.4 Default values

Click here to show setup code.

```
library(tidyverse)
library(here)


weather_path <- function(filename) {
  # Returned value
  here("data/weather", filename)
}

read_weather_file <- function(filename) {
  readxl::read_excel(weather_path(filename))
```

```
}

get_weather_file_for <- function(city_code) {
  paste0(city_code, ".xlsx")
}

get_weather_data_for <- function(city_code) {
  read_weather_file(get_weather_file_for(city_code))
}
```

For user-friendliness it is often good practice to provide default values for parameters

We start with the function`get_weather_data_for()` from section “Intermediate variables”.

Here an example of a boolean argument which when `TRUE` leads to dropping the data about Zurich.

```
read_weather_data <- function(omit_zurich = FALSE) {
  # Create ensemble dataset from files on disk
  weather_data <- bind_rows(
    berlin = get_weather_data_for("berlin"),
    toronto = get_weather_data_for("toronto"),
    tel_aviv = get_weather_data_for("tel_aviv"),
    zurich = get_weather_data_for("zurich"),
    .id = "city_code"
  )

  # Return it (filtered)
  weather_data %>%
    filter( !(city_code == "zurich" & omit_zurich) )
}
```

Set arguments with default values explicitly with or without using the name or leave them out to use the default value:

```
read_weather_data(TRUE)
```

```
## # A tibble: 147 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 144 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
```

```
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

```
read_weather_data(omit_zurich = TRUE)
```

```
## # A tibble: 147 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 144 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

```
read_weather_data()
```

```
## # A tibble: 196 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 193 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

2.4.1 Exercises

1. Update `get_weather_data_for()` to return Zurich data if called without arguments. Is this a good idea?

   ```
   get_weather_data_for <- _____ {
     _____
   }
   ```

   ```
   get_weather_data_for() %>%
     select(temperature)
   ```

   ```
   ## # A tibble: 49 x 1
   ##   temperature
   ##         <dbl>
   ## 1        6.96
   ```

```
## 2        7.14
## 3        7.32
## # ... with 46 more rows
```

```
get_weather_data_for("tel_aviv") %>%
  select(temperature)
```

```
## # A tibble: 49 x 1
##   temperature
##         <dbl>
## 1        23.9
## 2        23.1
## 3        22.4
## # ... with 46 more rows
```

2.5 Multiple arguments

Click here to show setup code.

```
library(tidyverse)
library(here)


weather_path <- function(filename) {
  # Returned value
  here("data/weather", filename)
}

read_weather_file <- function(filename) {
  readxl::read_excel(weather_path(filename))
}

get_weather_file_for <- function(city_code) {
  paste0(city_code, ".xlsx")
}

get_weather_data_for <- function(city_code) {
  read_weather_file(get_weather_file_for(city_code))
}
```

We start once more with the functions `weather_path()` from section "Arguments" and `get_weather_data_for()` from section "Intermediate variables".

What are the considerations when using multiple function arguments? You can add new parameters in a very straightforward manner like this:

```
read_weather_data <- function(omit_zurich = FALSE, omit_toronto = FALSE) {
  # Create ensemble dataset from files on disk
  weather_data <- bind_rows(
    berlin = get_weather_data_for("berlin"),
    toronto = get_weather_data_for("toronto"),
    tel_aviv = get_weather_data_for("tel_aviv"),
    zurich = get_weather_data_for("zurich"),
    .id = "city_code"
  )

  # Return it (filtered)
  weather_data %>%
    filter( !(city_code == "zurich" & omit_zurich) ) %>%
    filter( !(city_code == "toronto" & omit_toronto) )
}
```

Prefer passing arguments by name rather than only giving the value, especially if the intent of the value is not clear from just reading it.

```
# Good:
read_weather_data(omit_zurich = TRUE)
```

```
## # A tibble: 147 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 144 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

```
read_weather_data(omit_toronto = TRUE)
```

```
## # A tibble: 147 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 144 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

```
# Bad:
read_weather_data(TRUE)
```

```
## # A tibble: 147 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 144 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

Use the so called ellipsis (`...`) when you want to provide the possibility for the user to call your function with a list of arguments of unspecified length. This can be e.g. useful for passing arguments downstream:

```
weather_path <- function(...) {
  # All arguments are passed on
  here("data/weather", ...)
}

weather_path("berlin.xlsx")
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
```

```
weather_path("some", "subdir", "with", "a", "file.csv")
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/some/subdir/with/a/file.csv"
```

Mind, that despite altering the original function and adding new features to it, the original call still works as before:

```
read_weather_data()
```

```
## # A tibble: 196 x 19
##   city_code time                summary icon  precipIntensity
##   <chr>     <dttm>              <chr>   <chr>           <dbl>
## 1 berlin    2019-04-28 15:00:00 Mostly~ part~               0
## 2 berlin    2019-04-28 16:00:00 Mostly~ part~               0
## 3 berlin    2019-04-28 17:00:00 Mostly~ part~               0
## # ... with 193 more rows, and 14 more variables: precipProbability <dbl>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>, precipType <chr>
```

2.5.1 Exercises

1. What does the following return? Why?

```
read_weather_data(TRUE, omit_z = FALSE) %>%
  count(city_code)
```

   See the next section for ideas on avoiding this behavior.

2.6 Argument matching

Click here to show setup code.

How does `R` handle function calls with arguments?

Named arguments are assigned first, after that remaining slots are filled from left to right.

```
use_names <- function(one = 1, two = 2) {
  list(one = one, two = two)
}

use_names(3, 4)
```

```
## $one
## [1] 3
##
## $two
## [1] 4
```

```
use_names(one = 3, 4)
```

```
## $one
## [1] 3
##
## $two
## [1] 4
```

```
use_names(3, one = 4)
```

```
## $one
## [1] 4
##
## $two
## [1] 3
```

```
use_names(one = 3, two = 4)
```

```
## $one
```

```
## [1] 3
##
## $two
## [1] 4
```

```
use_names(two = 3, one = 4)
```

```
## $one
## [1] 4
##
## $two
## [1] 3
```

Arguments are matched partially, which can be convenient but is also a source of errors.

```
use_names(o = 3, 4)
```

```
## $one
## [1] 3
##
## $two
## [1] 4
```

```
use_names(3, o = 4)
```

```
## $one
## [1] 4
##
## $two
## [1] 3
```

```
use_names(o = 3, t = 4)
```

```
## $one
## [1] 3
##
## $two
## [1] 4
```

```
use_names(t = 3, o = 4)
```

```
## $one
## [1] 4
##
## $two
## [1] 3
```

The ellipsis can be used to enforce the user to fully name the function parameters when setting them:

```
only_names <- function(..., one = 1, two = 2) {
  list(one = one, two = two)
}

only_names(3, 4)
```

```
## $one
## [1] 1
##
## $two
## [1] 2
```

```
only_names(one = 3, 4)
```

```
## $one
## [1] 3
##
## $two
## [1] 2
```

```
only_names(one = 3, two = 4)
```

```
## $one
## [1] 3
##
## $two
## [1] 4
```

```
only_names(o = 3, t = 4)
```

```
## $one
## [1] 1
##
## $two
## [1] 2
```

Inside a function with an ellipsis as a parameter, you can capture the ellipsis with `list()`:

```
ellipsis_test <- function(...) {
  args <- list(...)
  names(args)
}

ellipsis_test(a = 1, 2, c = 3:5)
```

```
## [1] "a" ""  "c"
```

Arguments in ellipsis can be accessed with `..1`, `..2` etc.

```
ellipsis_direct_test <- function(...) {
  list(..1, ..2)
}

ellipsis_direct_test(a = 1, 2, c = 3:5)
```

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

2.6.1 Exercises

1. Naming, not naming or partly naming parameters in functions calls? What does the following return and why?

```
use_some_names <- function(one = 1, ..., two = 2) {
  list(one = one, two = two)
}

use_some_names(3, 4)
use_some_names(one = 3, 4)
use_some_names(3, one = 4)
use_some_names(one = 3, two = 4)
use_some_names(two = 4, 3)
```

2. Model a new `enforce_names()` function after `only_names()` to check if any unnamed or misnamed arguments have been used. Test this function in various combinations.

```
enforce_names <- function(..., one = 1, two = 2) {
  extra_args <- _____
  stopifnot(length(_____) == 0)

  list(_____)
}
```

```
try(enforce_names(3, 4))
```

```
## Error in enforce_names(3, 4) : length(extra_args) == 0 is not TRUE
```

```
try(enforce_names(one = 3, 4))
```

```
## Error in enforce_names(one = 3, 4) : length(extra_args) == 0 is not TRUE
```

```
try(enforce_names(3, one = 4))
```

```
## Error in enforce_names(3, one = 4) : length(extra_args) == 0 is not TRUE
```

```
try(enforce_names(two = 4, 3))
```

```
## Error in enforce_names(two = 4, 3) : length(extra_args) == 0 is not TRUE
```

```
try(enforce_names(o = 3, t = 4))
```

```
## Error in enforce_names(o = 3, t = 4) :
##   length(extra_args) == 0 is not TRUE
```

```
enforce_names(one = 3, two = 4)
```

```
## $one
## [1] 3
##
## $two
## [1] 4
```

Chapter 3

Simple iteration

> Processing multiple files that contain different parts of the same dataset

This chapter introduces iteration as a concept to repeat the same operation over a sequence of inputs. It is largely independent of the previous chapter.

The following packages are required throughout this chapter:

```
library(tidyverse)
library(here)
```

3.1 Vectors and columns

So far we focus on the data frame, or tibble, as primary object for data analysis. Internally, a tibble is a list of vectors of the same length. Accessing a row in a tibble requires finding the same index in that list of vectors.

Here we explore the relationship between columns of data frames and their corresponding vectors, i.e. the answer to "how to get from one to the other?":

We can e.g. get a vector with the files in a specific directory of our current project[1] like this:

```
files <- dir(here("data/weather"), full.names = TRUE)
files
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
## [2] "/home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx"
## [3] "/home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx"
```

[1] function `here::here()` is taking care of making sure we start from the root directory of our current project

```
## [4] "/home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx"
```

You can create a tibble from it using `tibble::enframe()`:

```
files_df <-
  files %>%
  enframe()

files_df
```

```
## # A tibble: 4 x 2
##    name value
##   <int> <chr>
## 1     1 /home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx
## 2     2 /home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx
## 3     3 /home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx
## 4     4 /home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx
```

The `name` column might be unwanted in some cases. Suppress its creation by setting `name = NULL`:

```
files_df_1 <-
  files %>%
  enframe(name = NULL)

files_df_1
```

```
## # A tibble: 4 x 1
##   value
##   <chr>
## 1 /home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx
## 2 /home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx
## 3 /home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx
## 4 /home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx
```

Another way to create a tibble from a vector is using `tibble::tibble()`. You can name the newly created columns by assigning the vectors they are created from to (quoted or unquoted) column names:

```
tibble(filename = files)
```

```
## # A tibble: 4 x 1
##   filename
##   <chr>
## 1 /home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx
## 2 /home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx
## 3 /home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx
## 4 /home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx
```

The other direction – producing a vector from a tibble column – works with

`dplyr::pull()`. By default `pull()` will turn the rightmost column into a vector and ignore the rest of the tibble:

```
files_df %>%
  pull()
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
## [2] "/home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx"
## [3] "/home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx"
## [4] "/home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx"
```

Turn a specific column into a vector by providing the desired column name to `pull()`, either quoted or unquoted:

```
files_df %>%
  pull(name)
```

```
## [1] 1 2 3 4
```

3.1.1 Exercises

1. Investigate the output of `fs::dir_ls()` with `enframe()`. Explain.

   ```
   # install.packages("fs")
   fs::dir_ls()
   fs::dir_ls() %>%
     ___()
   ```

   ```
   ## # A tibble: 45 x 2
   ##   name         value
   ##   <chr>        <fs::path>
   ## 1 1.Rmd        1.Rmd
   ## 2 12-intro.Rmd 12-intro.Rmd
   ## 3 2.Rmd        2.Rmd
   ## # ... with 42 more rows
   ```

3.2 Named vectors and two-column tibbles

Click here to show setup code.

```
library(tidyverse)
library(here)
```

Here we look at tidyverse-functions to work with named vectors and tibbles with more columns and the relations netween the two.

As seen in section "Data", load a table – here a dictionary detailing information related to an id-like name – from an MS Excel file with `readxl::read_excel()`:

```
dict <- readxl::read_excel(here("data/cities.xlsx"))
dict
```

```
## # A tibble: 4 x 5
##   city_code weather_filename           name        lng   lat
##   <chr>     <chr>                      <chr>     <dbl> <dbl>
## 1 berlin    data/weather/berlin.xlsx   Berlin     13.4   52.5
## 2 toronto   data/weather/toronto.xlsx  Toronto   -79.4   43.7
## 3 tel_aviv  data/weather/tel_aviv.xlsx Tel Aviv  34.8   32.1
## 4 zurich    data/weather/zurich.xlsx   Zürich      8.54  47.4
```

Use `pull()` as seen in the last chapter:

```
dict %>%
  pull(weather_filename)
```

```
## [1] "data/weather/berlin.xlsx"   "data/weather/toronto.xlsx"
## [3] "data/weather/tel_aviv.xlsx" "data/weather/zurich.xlsx"
```

Create absolute paths using `here::here()`:

```
dict %>%
  pull(weather_filename) %>%
  here()
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
## [2] "/home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx"
## [3] "/home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx"
## [4] "/home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx"
```

Produce a named vector with `tibble::deframe()`, which is thought as the inverse function to `enframe()`. When given a 2-column tibble, `deframe()` will by default use the first column for the names and the second column for the values of the resulting vector. When given a 1-column tibble, it creates an unnamed vector. When given a more-than-2-column tibble, it will use the first two columns as name- and value-columns for the resulting vector, ignore the rest and in addition give a warning that it expects a one- or two-column data frame.

```
weather_filenames <-
  dict %>%
  select(city_code, weather_filename) %>%
  deframe()
weather_filenames
```

```
##                        berlin                       toronto
##   "data/weather/berlin.xlsx"  "data/weather/toronto.xlsx"
##                      tel_aviv                        zurich
```

```
## "data/weather/tel_aviv.xlsx"   "data/weather/zurich.xlsx"
```

The `names()` function accesses the names of a vector:

```
weather_filenames %>%
  names()
```

```
## [1] "berlin"   "toronto"  "tel_aviv" "zurich"
```

Some operations producing vectors from vectors cause the names to be lost:

```
paste0("'", weather_filenames, "'")
```

```
## [1] "'data/weather/berlin.xlsx'"   "'data/weather/toronto.xlsx'"
## [3] "'data/weather/tel_aviv.xlsx'" "'data/weather/zurich.xlsx'"
```

```
weather_filenames %>%
  here()
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
## [2] "/home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx"
## [3] "/home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx"
## [4] "/home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx"
```

A possible solution can be in many cases to change the order of the transformations, so that the creation of the named vector comes last (or at least later):

```
dict %>%
  mutate(weather_filename_here = here(weather_filename))
```

```
## # A tibble: 4 x 6
##   city_code weather_filename   name      lng   lat weather_filename_here
##   <chr>     <chr>              <chr>   <dbl> <dbl> <chr>
## 1 berlin    data/weather/berl~ Berlin  13.4   52.5 /home/travis/build/krlm~
## 2 toronto   data/weather/toro~ Toron~ -79.4   43.7 /home/travis/build/krlm~
## 3 tel_aviv  data/weather/tel_~ Tel A~  34.8   32.1 /home/travis/build/krlm~
## 4 zurich    data/weather/zuri~ Zürich   8.54  47.4 /home/travis/build/krlm~
```

```
dict %>%
  mutate(weather_filename_here = here(weather_filename)) %>%
  select(city_code, weather_filename_here)
```

```
## # A tibble: 4 x 2
##   city_code weather_filename_here
##   <chr>     <chr>
## 1 berlin    /home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx
## 2 toronto   /home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx
## 3 tel_aviv  /home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx
## 4 zurich    /home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx
```

```
dict %>%
  mutate(weather_filename_here = here(weather_filename)) %>%
  select(city_code, weather_filename_here) %>%
  deframe()
```

```
##                                                                berlin
##   "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
##                                                              toronto
##  "/home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx"
##                                                            tel_aviv
## "/home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx"
##                                                               zurich
##   "/home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx"
```

3.2.1 Exercises

1. Obtain a mapping between city code and city name as a named vector.

   ```
   dict %>%
     select(___, ___) %>%
     deframe()
   ```

   ```
   ##     berlin    toronto   tel_aviv     zurich
   ##   "Berlin"  "Toronto" "Tel Aviv"   "Zürich"
   ```

2. Convert the output of `fs::dir_info()` to that seen from `fs::dir_ls()`. How do you make sure that the vector is named?

   ```
   # install.packages("fs")
   fs::dir_info()
   fs::dir_info() %>%
     pull(___)
   fs::dir_info() %>%
     select(_____, _____) %>%
     ___()
   ```

   ```
   ## # A tibble: 45 x 18
   ##   path       type     size permissions modification_time   user  group
   ##   <fs::path> <fct> <fs::b> <fs::perms> <dttm>              <chr> <chr>
   ## 1 1.Rmd      file       56 rw-rw-r--   2019-05-29 19:54:06 trav~ trav~
   ## 2 12-intro.~ file    5.85K rw-rw-r--   2019-05-29 19:54:06 trav~ trav~
   ## 3 2.Rmd      file      370 rw-rw-r--   2019-05-29 19:54:06 trav~ trav~
   ## # ... with 42 more rows, and 11 more variables: device_id <dbl>,
   ## #   hard_links <dbl>, special_device_id <dbl>, inode <dbl>,
   ## #   block_size <dbl>, blocks <dbl>, flags <int>, generation <dbl>,
   ## #   access_time <dttm>, change_time <dttm>, birth_time <dttm>
   ```

```
## 1.Rmd                 12-intro.Rmd          2.Rmd
## 21-function.Rmd       22-args.Rmd           23-intermediate.Rmd
## 24-args-default.Rmd   25-args-multi.Rmd     26-args-matching.Rmd
## 3.Rmd                 31-dir.Rmd            32-names.Rmd
## 33-index.Rmd          34-construct.Rmd      35-map.Rmd
## 36-map-manip.Rmd      37-map-type.Rmd       4.Rmd
## 41-map2.Rmd           42-mutate-map.Rmd
```


```
## 1.Rmd                 12-intro.Rmd          2.Rmd
## 21-function.Rmd       22-args.Rmd           23-intermediate.Rmd
## 24-args-default.Rmd   25-args-multi.Rmd     26-args-matching.Rmd
## 3.Rmd                 31-dir.Rmd            32-names.Rmd
## 33-index.Rmd          34-construct.Rmd      35-map.Rmd
## 36-map-manip.Rmd      37-map-type.Rmd       4.Rmd
## 41-map2.Rmd           42-mutate-map.Rmd
```


3.3 Indexing/subsetting

Click here to show setup code.

```
library(tidyverse)
library(here)

dict <- readxl::read_excel(here("data/cities.xlsx"))
```

Here we look at the indexing of a named vector (works equivalently for a named list).

We start with the data frame `dict` from section "Named vectors and two-column tibbles". Create named vector of the – future – input files:

```
input_files <-
  dict %>%
  select(city_code, weather_filename) %>%
  deframe()
input_files
```

```
##                        berlin                       toronto
##   "data/weather/berlin.xlsx"  "data/weather/toronto.xlsx"
##                     tel_aviv                        zurich
## "data/weather/tel_aviv.xlsx"   "data/weather/zurich.xlsx"
```

```
names(input_files)
```

```
## [1] "berlin"   "toronto"  "tel_aviv" "zurich"
```

There are different ways of accessing individual entries of the named vector:

```
input_files[1]
```

```
##                      berlin
## "data/weather/berlin.xlsx"
```

```
input_files[[1]]
```

```
## [1] "data/weather/berlin.xlsx"
```

```
input_files["berlin"]
```

```
##                      berlin
## "data/weather/berlin.xlsx"
```

```
input_files[["berlin"]]
```

```
## [1] "data/weather/berlin.xlsx"
```

Choose multiple entries with:

```
input_files[1:2]
```

```
##                       berlin                      toronto
##  "data/weather/berlin.xlsx" "data/weather/toronto.xlsx"
```

```
input_files[c("berlin", "zurich")]
```

```
##                      berlin                      zurich
## "data/weather/berlin.xlsx" "data/weather/zurich.xlsx"
```

Consistent pipe-friendly access of single elements:

```
input_files %>%
  pluck(1)
```

```
## [1] "data/weather/berlin.xlsx"
```

```
input_files %>%
  pluck("berlin")
```

```
## [1] "data/weather/berlin.xlsx"
```

3.3.1 Exercises

1. Explain the difference between `[` and `[[` subsetting.
2. Implement a variant of subsetting in "tibble-land" with a combination of `enframe()`, `slice()` or `filter()`, and `deframe()`:

```
input_files %>%
  enframe() %>%
```

```
  ___(_____) %>%
  deframe()
```

```
##                       toronto
## "data/weather/toronto.xlsx"
```

3.4 Construction

Click here to show setup code.

The `c()` function constructs vectors. All elements of a vector must have the same type.

```
c(1, 2, 3)
```

```
## [1] 1 2 3
```

```
c(1:3, 5)
```

```
## [1] 1 2 3 5
```

```
c(1:3, "5")
```

```
## [1] "1" "2" "3" "5"
```

Lists are constructed with `list()`. They are a special type of vector – they can contain elements of different type and length.

```
list(1, 2, 3)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

```
list(1:3, 5)
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 5
```

```
list(1:3, "5")
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "5"
```

Lists can also contain other lists.

```
nested <- list(
  1:3,
  list(4, "5"),
  list(
    list(letters[6:8]),
    9
  )
)

nested
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [[2]][[1]]
## [1] 4
##
## [[2]][[2]]
## [1] "5"
##
##
## [[3]]
## [[3]][[1]]
## [[3]][[1]][[1]]
## [1] "f" "g" "h"
##
##
## [[3]][[2]]
## [1] 9
```

```
str(nested)
```

```
## List of 3
##  $ : int [1:3] 1 2 3
##  $ :List of 2
##   ..$ : num 4
##   ..$ : chr "5"
##  $ :List of 2
##   ..$ :List of 1
```

```
##   .. ..$ : chr [1:3] "f" "g" "h"
##   ..$ : num 9
```

Vectors (and also lists) can have names.

```
c(a = 1, b = 2, c = 3)
```

```
## a b c
## 1 2 3
```

```
list(a = 1:3, b = 5)
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] 5
```

```
rlang::set_names(1:3, letters[1:3])
```

```
## a b c
## 1 2 3
```

The new {vctrs} package defines a data type for lists where all elements have the same type: a stricter list, but more powerful than a bare vector.

```
#vctrs::list_of(1, 2, 3)
#try(vctrs::list_of(1, 2, "3"))
#vctrs::list_of(letters[1:3], "e")
```

3.4.1 Exercises

1. Explain the differences between the outputs below.

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

   ```
   ## $a
   ## [1] 1 2 3
   ##
   ## $b
   ## [1] 4 5
   ```

   ```
   list(a = list(1:3), b = list(4:5))
   ```

   ```
   ## $a
   ## $a[[1]]
   ## [1] 1 2 3
   ##
   ##
   ## $b
   ```

```
## $b[[1]]
## [1] 4 5
```

3.5 Processing multiple files

Click here to show setup code.

```
library(tidyverse)
library(here)

dict <- readxl::read_excel(here("data/cities.xlsx"))

input_files <-
  dict %>%
  select(city_code, weather_filename) %>%
  deframe()
```

Here we look at how to act on each entry of a list or a vector using `purrr::map()`:

We start with the named vector `input_files` from section "Indexing". As just seen, manually choosing just one entry of a vector works like so:

```
input_files[[1]]
```

```
## [1] "data/weather/berlin.xlsx"
```

```
here(input_files[[1]])
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
```

```
readxl::read_excel(here(input_files[[1]]))
```

```
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
```

However, if we try to read all files from the vector with `read_excel()`, this fails:

```
here(input_files)
```

```
## [1] "/home/travis/build/krlmlr/tidyprog/data/weather/berlin.xlsx"
```

```
## [2] "/home/travis/build/krlmlr/tidyprog/data/weather/toronto.xlsx"
## [3] "/home/travis/build/krlmlr/tidyprog/data/weather/tel_aviv.xlsx"
## [4] "/home/travis/build/krlmlr/tidyprog/data/weather/zurich.xlsx"
```

```
try(readxl::read_excel(here(input_files)))
```

```
## Error : `path` must be a string
```

Unlike `here()`, the `read_excel()` function can process only one file at a time. We need to iterate explicitly.

With `map()`, you can successively work through the whole vector and each time let the same function deal with the respective entry. The output of `map()` is a list where each element contains one results. The list is named if the input is named:

```
input_data <-
  map(input_files, ~ readxl::read_excel(here(.)))
```

The `map()` call above is equivalent to the following code:

```
input_data <-
  list(
    berlin = readxl::read_excel(here(input_files[[1]])),
    toronto = readxl::read_excel(here(input_files[[2]])),
    tel_aviv = readxl::read_excel(here(input_files[[3]])),
    zurich = readxl::read_excel(here(input_files[[4]]))
  )
```

Let's take a closer look at what we produced:

```
input_data
```

```
## $berlin
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $toronto
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
```

```
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $tel_aviv
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>
##
## $zurich
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~           0.267             0.28
## 2 2019-04-28 16:00:00 Mostly~ part~           0.198             0.27
## 3 2019-04-28 17:00:00 Mostly~ part~           0.137             0.25
## # ... with 46 more rows, and 13 more variables: precipType <chr>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>
```

```
input_data[[1]]
```

```
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
```

```
## #   precipType <chr>
```

```
names(input_data)
```

```
## [1] "berlin"   "toronto"  "tel_aviv" "zurich"
```

map() can be included in your pipe in the following way:

```
input_files %>%
  map(~ readxl::read_excel(here(.)))
```

```
## $berlin
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $toronto
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $tel_aviv
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
```

```
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>
##
## $zurich
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~           0.267             0.28
## 2 2019-04-28 16:00:00 Mostly~ part~           0.198             0.27
## 3 2019-04-28 17:00:00 Mostly~ part~           0.137             0.25
## # ... with 46 more rows, and 13 more variables: precipType <chr>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>
```

3.5.1 Exercises

1. Read only the data for Toronto and Tel Aviv, using subsetting or `filter()`. Compare.

```
input_files[_____] %>%
  map(~ readxl::read_excel(here(.)))

input_files %>%
  ___() %>%
  filter(name %in% c(_____)) %>%
  ___() %>%
  map(~ readxl::read_excel(here(.)))
```

```
## $toronto
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $tel_aviv
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
```

```
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>

## $toronto
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $tel_aviv
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>
```

2. Compute the absolute file name with `here()` outside of the `map()` call.

```
input_files %>%
  enframe() %>%
  ___(_____) %>%
  deframe() %>%
  map(~ readxl::read_excel(.))
```

```
## $berlin
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
```

```
## 2 2019-04-28 16:00:00 Mostly~ part~              0               0
## 3 2019-04-28 17:00:00 Mostly~ part~              0               0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $toronto
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## $tel_aviv
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>
##
## $zurich
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~           0.267             0.28
## 2 2019-04-28 16:00:00 Mostly~ part~           0.198             0.27
## 3 2019-04-28 17:00:00 Mostly~ part~           0.137             0.25
## # ... with 46 more rows, and 13 more variables: precipType <chr>,
## #   temperature <dbl>, apparentTemperature <dbl>, dewPoint <dbl>,
## #   humidity <dbl>, pressure <dbl>, windSpeed <dbl>, windGust <dbl>,
## #   windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>,
## #   ozone <dbl>
```

3. Can you explain what happens when you call `enframe()` on the result?

```
input_files %>%
  map(~ readxl::read_excel(here(.))) %>%
  enframe()
```

3.6 Manipulating all datasets

Click here to show setup code.

```
library(tidyverse)
library(here)

dict <- readxl::read_excel(here("data/cities.xlsx"))

input_data <-
  dict %>%
  select(city_code, weather_filename) %>%
  deframe() %>%
  map(~ readxl::read_excel(here(.)))
```

How to selectively manipulate specific parts of a list of datasets?

We start with the named list of tibbles called `input_data` from section "Processing all files". Of each tibble we only want the column `time` and all the columns whose name contains "emperature". We test with the first entry:

```
input_data[[1]] %>%
  select(time, contains("emperature"))
```

```
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        13.4                13.4
## 2 2019-04-28 16:00:00        13.6                13.6
## 3 2019-04-28 17:00:00        14.1                14.1
## # ... with 46 more rows
```

To apply this on all entries, we use `map()` again. Note that we need an explicit dot (`.`) in the `select()` call here, to indicate the position where each sub-dataset will be plugged in.

```
input_data %>%
  map(~ select(., time, contains("emperature")))
```

```
## $berlin
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
```

```
##   <dttm>                   <dbl>               <dbl>
## 1 2019-04-28 15:00:00       13.4                13.4
## 2 2019-04-28 16:00:00       13.6                13.6
## 3 2019-04-28 17:00:00       14.1                14.1
## # ... with 46 more rows
##
## $toronto
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        7.46                3.96
## 2 2019-04-28 16:00:00        8.17                5.04
## 3 2019-04-28 17:00:00        8.82                6.52
## # ... with 46 more rows
##
## $tel_aviv
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        23.9                23.9
## 2 2019-04-28 16:00:00        23.1                23.1
## 3 2019-04-28 17:00:00        22.4                22.4
## # ... with 46 more rows
##
## $zurich
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        6.96                3.89
## 2 2019-04-28 16:00:00        7.14                4.33
## 3 2019-04-28 17:00:00        7.32                5.41
## # ... with 46 more rows
```

We can extend this to preserve only the observations with `temperature` greater or equal than 14°C:

```
input_data %>%
  map(~ select(., time, contains("emperature"))) %>%
  map(~ filter(., temperature >= 14))
```

```
## $berlin
## # A tibble: 16 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 17:00:00       14.1                14.1
## 2 2019-04-29 12:00:00       15.6                15.6
## 3 2019-04-29 13:00:00       17.4                17.4
```

```
## # ... with 13 more rows
##
## $toronto
## # A tibble: 0 x 3
## # ... with 3 variables: time <dttm>, temperature <dbl>,
## #   apparentTemperature <dbl>
##
## $tel_aviv
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        23.9                23.9
## 2 2019-04-28 16:00:00        23.1                23.1
## 3 2019-04-28 17:00:00        22.4                22.4
## # ... with 46 more rows
##
## $zurich
## # A tibble: 1 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-30 15:00:00        14.3                14.3
```

Create a custom function for that specific purpose in a call to `map()`:

```
find_good_times <- function(data) {
  data %>%
    select(time, contains("emperature")) %>%
    filter(temperature >= 14)
}
```

Let's look at the object `manipulator`, that we created:

```
find_good_times
```

```
## function(data) {
##   data %>%
##     select(time, contains("emperature")) %>%
##     filter(temperature >= 14)
## }
## <environment: 0x36a96e8>
```

See the "Function basics" chapter for a more extensive introduction to functions.

Testing the function:

```
find_good_times(input_data[[4]])
```

```
## # A tibble: 1 x 3
##   time                temperature apparentTemperature
```

```
##   <dttm>                     <dbl>               <dbl>
## 1 2019-04-30 15:00:00         14.3                14.3
```

Now let's use `map()` to let our function act on the entire dataset:

```
good_times <- map(input_data, ~ find_good_times(.))
good_times
```

```
## $berlin
## # A tibble: 16 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 17:00:00        14.1                14.1
## 2 2019-04-29 12:00:00        15.6                15.6
## 3 2019-04-29 13:00:00        17.4                17.4
## # ... with 13 more rows
##
## $toronto
## # A tibble: 0 x 3
## # ... with 3 variables: time <dttm>, temperature <dbl>,
## #   apparentTemperature <dbl>
##
## $tel_aviv
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        23.9                23.9
## 2 2019-04-28 16:00:00        23.1                23.1
## 3 2019-04-28 17:00:00        22.4                22.4
## # ... with 46 more rows
##
## $zurich
## # A tibble: 1 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-30 15:00:00        14.3                14.3
```

`map()` allows for the following shortcut notation for functions with one argument only:

```
map(input_data, find_good_times)
```

```
## $berlin
## # A tibble: 16 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 17:00:00        14.1                14.1
## 2 2019-04-29 12:00:00        15.6                15.6
```

```
## 3 2019-04-29 13:00:00        17.4                17.4
## # ... with 13 more rows
##
## $toronto
## # A tibble: 0 x 3
## # ... with 3 variables: time <dttm>, temperature <dbl>,
## #   apparentTemperature <dbl>
##
## $tel_aviv
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        23.9                23.9
## 2 2019-04-28 16:00:00        23.1                23.1
## 3 2019-04-28 17:00:00        22.4                22.4
## # ... with 46 more rows
##
## $zurich
## # A tibble: 1 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-30 15:00:00        14.3                14.3
```

3.6.1 Exercises

1. Use `summarize()` to compute the mean temperature and humidity for each city during that period.

```
input_data %>%
  ___(~ ___(., mean(___), mean(___)))
```

```
## $berlin
## # A tibble: 1 x 2
##   `mean(temperature)` `mean(humidity)`
##                 <dbl>            <dbl>
## 1                12.5            0.634
##
## $toronto
## # A tibble: 1 x 2
##   `mean(temperature)` `mean(humidity)`
##                 <dbl>            <dbl>
## 1                6.39            0.597
##
## $tel_aviv
## # A tibble: 1 x 2
```

```
##   `mean(temperature)` `mean(humidity)`
##                 <dbl>            <dbl>
## 1                22.6            0.526
##
## $zurich
## # A tibble: 1 x 2
##   `mean(temperature)` `mean(humidity)`
##                 <dbl>            <dbl>
## 1                7.15            0.776
```

2. Create a function to compute the daily mean of these values for each dataset:

```
compute_daily_mean <- ___(data) {
  data %>%
    ___(as.Date(time)) %>%
    ___(_____) %>%
    ungroup()
}
input_data %>%
  ___(___)
```

```
## $berlin
## # A tibble: 3 x 3
##   `as.Date(time)` `mean(temperature)` `mean(humidity)`
##   <date>                        <dbl>            <dbl>
## 1 2019-04-28                     12.2            0.636
## 2 2019-04-29                     12.7            0.690
## 3 2019-04-30                     12.3            0.551
##
## $toronto
## # A tibble: 3 x 3
##   `as.Date(time)` `mean(temperature)` `mean(humidity)`
##   <date>                        <dbl>            <dbl>
## 1 2019-04-28                     8.96            0.398
## 2 2019-04-29                     5.84            0.554
## 3 2019-04-30                     5.76            0.774
##
## $tel_aviv
## # A tibble: 3 x 3
##   `as.Date(time)` `mean(temperature)` `mean(humidity)`
##   <date>                        <dbl>            <dbl>
## 1 2019-04-28                     21.9            0.542
## 2 2019-04-29                     23.6            0.477
## 3 2019-04-30                     21.7            0.591
##
## $zurich
```

```
## # A tibble: 3 x 3
##   `as.Date(time)` `mean(temperature)` `mean(humidity)`
##   <date>                        <dbl>            <dbl>
## 1 2019-04-28                     5.80            0.778
## 2 2019-04-29                     7.09            0.756
## 3 2019-04-30                     7.98            0.803
```

3. Use the `dim()` function to compute the dimensions of each sub-dataset. Then, use `prod()` to compute the number of cells. Discuss your observation.

```
input_data %>%
  _____
input_data %>%
  _____ %>%
  _____
```

```
## $berlin
## [1] 49 18
##
## $toronto
## [1] 49 18
##
## $tel_aviv
## [1] 49 17
##
## $zurich
## [1] 49 18
```

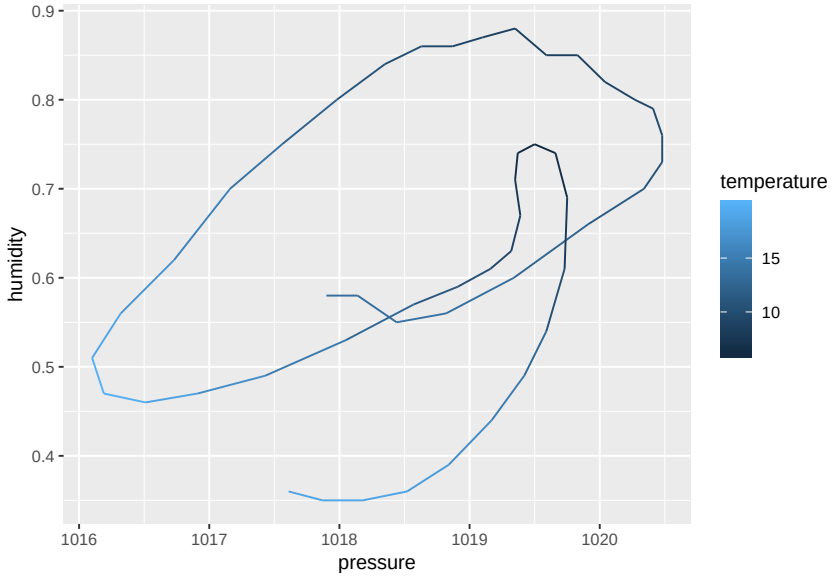
```
## $berlin
## [1] 882
##
## $toronto
## [1] 882
##
## $tel_aviv
## [1] 833
##
## $zurich
## [1] 882
```

4. Create four plots of humidity vs. pressure, one for each city. Use `geom_path()`, map `temperature` to the `color` aesthetic:

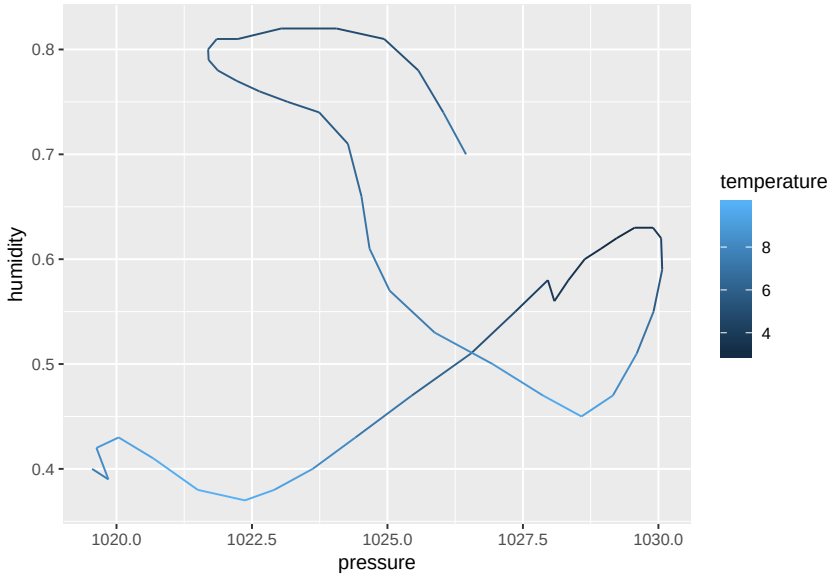
```
create_plot <- function(___) {
  ___ %>%
    ggplot(aes(_____)) +
    geom_path()
}
```

```
___ %>%
  ___(___)
```

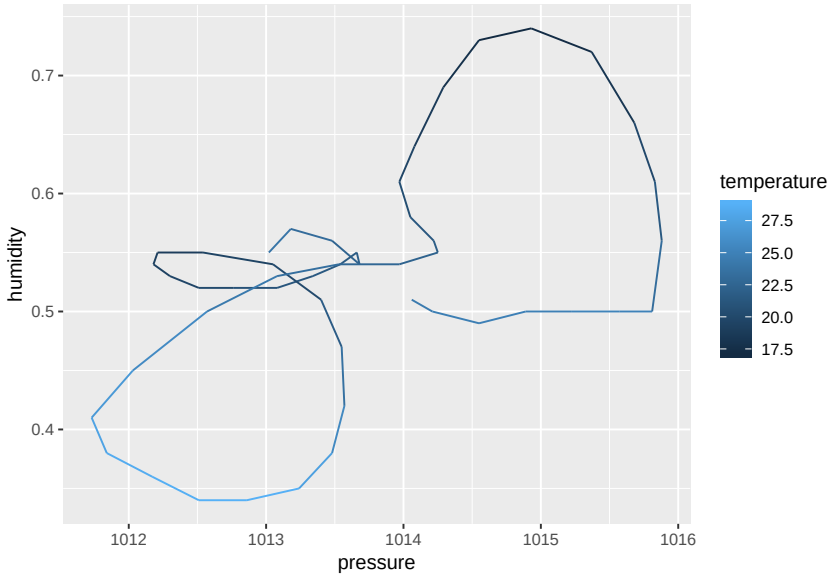
```
## $berlin
```



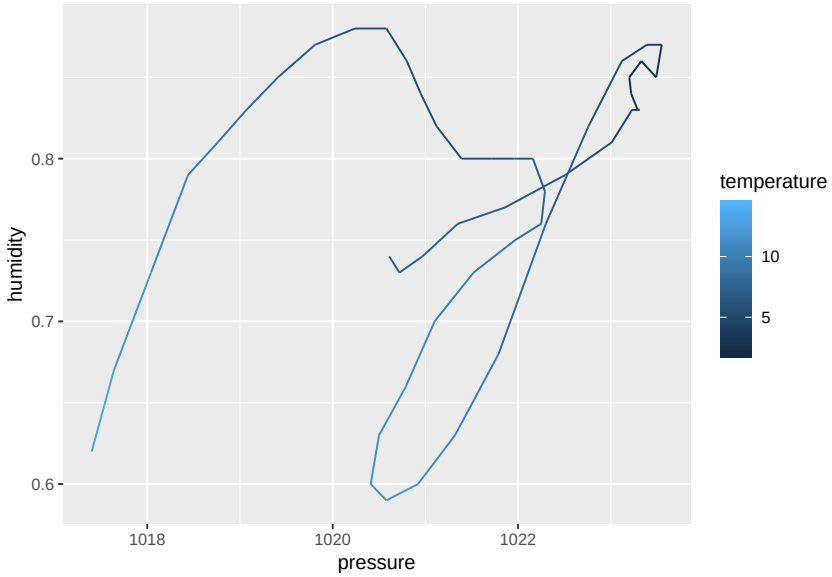

```
##
## $toronto
```



```
##
## $tel_aviv
```




```
##
## $zurich
```



3.7 Typed output

Click here to show setup code.

```
library(tidyverse)
library(here)

dict <- readxl::read_excel(here("data/cities.xlsx"))

input_data <-
  dict %>%
  select(city_code, weather_filename) %>%
  deframe() %>%
  map(~ readxl::read_excel(here(.)))
```

If we know, what the output of each function call in a `map()` sequence looks like, we can often call a sub-type of `map()` to produce a more condensed output.

We start with the named list of tibbles called `input_data` from section "Processing all files".

We want to know the number of rows of each tibble in `input_data`:

```
input_data %>%
  map(~ nrow(.))
```

```
## $berlin
## [1] 49
##
## $toronto
## [1] 49
##
## $tel_aviv
## [1] 49
##
## $zurich
## [1] 49
```

Each time an integer is produced. Therefore we can call `map_int()`, to create a named integer vector:

```
input_data %>%
  map_int(~ nrow(.))
```

```
##   berlin  toronto tel_aviv   zurich
##       49       49       49       49
```

If the output is of type character, use `map_chr()`:

```
input_data %>%
  map_chr(~ nrow(.))
```

```
##   berlin  toronto tel_aviv   zurich
##     "49"     "49"     "49"     "49"
```

```
input_data %>%
  map_chr(~ as.character(nrow(.)))
```

```
##   berlin  toronto tel_aviv   zurich
##     "49"     "49"     "49"     "49"
```

There are sub-types of the `map()` function for each atomic type:

- integer: `map_int()`
- numeric (double-precision value): `map_dbl()`
- character (strings): `map_chr()`
- logical (flags): `map_lgl()`
- raw (bytes): `map_raw()`

3.7.1 Exercises

1. Explain what happens if you try to use `map_dbl()` with the `dim()` output:

   ```
   input_data %>%
     map_dbl(dim)
   ```

2. Extract a concise version of the first temperature value for each dataset:

   ```
   input_data %>%
     map(~ slice(., 1)) %>%
     ___(~ pull(_____))
   ```

   ```
   ##   berlin  toronto tel_aviv   zurich
   ##    13.43     7.46    23.90     6.96
   ```

3. Use `paste0()` to build a textual description for the weather during the observed period in a function. Create a two-column tibble.

   ```
   summarize_weather <- _____ {
     ___ %>%
       ___(
         _____,
         _____,
         _____,
         summary = paste(rle(summary)$values, collapse = ", then ")
       )
   ```

```
}

describe_weather <- function(weather_summary) {
  weather_summary %>%
    mutate(
      text = paste0(
        "We had temperatures between ", min_temp, " and ", max_temp, " °C.",
        "The average humidity was ", round(mean_humidity * 100), " %. ",
        "The weather was ", summary, "."
      )
    ) %>%
    pull()
}

input_data %>%
  ___(___) %>%
  ___(___) %>%
  ___()
```

```
## # A tibble: 4 x 2
##   name     value
##   <chr>    <chr>
## 1 berlin   We had temperatures between 6.14 and 19.98 °C.The average humid~
## 2 toronto  We had temperatures between 3.03 and 9.99 °C.The average humidi~
## 3 tel_aviv We had temperatures between 17.15 and 28.77 °C.The average humi~
## 4 zurich   We had temperatures between 2.01 and 14.3 °C.The average humidi~
```

Chapter 4

Pairwise iteration and nesting

This chapter explores iterating over pairs (or generally lists) of vectors of the same length. The relationship between vectors and data frame columns is especially helpful here, because values in one row of a tibble naturally correspond to accessing the same index in multiple vectors.

This chapter uses the `manipulated_data` object from the "Manipulating all datasets" section.

```
library(tidyverse)
library(here)

dict <- readxl::read_excel(here("data/cities.xlsx"))

input_data <-
  dict %>%
  select(city_code, weather_filename) %>%
  deframe() %>%
  map(~ readxl::read_excel(here(.)))

find_good_times <- function(data) {
  data %>%
    select(time, contains("emperature")) %>%
    filter(temperature >= 14)
}

good_times <-
  input_data %>%
  map(find_good_times)
```

```
good_times
```

```
## $berlin
## # A tibble: 16 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 17:00:00        14.1                14.1
## 2 2019-04-29 12:00:00        15.6                15.6
## 3 2019-04-29 13:00:00        17.4                17.4
## # ... with 13 more rows
##
## $toronto
## # A tibble: 0 x 3
## # ... with 3 variables: time <dttm>, temperature <dbl>,
## #   apparentTemperature <dbl>
##
## $tel_aviv
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        23.9                23.9
## 2 2019-04-28 16:00:00        23.1                23.1
## 3 2019-04-28 17:00:00        22.4                22.4
## # ... with 46 more rows
##
## $zurich
## # A tibble: 1 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-30 15:00:00        14.3                14.3
```

4.1 Manipulating pairwise

Here we discuss cases when you want to iterate through two lists (of the same length) in parallel and use each value pair as two of the input parameters of a function.

We first prepare a list of future output filenames:

```
output_filenames <- tempfile(names(good_times), fileext = ".csv")
output_filenames
```

```
## [1] "/tmp/RtmpCquLue/berlin2db82d59d14a.csv"
## [2] "/tmp/RtmpCquLue/toronto2db85d81f229.csv"
## [3] "/tmp/RtmpCquLue/tel_aviv2db853a25188.csv"
```

```
## [4] "/tmp/RtmpCquLue/zurich2db8cd845a8.csv"
```

We want to use `readr::write_csv()` to write each tibble into the respective file. `write_csv()` needs at least 2 arguments: the tibble itself and the path to the filename. For illustration, we implement a file-centric wrapper function that takes the file name as first argument and also prints a message every time a file is written. We use `map2()` to handle this:

```
process_csv <- function(file, data) {
  readr::write_csv(data, file)
  message("Writing ", file)
  invisible(file)
}

map2(good_times, output_filenames, ~ process_csv(..2, ..1))
```

```
## Writing /tmp/RtmpCquLue/berlin2db82d59d14a.csv

## Writing /tmp/RtmpCquLue/toronto2db85d81f229.csv

## Writing /tmp/RtmpCquLue/tel_aviv2db853a25188.csv

## Writing /tmp/RtmpCquLue/zurich2db8cd845a8.csv

## $berlin
## [1] "/tmp/RtmpCquLue/berlin2db82d59d14a.csv"
##
## $toronto
## [1] "/tmp/RtmpCquLue/toronto2db85d81f229.csv"
##
## $tel_aviv
## [1] "/tmp/RtmpCquLue/tel_aviv2db853a25188.csv"
##
## $zurich
## [1] "/tmp/RtmpCquLue/zurich2db8cd845a8.csv"
```

```
invisible(map2(good_times, output_filenames, ~ process_csv(..2, ..1)))
```

```
## Writing /tmp/RtmpCquLue/berlin2db82d59d14a.csv

## Writing /tmp/RtmpCquLue/toronto2db85d81f229.csv

## Writing /tmp/RtmpCquLue/tel_aviv2db853a25188.csv

## Writing /tmp/RtmpCquLue/zurich2db8cd845a8.csv
```

Because `process_csv()` returns the file name, it is available as output. Since we are just interested in the side-effects of `write_csv()` and not in the displayed output, we can use the related function `walk2()`.

```
walk2(good_times, output_filenames, ~ process_csv(..2, ..1))
```

```
## Writing /tmp/RtmpCquLue/berlin2db82d59d14a.csv

## Writing /tmp/RtmpCquLue/toronto2db85d81f229.csv

## Writing /tmp/RtmpCquLue/tel_aviv2db853a25188.csv

## Writing /tmp/RtmpCquLue/zurich2db8cd845a8.csv
```

```
print(walk2(good_times, output_filenames, ~ process_csv(..2, ..1)))
```

```
## Writing /tmp/RtmpCquLue/berlin2db82d59d14a.csv

## Writing /tmp/RtmpCquLue/toronto2db85d81f229.csv

## Writing /tmp/RtmpCquLue/tel_aviv2db853a25188.csv

## Writing /tmp/RtmpCquLue/zurich2db8cd845a8.csv

## $berlin
## # A tibble: 16 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 17:00:00        14.1                14.1
## 2 2019-04-29 12:00:00        15.6                15.6
## 3 2019-04-29 13:00:00        17.4                17.4
## # ... with 13 more rows
##
## $toronto
## # A tibble: 0 x 3
## # ... with 3 variables: time <dttm>, temperature <dbl>,
## #   apparentTemperature <dbl>
##
## $tel_aviv
## # A tibble: 49 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-28 15:00:00        23.9                23.9
## 2 2019-04-28 16:00:00        23.1                23.1
## 3 2019-04-28 17:00:00        22.4                22.4
## # ... with 46 more rows
##
## $zurich
## # A tibble: 1 x 3
##   time                temperature apparentTemperature
##   <dttm>                    <dbl>               <dbl>
## 1 2019-04-30 15:00:00        14.3                14.3
```

`walk2()` returns its first argument so that it can be used in a pipe.

4.1.1 Exercises

1. What does the following code display?

```
good_times %>%
  walk2(output_filenames, ~ readr::write_csv(..1, ..2)) %>%
  map_int(nrow)
```

4.2 Moving to tibble-land

Click here to show setup code.

```
library(tidyverse)
library(here)

dict <- readxl::read_excel(here("data/cities.xlsx"))

input_data <-
  dict %>%
  select(city_code, weather_filename) %>%
  deframe() %>%
  map(~ readxl::read_excel(here(.)))

find_good_times <- function(data) {
  data %>%
    select(time, contains("emperature")) %>%
    filter(temperature >= 14)
}

good_times <-
  input_data %>%
  map(find_good_times)
```

How to combine the abilities of `map()` & co., which work on vectors and lists, with our commonly used data structure, the tibble?

We start with the named list of tibbles called `input_data` from section "Processing all files" and with `dict` from section "Named vectors and two-column tibbles".

Calling `enframe()` to produce a data frame from `input_data` leads to a maybe at first surprising, but oftentimes useful result:

```
nested_input_data <-
  input_data %>%
  enframe()
```

```
nested_input_data
```

```
## # A tibble: 4 x 2
##   name     value
##   <chr>    <list>
## 1 berlin   <tibble [49 x 18]>
## 2 toronto  <tibble [49 x 18]>
## 3 tel_aviv <tibble [49 x 17]>
## 4 zurich   <tibble [49 x 18]>
```

This is because lists are also vectors. In our case each list entry contains a tibble, which can be "nested" into each entry of column `value`.

Starting with the tibble `dict` we can see how `dpylr::mutate()` and `map()` can nicely work together to produce a somewhat similar result:

```
dict %>%
  select(city_code, weather_filename) %>%
  mutate(
    data = map(weather_filename, ~ readxl::read_excel(here(.)))
  )
```

```
## # A tibble: 4 x 3
##   city_code weather_filename           data
##   <chr>     <chr>                      <list>
## 1 berlin    data/weather/berlin.xlsx   <tibble [49 x 18]>
## 2 toronto   data/weather/toronto.xlsx  <tibble [49 x 18]>
## 3 tel_aviv  data/weather/tel_aviv.xlsx <tibble [49 x 17]>
## 4 zurich    data/weather/zurich.xlsx   <tibble [49 x 18]>
```

This works because `R` interprets columns of tibbles as vectors, which can be fed to `map()`. To simplify the `map()` call, we create an intermediate column:

```
dict %>%
  select(city_code, weather_filename) %>%
  mutate(path = here(weather_filename)) %>%
  mutate(data = map(path, readxl::read_excel))
```

```
## # A tibble: 4 x 4
##   city_code weather_filename      path                           data
##   <chr>     <chr>                 <chr>                          <list>
## 1 berlin    data/weather/berlin.~ /home/travis/build/krlmlr/tid~ <tibble [~
## 2 toronto   data/weather/toronto~ /home/travis/build/krlmlr/tid~ <tibble [~
## 3 tel_aviv  data/weather/tel_avi~ /home/travis/build/krlmlr/tid~ <tibble [~
## 4 zurich    data/weather/zurich.~ /home/travis/build/krlmlr/tid~ <tibble [~
```

Staying in "tibble-land" as long as possible helps retaining other important components of the data you are processing, so that you can keep using familiar

data transformation tools.

```
dict_data <-
  dict %>%
  mutate(
    data = map(weather_filename, ~ readxl::read_excel(here(.))),
    rows = map_int(data, nrow),
  ) %>%
  select(-weather_filename)
dict_data
```

```
## # A tibble: 4 x 6
##   city_code name        lng   lat data                rows
##   <chr>     <chr>     <dbl> <dbl> <list>             <int>
## 1 berlin    Berlin     13.4   52.5 <tibble [49 x 18]>    49
## 2 toronto   Toronto   -79.4   43.7 <tibble [49 x 18]>    49
## 3 tel_aviv  Tel Aviv   34.8   32.1 <tibble [49 x 17]>    49
## 4 zurich    Zürich      8.54  47.4 <tibble [49 x 18]>    49
```

This pattern can also be used with the `map2()` family of functions:

```
dict_data_with_desc <-
  dict_data %>%
  mutate(
    desc = map2_chr(
      name, rows,
      ~ paste0(..2, " rows in data for ", ..1)
    )
  )
```

Because `mutate()` always appends to the end, the most recently added column can always be accessed with `pull()`:

```
dict_data_with_desc %>%
  pull()
```

```
## [1] "49 rows in data for Berlin"   "49 rows in data for Toronto"
## [3] "49 rows in data for Tel Aviv" "49 rows in data for Zürich"
```

More generally, `pmap()` supports functions with an arbitrary number of arguments:

```
dict_data %>%
  mutate(
    cols = map_int(data, ncol),
    desc = pmap_chr(
      list(name, rows, cols),
      ~ paste0(..2, " rows and ", ..3, " cols in data for ", ..1)
    )
```

```
  )
```

```
## # A tibble: 4 x 8
##   city_code name       lng   lat data        rows  cols desc
##   <chr>     <chr>    <dbl> <dbl> <list>     <int> <int> <chr>
## 1 berlin    Berlin   13.4   52.5 <tibble [~    49    18 49 rows and 18 col~
## 2 toronto   Toronto -79.4   43.7 <tibble [~    49    18 49 rows and 18 col~
## 3 tel_aviv  Tel Av~  34.8   32.1 <tibble [~    49    17 49 rows and 17 col~
## 4 zurich    Zürich    8.54  47.4 <tibble [~    49    18 49 rows and 18 col~
```

4.2.1 Exercises

1. The `imap()` family of functions iterates over a vector and its names:

```
input_data %>%
  imap_chr(~ paste0(.y, ": ", nrow(.x), " rows"))
```

```
##              berlin             toronto            tel_aviv
##   "berlin: 49 rows"  "toronto: 49 rows" "tel_aviv: 49 rows"
##              zurich
##   "zurich: 49 rows"
```

   Implement the same functionality using `map2()` inside a `mutate()`, and `enframe()`:

```
good_times %>%
  ___() %>%
  mutate(___ = map2()) %>%
  deframe()
```

4.3 Nesting and unnesting

Click here to show setup code.

```
library(tidyverse)
library(here)

dict <- readxl::read_excel(here("data/cities.xlsx"))

dict_data <-
  dict %>%
  mutate(data = map(weather_filename, ~ readxl::read_excel(here(.)))) %>%
  select(-weather_filename)
```

How to work with nested data?

We start with the tibble `dict_data` from section "Moving to tibble-land", which includes the nested tibbles in its column `data`.

If we want to actually look at the data we can directly use `tidyr::unnest()` on the whole tibble, which by default acts on all list-columns. This expands our tibble by repeating the formerly unnested column entries as many times, as each nested tibble has rows:

```
dict_data %>%
  unnest()
```

```
## # A tibble: 196 x 22
##   city_code name    lng   lat time                summary icon
##   <chr>     <chr> <dbl> <dbl> <dttm>              <chr>   <chr>
## 1 berlin    Berl~  13.4  52.5 2019-04-28 15:00:00 Mostly~ part~
## 2 berlin    Berl~  13.4  52.5 2019-04-28 16:00:00 Mostly~ part~
## 3 berlin    Berl~  13.4  52.5 2019-04-28 17:00:00 Mostly~ part~
## # ... with 193 more rows, and 15 more variables: precipIntensity <dbl>,
## #   precipProbability <dbl>, temperature <dbl>, apparentTemperature <dbl>,
## #   dewPoint <dbl>, humidity <dbl>, pressure <dbl>, windSpeed <dbl>,
## #   windGust <dbl>, windBearing <dbl>, cloudCover <dbl>, uvIndex <dbl>,
## #   visibility <dbl>, ozone <dbl>, precipType <chr>
```

This is very similar to `bind_rows()` of the `data` column.

```
dict_data %>%
  pull(data) %>%
  bind_rows()
```

```
## # A tibble: 196 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 193 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
```

```
check_columns_same <- function(x, y) {
  stopifnot(identical(colnames(x), colnames(y)))
}

bind_rows <- function(data_frames) {
  # Called for the side effect
  reduce(data_frames, check_columns_same)
```

```
  dplyr::bind_rows(data_frames)
}

try(
  dict_data %>%
    pull(data) %>%
    bind_rows()
)
```

```
## Error in fn(out, elt, ...) :
##   identical(colnames(x), colnames(y)) is not TRUE
```

Data flattened in this way is useful if the parts can be combined naturally into a larger dataset. Iterating over columns in the nested view corresponds to grouped operations in the flat view.

```
dict_data %>%
  mutate(n = map_int(data, nrow)) %>%
  select(-data)
```

```
## # A tibble: 4 x 5
##   city_code name        lng   lat     n
##   <chr>     <chr>     <dbl> <dbl> <int>
## 1 berlin    Berlin     13.4   52.5    49
## 2 toronto   Toronto   -79.4   43.7    49
## 3 tel_aviv  Tel Aviv   34.8   32.1    49
## 4 zurich    Zürich      8.54  47.4    49
```

```
dict_data %>%
  unnest() %>%
  count(name)
```

```
## # A tibble: 4 x 2
##   name         n
##   <chr>    <int>
## 1 Berlin      49
## 2 Tel Aviv    49
## 3 Toronto     49
## 4 Zürich      49
```

Inversely, if you want to have a more condensed view of your data, you can nest again. By default, the function `tidyr::nest()` will nest all data. Therefore it is often useful to tell it, which columns to ignore:

```
dict_data %>%
  unnest() %>%
  nest(-city_code, -name, -lng, -lat)
```

```
## # A tibble: 4 x 5
```

```
##   city_code name        lng   lat data
##   <chr>     <chr>     <dbl> <dbl> <list>
## 1 berlin    Berlin     13.4   52.5 <tibble [49 x 18]>
## 2 toronto   Toronto   -79.4   43.7 <tibble [49 x 18]>
## 3 tel_aviv  Tel Aviv   34.8   32.1 <tibble [49 x 18]>
## 4 zurich    Zürich      8.54  47.4 <tibble [49 x 18]>
```

Using this, we structure our data in new, customized ways. For processing of daily data over all cities, we create a new column `date`:

```
dict_data %>%
  unnest() %>%
  mutate(date = as.Date(time)) %>%
  nest(-date)
```

```
## # A tibble: 3 x 2
##   date       data
##   <date>     <list>
## 1 2019-04-28 <tibble [36 x 22]>
## 2 2019-04-29 <tibble [96 x 22]>
## 3 2019-04-30 <tibble [64 x 22]>
```

4.3.1 Exercises

1. Implement the following code as a mapping over a nested tibble. Use a helper function:

```
iris %>%
  group_by(Species) %>%
  summarize_all(list(Mean = mean)) %>%
  ungroup()
```

```
## # A tibble: 3 x 5
##   Species Sepal.Length_Me~ Sepal.Width_Mean Petal.Length_Me~
##   <fct>              <dbl>            <dbl>            <dbl>
## 1 setosa              5.01             3.43             1.46
## 2 versic~             5.94             2.77             4.26
## 3 virgin~             6.59             2.97             5.55
## # ... with 1 more variable: Petal.Width_Mean <dbl>
```

```
summarize_to_mean <- function(data) {
  data %>%
    ___(_____)
}

iris %>%
  nest(___) %>%
```

```
  mutate(data = map(___, summarize_to_mean)) %>%
  unnest()
```

2. When is a grouped operation preferable over nesting? Discuss.
3. Data frames are lists under the hood. Explain the output of the following code. What use cases can you imagine?

```
dict_data %>%
  as.list() %>%
  enframe()
```

```
## # A tibble: 5 x 2
##   name      value
##   <chr>     <list>
## 1 city_code <chr [4]>
## 2 name      <chr [4]>
## 3 lng       <dbl [4]>
## 4 lat       <dbl [4]>
## 5 data      <list [4]>
```

Chapter 5

Scoping and flow control

This chapter discusses a few details regarding functions.

5.1 Scope

What happens if a function defines variables that have a variable by the same name in the global environment?

We start with a variable defined in the global environment:

```
a <- 5
```

A function can access global variables:

```
f <- function() {
  a
}

f()
```

```
## [1] 5
```

On the other hand, a variable which is defined inside a function is contained in that function. It will not be known outside of that function. Respectively, it won't overwrite the value of global variables.

```
f <- function() {
  a <- 2
  a
}

f()
```

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

```
a
```

```
## [1] 5
```

Global variables are a (hidden) part of a function's interface. Ideally, functions are be self-contained, independent of global variables. Notable exceptions are objects are used across your entire analysis, such as "the dataset". (Otherwise you would need to pass them across many layers.)

5.1.1 Exercises

1. Double-check what happens if two functions declare/use a variable of the same name.

```
# Variables in different functions
f1 <- function() {
  a <- 3
  a + f2()
}

f2 <- function() {
  a
}

f1()
f2()
a
```

5.2 Pure functions and side effects

Click here to show setup code.

```
library(tidyverse)
```

Functions should do one thing, and do it well.[1]

A *pure function* is one that is called for its return value and which has no side effects:

```
pure_function <- function(x) {
  x + 1
}

pure_function(1)
```

[1] Unix philosophy, originated by Ken Thompson

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

For functions with side effect, it is good practice to return the input invisibly:

```
side_effect_function <- function(x) {
  file <- tempfile()
  writeLines(format(x), tempfile())
  print(x)
  message(x, " written to ", file)

  invisible(x)
}

side_effect_function(2)
```

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

```
## 2 written to /tmp/RtmpCquLue/file2db857e8d848
```

Separation helps isolate the side effects. If side effect functions return the input, they remain composable with pure functions:

```
5 %>%
  pure_function() %>%
  side_effect_function() %>%
  pure_function()
```

```
## [1] 6
```

```
## 6 written to /tmp/RtmpCquLue/file2db8125e18b7
```

```
## [1] 7
```

5.2.1 Exercises

1. In the above example, which part of the pipe triggers the display of `6` and `7`, respectively?
2. How do you create a function that returns more than one value?
3. Implement your own purely functional version of `sum()` by using `reduce()`. (Hint: `` `+` `` is a function that takes two arguments and returns the sum.)

   ```
   reduce(1:5, ___)
   ```

   ```
   ## [1] 15
   ```

4. Implement your own purely functional version of `cumsum()` by using `accumulate()`.

```
accumulate(1:5, ___)
```

```
## [1]  1  3  6 10 15
```

5. Implement your own purely functional version of `cumsum()` by using `reduce()` only. (Hint: Use `tail(., 1)` to access the last element of a vector.)

```
reduce(1:5, ~ _____)
```

```
## [1]  1  3  6 10 15
```

5.3 Control flow

Click here to show setup code.

```
library(tidyverse)
library(here)

weather_path <- function(filename) {
  # Returned value
  here("data/weather", filename)
}
read_weather_file <- function(filename) {
  readxl::read_excel(weather_path(filename))
}
```

We start once more with the functions `weather_path()` from section "Arguments" and `read_weather_file()` from section "Intermediate variables".

A way to regulate the control flow is by using `if ()`:

```
read_weather_data <- function(omit_zurich = FALSE, omit_toronto = FALSE) {
  # Create ensemble dataset from files on disk
  weather_data <- bind_rows(
    berlin = read_weather_file("berlin.xlsx"),
    toronto = read_weather_file("toronto.xlsx"),
    tel_aviv = read_weather_file("tel_aviv.xlsx"),
    zurich = read_weather_file("zurich.xlsx"),
    .id = "city_code"
  )

  # Filter, conditionally
  if (omit_zurich) {
    weather_data <-
      weather_data %>%
      filter(city_code != "zurich")
```

```
  }

  if (omit_toronto) {
    weather_data <-
      weather_data %>%
      filter(city_code != "toronto")
  }

  # Return result
  weather_data
}

read_weather_data(omit_toronto = TRUE, omit_zurich = TRUE) %>%
  count(city_code)
```

```
## # A tibble: 2 x 2
##   city_code     n
##   <chr>     <int>
## 1 berlin       49
## 2 tel_aviv     49
```

```
read_weather_data(omit_toronto = TRUE, omit_zurich = FALSE) %>%
  count(city_code)
```

```
## # A tibble: 3 x 2
##   city_code     n
##   <chr>     <int>
## 1 berlin       49
## 2 tel_aviv     49
## 3 zurich       49
```

This can be useful if aiming at a possible early return:

```
read_weather_data <- function(omit_zurich = FALSE, omit_toronto = FALSE) {
  # Create ensemble dataset from files on disk
  weather_data <- bind_rows(
    berlin = read_weather_file("berlin.xlsx"),
    toronto = read_weather_file("toronto.xlsx"),
    tel_aviv = read_weather_file("tel_aviv.xlsx"),
    zurich = read_weather_file("zurich.xlsx"),
    .id = "city_code"
  )

  # Can keep original data?
  if (!omit_zurich && !omit_toronto) {
    return(weather_data)
  }
```

```
  # Filter, conditionally
  if (omit_zurich) {
    weather_data <-
      weather_data %>%
      filter(city_code != "zurich")
  }

  if (omit_toronto) {
    weather_data <-
      weather_data %>%
      filter(city_code != "toronto")
  }

  # Return result
  weather_data
}
```

Conditional branching with if-else-logic. (This is just for illustration, you should not implement code like this!)

```
read_weather_data <- function(omit_zurich = FALSE, omit_toronto = FALSE) {
  # Create ensemble dataset from files on disk
  weather_data <- bind_rows(
    berlin = read_weather_file("berlin.xlsx"),
    toronto = read_weather_file("toronto.xlsx"),
    tel_aviv = read_weather_file("tel_aviv.xlsx"),
    zurich = read_weather_file("zurich.xlsx"),
    .id = "city_code"
  )

  # Filter, conditionally, and return
  if (!omit_zurich && !omit_toronto) {
    weather_data
  } else if (omit_zurich && !omit_toronto) {
    weather_data %>%
      filter(city_code != "zurich")
  } else if (!omit_zurich && omit_toronto) {
    weather_data %>%
      filter(city_code != "toronto")
  } else {
    # Filter both
    weather_data %>%
      filter(city_code != "zurich") %>%
      filter(city_code != "toronto")
  }
```

```
}

read_weather_data(omit_toronto = TRUE) %>%
  count(city_code)
```

```
## # A tibble: 3 x 2
##   city_code     n
##   <chr>     <int>
## 1 berlin       49
## 2 tel_aviv     49
## 3 zurich       49
```

```
read_weather_data(omit_zurich = TRUE) %>%
  count(city_code)
```

```
## # A tibble: 3 x 2
##   city_code     n
##   <chr>     <int>
## 1 berlin       49
## 2 tel_aviv     49
## 3 toronto      49
```

5.3.1 Exercises

1. Implement a function that branches over an argument and returns the sum or the product of the input, respectively.

```
agg <- function(_____) {
  if (fun == "___") {
    sum(x)
  } else if (_____) {
    prod(___)
  } else {
    rlang::abort('`fun` must be "sum" or "prod".')
  }
}
```

```
agg(1:4, "sum")
```

```
## [1] 10
```

```
agg(1:4, "prod")
```

```
## [1] 24
```

5.4 Closures

Click here to show setup code.

```
library(tidyverse)
library(here)

weather_path <- function(filename) {
  # Returned value
  here("data/weather", filename)
}

read_weather_file <- function(filename) {
  readxl::read_excel(weather_path(filename))
}

get_weather_file_for <- function(city_code) {
  paste0(city_code, ".xlsx")
}

get_weather_data_for <- function(city_code) {
  read_weather_file(get_weather_file_for(city_code))
}
```

Closures can e.g. be used during function definition.

We start once more with the functions `weather_path()` from section "Arguments" and `read_weather_file()` from section "Intermediate variables".

Here we create a function that loads a particular dataset:

```
make_read_weather_file <- function(filename) {
  # Avoid odd effects due to lazy evaluation
  force(filename)

  # This function (closure) accesses the filename from the
  # outer function
  f <- function() {
    read_weather_file(filename)
  }

  f
}

read_berlin <- make_read_weather_file("berlin.xlsx")
read_toronto <- make_read_weather_file("toronto.xlsx")
read_tel_aviv <- make_read_weather_file("tel_aviv.xlsx")
```

```
read_zurich <- make_read_weather_file("zurich.xlsx")

read_berlin
```

```
## function() {
##     read_weather_file(filename)
##   }
## <environment: 0xa373998>
```

```
read_toronto
```

```
## function() {
##     read_weather_file(filename)
##   }
## <environment: 0x9aa9290>
```

```
read_berlin()
```

```
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
```

```
read_toronto()
```

```
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
```

Use closures as wrappers for other verbs/functions (such functions are also called "adverbs"):

```
loudly <- function(f) {
  force(f)
```

```
  function(...) {
    args <- list(...)
    msg <- paste0(length(args), " argument(s)")
    message(msg)

    f(...)
  }
}

read_loudly <- loudly(read_weather_file)
read_loudly
```

```
## function(...) {
##     args <- list(...)
##     msg <- paste0(length(args), " argument(s)")
##     message(msg)
##
##     f(...)
##   }
## <environment: 0x7161160>
```

```
read_loudly("berlin.xlsx")
```

```
## 1 argument(s)
```

```
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
```

The `safely()` function is another example from the purrr package:

```
cities <- list("berlin", "toronto", "milan", "tel_aviv")
try(map(cities, get_weather_data_for))
```

```
## Error : `path` does not exist: '/home/travis/build/krlmlr/tidyprog/data/weather/mila
```

```
map(cities, safely(get_weather_data_for))
```

```
## [[1]]
## [[1]]$result
```

```
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## [[1]]$error
## NULL
##
##
## [[2]]
## [[2]]$result
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## NULL
##
## [[3]]$error
## <simpleError: `path` does not exist: '/home/travis/build/krlmlr/tidyprog/data/weather/milan.xl
##
##
## [[4]]
## [[4]]$result
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
```

```
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>
##
## [[4]]$error
## NULL
```

```
safely(get_weather_data_for)
```

```
## function (...)
## capture_error(.f(...), otherwise, quiet)
## <bytecode: 0x93fe5d0>
## <environment: 0x94fcff8>
```

```
map(cities, ~ safely(get_weather_data_for)(.))
```

```
## [[1]]
## [[1]]$result
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## [[1]]$error
## NULL
##
##
## [[2]]
## [[2]]$result
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
```

```
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## NULL
##
## [[3]]$error
## <simpleError: `path` does not exist: '/home/travis/build/krlmlr/tidyprog/data/weather/milan.xl
##
##
## [[4]]
## [[4]]$result
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>
##
## [[4]]$error
## NULL
```

```
safe_get_weather_data_for <- safely(get_weather_data_for)
map(cities, ~ safe_get_weather_data_for(.))
```

```
## [[1]]
## [[1]]$result
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Mostly~ part~               0                0
## 2 2019-04-28 16:00:00 Mostly~ part~               0                0
## 3 2019-04-28 17:00:00 Mostly~ part~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
```

```
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## [[1]]$error
## NULL
##
##
## [[2]]
## [[2]]$result
## # A tibble: 49 x 18
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 13 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>,
## #   precipType <chr>
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## NULL
##
## [[3]]$error
## <simpleError: `path` does not exist: '/home/travis/build/krlmlr/tidyprog/data/weathe
##
##
## [[4]]
## [[4]]$result
## # A tibble: 49 x 17
##   time                summary icon  precipIntensity precipProbabili~
##   <dttm>              <chr>   <chr>           <dbl>            <dbl>
## 1 2019-04-28 15:00:00 Partly~ part~               0                0
## 2 2019-04-28 16:00:00 Clear   clea~               0                0
## 3 2019-04-28 17:00:00 Clear   clea~               0                0
## # ... with 46 more rows, and 12 more variables: temperature <dbl>,
## #   apparentTemperature <dbl>, dewPoint <dbl>, humidity <dbl>,
## #   pressure <dbl>, windSpeed <dbl>, windGust <dbl>, windBearing <dbl>,
```

```
## #   cloudCover <dbl>, uvIndex <dbl>, visibility <dbl>, ozone <dbl>
##
## [[4]]$error
## NULL
```

5.4.1 Exercises

1. Review the help and the implementation of `safely()` and `possibly()`.

```
safely
```

```
## function (.f, otherwise = NULL, quiet = TRUE)
## {
##     .f <- as_mapper(.f)
##     function(...) capture_error(.f(...), otherwise, quiet)
## }
## <bytecode: 0x93fead8>
## <environment: namespace:purrr>
```

```
possibly
```

```
## function (.f, otherwise, quiet = TRUE)
## {
##     .f <- as_mapper(.f)
##     force(otherwise)
##     function(...) {
##         tryCatch(.f(...), error = function(e) {
##             if (!quiet)
##                 message("Error: ", e$message)
##             otherwise
##         }, interrupt = function(e) {
##             stop("Terminated by user", call. = FALSE)
##         })
##     }
## }
## <bytecode: 0x7c3b2e8>
## <environment: namespace:purrr>
```

Chapter 6

Non-rectangular data

working with raw data from online services (JSON)

This chapter gives an example for processing deeply nested lists and converting them to data frames.

6.1 Traversing

Click here to show setup code.

```
library(tidyverse)
library(here)
```

We are now working with the results from downloading geolocation data from photon.komoot.de. This is stored in the file `here("data/komoot-berlin.rds")` and we can read it with `readRDS()`:

```
berlin <- readRDS(here("data/komoot-berlin.rds"))

berlin
```

```
## $features
## $features[[1]]
## $features[[1]]$geometry
## $features[[1]]$geometry$coordinates
## $features[[1]]$geometry$coordinates[[1]]
## [1] 13.38886
##
## $features[[1]]$geometry$coordinates[[2]]
## [1] 52.51704
##
```

```
##
## $features[[1]]$geometry$type
## [1] "Point"
##
##
## $features[[1]]$type
## [1] "Feature"
##
## $features[[1]]$properties
## $features[[1]]$properties$osm_id
## [1] 240109189
##
## $features[[1]]$properties$osm_type
## [1] "N"
##
## $features[[1]]$properties$country
## [1] "Germany"
##
## $features[[1]]$properties$osm_key
## [1] "place"
##
## $features[[1]]$properties$city
## [1] "Berlin"
##
## $features[[1]]$properties$osm_value
## [1] "city"
##
## $features[[1]]$properties$postcode
## [1] "10117"
##
## $features[[1]]$properties$name
## [1] "Berlin"
##
##
##
##
## $type
## [1] "FeatureCollection"
```

```
str(berlin)
```

```
## List of 2
##  $ features:List of 1
##   ..$ :List of 3
##   .. ..$ geometry  :List of 2
##   .. .. ..$ coordinates:List of 2
```

```
##   .. .. .. ..$ : num 13.4
##   .. .. .. ..$ : num 52.5
##   .. .. ..$ type       : chr "Point"
##   .. ..$ type      : chr "Feature"
##   .. ..$ properties:List of 8
##   .. .. ..$ osm_id   : int 240109189
##   .. .. ..$ osm_type : chr "N"
##   .. .. ..$ country  : chr "Germany"
##   .. .. ..$ osm_key  : chr "place"
##   .. .. ..$ city     : chr "Berlin"
##   .. .. ..$ osm_value: chr "city"
##   .. .. ..$ postcode : chr "10117"
##   .. .. ..$ name     : chr "Berlin"
##  $ type    : chr "FeatureCollection"
```

As you can see it is a somewhat complex list structure. We know from "Indexing" that we can access it's components in the following way:

```
berlin$type
```

```
## [1] "FeatureCollection"
```

```
berlin$features
```

```
## [[1]]
## [[1]]$geometry
## [[1]]$geometry$coordinates
## [[1]]$geometry$coordinates[[1]]
## [1] 13.38886
##
## [[1]]$geometry$coordinates[[2]]
## [1] 52.51704
##
##
## [[1]]$geometry$type
## [1] "Point"
##
##
## [[1]]$type
## [1] "Feature"
##
## [[1]]$properties
## [[1]]$properties$osm_id
## [1] 240109189
##
## [[1]]$properties$osm_type
## [1] "N"
##
```

```
## [[1]]$properties$country
## [1] "Germany"
##
## [[1]]$properties$osm_key
## [1] "place"
##
## [[1]]$properties$city
## [1] "Berlin"
##
## [[1]]$properties$osm_value
## [1] "city"
##
## [[1]]$properties$postcode
## [1] "10117"
##
## [[1]]$properties$name
## [1] "Berlin"
```

```
berlin$features[[1]]
```

```
## $geometry
## $geometry$coordinates
## $geometry$coordinates[[1]]
## [1] 13.38886
##
## $geometry$coordinates[[2]]
## [1] 52.51704
##
##
## $geometry$type
## [1] "Point"
##
##
## $type
## [1] "Feature"
##
## $properties
## $properties$osm_id
## [1] 240109189
##
## $properties$osm_type
## [1] "N"
##
## $properties$country
## [1] "Germany"
##
```

```
## $properties$osm_key
## [1] "place"
##
## $properties$city
## [1] "Berlin"
##
## $properties$osm_value
## [1] "city"
##
## $properties$postcode
## [1] "10117"
##
## $properties$name
## [1] "Berlin"
```

With the function `purrr::pluck()`, there is however a more universal tool available for accessing elements of more complex lists:

```
berlin %>%
  pluck("type")
```

```
## [1] "FeatureCollection"
```

```
berlin[["type"]]
```

```
## [1] "FeatureCollection"
```

```
berlin %>%
  pluck("features")
```

```
## [[1]]
## [[1]]$geometry
## [[1]]$geometry$coordinates
## [[1]]$geometry$coordinates[[1]]
## [1] 13.38886
##
## [[1]]$geometry$coordinates[[2]]
## [1] 52.51704
##
##
## [[1]]$geometry$type
## [1] "Point"
##
##
## [[1]]$type
## [1] "Feature"
##
## [[1]]$properties
```

```
## [[1]]$properties$osm_id
## [1] 240109189
##
## [[1]]$properties$osm_type
## [1] "N"
##
## [[1]]$properties$country
## [1] "Germany"
##
## [[1]]$properties$osm_key
## [1] "place"
##
## [[1]]$properties$city
## [1] "Berlin"
##
## [[1]]$properties$osm_value
## [1] "city"
##
## [[1]]$properties$postcode
## [1] "10117"
##
## [[1]]$properties$name
## [1] "Berlin"
```

```
berlin %>%
  pluck("features", 1)
```

```
## $geometry
## $geometry$coordinates
## $geometry$coordinates[[1]]
## [1] 13.38886
##
## $geometry$coordinates[[2]]
## [1] 52.51704
##
##
## $geometry$type
## [1] "Point"
##
##
## $type
## [1] "Feature"
##
## $properties
## $properties$osm_id
## [1] 240109189
```

```
##
## $properties$osm_type
## [1] "N"
##
## $properties$country
## [1] "Germany"
##
## $properties$osm_key
## [1] "place"
##
## $properties$city
## [1] "Berlin"
##
## $properties$osm_value
## [1] "city"
##
## $properties$postcode
## [1] "10117"
##
## $properties$name
## [1] "Berlin"
```

```
berlin[["features"]][[1]]
```

```
## $geometry
## $geometry$coordinates
## $geometry$coordinates[[1]]
## [1] 13.38886
##
## $geometry$coordinates[[2]]
## [1] 52.51704
##
##
## $geometry$type
## [1] "Point"
##
##
## $type
## [1] "Feature"
##
## $properties
## $properties$osm_id
## [1] 240109189
##
## $properties$osm_type
## [1] "N"
```

```
##
## $properties$country
## [1] "Germany"
##
## $properties$osm_key
## [1] "place"
##
## $properties$city
## [1] "Berlin"
##
## $properties$osm_value
## [1] "city"
##
## $properties$postcode
## [1] "10117"
##
## $properties$name
## [1] "Berlin"
```

```
berlin %>%
  pluck("features", 1, "geometry")
```

```
## $coordinates
## $coordinates[[1]]
## [1] 13.38886
##
## $coordinates[[2]]
## [1] 52.51704
##
##
## $type
## [1] "Point"
```

```
berlin[["features"]][[1]][["geometry"]]
```

```
## $coordinates
## $coordinates[[1]]
## [1] 13.38886
##
## $coordinates[[2]]
## [1] 52.51704
##
##
## $type
## [1] "Point"
```

```
berlin %>%
  pluck("features", 1, "geometry", "coordinates")
```

```
## [[1]]
## [1] 13.38886
##
## [[2]]
## [1] 52.51704
```

Similarly:

```
berlin %>%
  pluck("features", 1, "properties", "country")
```

```
## [1] "Germany"
```

And as one more important characteristic of a `tidyverse`-function, `pluck()` is pipe-able:

```
berlin %>%
  pluck("features", 1) %>%
  pluck("properties", "country")
```

```
## [1] "Germany"
```

6.1.1 Exercises

1. Introduce a variable for the first feature. Collect the coordinates, the country and the postal code.

```
first_feature <-
  berlin %>%
  ___(_____)

first_feature %>%
  ___(_____)

first_feature %>%
  ___(_____)

first_feature %>%
  ___(_____)
```

```
## NULL
```

```
## [1] "Germany"
```

```
## [1] "10117"
```

6.2 Iterating and traversing

Click here to show setup code.

```
library(tidyverse)
library(here)
```

Now we are not only working with the geolocation data for Berlin, but we are adding data for our usual suspects:

```
komoot <- readRDS(here("data/komoot.rds"))

komoot
```

```
## # A tibble: 4 x 6
##   name    url_name   url                            res     status content
##   <chr>   <chr>      <chr>                          <list>  <list> <list>
## 1 Berlin  Berlin     https://photon.komoot.de/api/~ <respo~ <NULL> <list [~
## 2 Toronto Toronto    https://photon.komoot.de/api/~ <respo~ <NULL> <list [~
## 3 Tel Av~ Tel%20Aviv https://photon.komoot.de/api/~ <respo~ <NULL> <list [~
## 4 Zürich  Z%C3%BCri~ https://photon.komoot.de/api/~ <respo~ <NULL> <list [~
```

It looks slightly different from the list `berlin` from section "Traversing". That is because we have the list-of-2 stored for each city in the column `content`. By using `pull()` on `content`, we can produce a list containing the information for all cities:

```
komoot_content <-
  komoot %>%
  pull(content)

berlin <-
  komoot_content %>%
  pluck(1)

berlin %>%
  pluck("features", 1, "geometry", "coordinates")
```

```
## [[1]]
## [1] 13.38886
##
## [[2]]
## [1] 52.51704
```

```
toronto <-
  komoot_content %>%
  pluck(2)

toronto %>%
```

```
  pluck("features", 1, "geometry", "coordinates")
```

```
## [[1]]
## [1] -79.38721
##
## [[2]]
## [1] 43.65396
```

With `map()` we can access the same element of the respective list for each city:

```
komoot_content %>%
  map(~ pluck(., "features", 1, "geometry", "coordinates"))
```

```
## [[1]]
## [[1]][[1]]
## [1] 13.38886
##
## [[1]][[2]]
## [1] 52.51704
##
##
## [[2]]
## [[2]][[1]]
## [1] -79.38721
##
## [[2]][[2]]
## [1] 43.65396
##
##
## [[3]]
## [[3]][[1]]
## [1] 34.78053
##
## [[3]][[2]]
## [1] 32.08048
##
##
## [[4]]
## [[4]][[1]]
## [1] 8.542322
##
## [[4]][[2]]
## [1] 47.3724
```

With `map()` we can also use a shorthand notation for this, without the need to use `pluck()`. We can just give it a list of the arguments which we would normally use as arguments for `pluck()`:

```
komoot_content %>%
  map(list("features", 1, "geometry", "coordinates"))
```

```
## [[1]]
## [[1]][[1]]
## [1] 13.38886
##
## [[1]][[2]]
## [1] 52.51704
##
##
## [[2]]
## [[2]][[1]]
## [1] -79.38721
##
## [[2]][[2]]
## [1] 43.65396
##
##
## [[3]]
## [[3]][[1]]
## [1] 34.78053
##
## [[3]][[2]]
## [1] 32.08048
##
##
## [[4]]
## [[4]][[1]]
## [1] 8.542322
##
## [[4]][[2]]
## [1] 47.3724
```

The access path can also be stored in a variable:

```
accessor <- list("features", 1, "geometry", "coordinates")
coordinates <-
  komoot_content %>%
  map(accessor)
coordinates
```

```
## [[1]]
## [[1]][[1]]
## [1] 13.38886
##
## [[1]][[2]]
```

```
## [1] 52.51704
##
##
## [[2]]
## [[2]][[1]]
## [1] -79.38721
##
## [[2]][[2]]
## [1] 43.65396
##
##
## [[3]]
## [[3]][[1]]
## [1] 34.78053
##
## [[3]][[2]]
## [1] 32.08048
##
##
## [[4]]
## [[4]][[1]]
## [1] 8.542322
##
## [[4]][[2]]
## [1] 47.3724
```

6.2.1 Exercises

1. Augment `komoot` with a columns containing information on the first feature only.

```
## # A tibble: 4 x 7
##   name   url_name  url                 res     status content first_feature
##   <chr>  <chr>     <chr>               <list>  <list> <list>  <list>
## 1 Berlin Berlin    https://photon.kom~ <respo~ <NULL> <list ~ <list [3]>
## 2 Toron~ Toronto   https://photon.kom~ <respo~ <NULL> <list ~ <list [3]>
## 3 Tel A~ Tel%20Av~ https://photon.kom~ <respo~ <NULL> <list ~ <list [3]>
## 4 Zürich Z%C3%BCr~ https://photon.kom~ <respo~ <NULL> <list ~ <list [3]>
```

2. Augment `komoot_first` with columns containing information on coordinates, place and postal code. Use accessors and appropriate types for the columns.

```
acc_coordinates <- _____
acc_country <- _____
komoot_first %>%
```

```
  mutate(
    coordinates = ___(___, acc_coordinates),
    country = _____,
    postcode = map_chr(_____, ~ pluck(___, .default = NA))
  )
```

```
## # A tibble: 4 x 10
##   name  url_name url   res   status content first_feature coordinates
##   <chr> <chr>    <chr> <lis> <list> <list>  <list>        <list>
## 1 Berl~ Berlin   http~ <res~ <NULL> <list ~ <list [3]>    <NULL>
## 2 Toro~ Toronto  http~ <res~ <NULL> <list ~ <list [3]>    <NULL>
## 3 Tel ~ Tel%20A~ http~ <res~ <NULL> <list ~ <list [3]>    <NULL>
## 4 Züri~ Z%C3%BC~ http~ <res~ <NULL> <list ~ <list [3]>    <NULL>
## # ... with 2 more variables: country <chr>, postcode <chr>
```

6.3 Plucking multiple locations

Click here to show setup code.

```
library(tidyverse)
library(here)

komoot <- readRDS(here("data/komoot.rds"))
komoot_content <-
  komoot %>%
  pull(content)
```

What if we want to access two different pieces of information of each main list point at once?

We are again starting in the setup with the list `komoot_content` from "Iterating and traversing".

Let's define the two locations of the city-lists we would like to access:

```
accessor_coords <- list("features", 1, "geometry", "coordinates")

komoot_content %>%
  map(accessor_coords)
```

```
## [[1]]
## [[1]][[1]]
## [1] 13.38886
##
## [[1]][[2]]
## [1] 52.51704
```

```
##
##
## [[2]]
## [[2]][[1]]
## [1] -79.38721
##
## [[2]][[2]]
## [1] 43.65396
##
##
## [[3]]
## [[3]][[1]]
## [1] 34.78053
##
## [[3]][[2]]
## [1] 32.08048
##
##
## [[4]]
## [[4]][[1]]
## [1] 8.542322
##
## [[4]][[2]]
## [1] 47.3724
```

```
accessor_country <- list("features", 1, "properties", "country")
komoot_content %>%
  map(accessor_country)
```

```
## [[1]]
## [1] "Germany"
##
## [[2]]
## [1] "Canada"
##
## [[3]]
## [1] "Israel"
##
## [[4]]
## [1] "Switzerland"
```

Combine them as a list of lists and hand it over to a `map()` inside a `map()`:

```
accessors <- list(coords = accessor_coords, country = accessor_country)

accessors %>%
  map(~ map(komoot_content, .))
```

```
## $coords
## $coords[[1]]
## $coords[[1]][[1]]
## [1] 13.38886
##
## $coords[[1]][[2]]
## [1] 52.51704
##
##
## $coords[[2]]
## $coords[[2]][[1]]
## [1] -79.38721
##
## $coords[[2]][[2]]
## [1] 43.65396
##
##
## $coords[[3]]
## $coords[[3]][[1]]
## [1] 34.78053
##
## $coords[[3]][[2]]
## [1] 32.08048
##
##
## $coords[[4]]
## $coords[[4]][[1]]
## [1] 8.542322
##
## $coords[[4]][[2]]
## [1] 47.3724
##
##
##
## $country
## $country[[1]]
## [1] "Germany"
##
## $country[[2]]
## [1] "Canada"
##
## $country[[3]]
## [1] "Israel"
##
## $country[[4]]
## [1] "Switzerland"
```

6.4 Flattening

Click here to show setup code.

```
library(tidyverse)
library(here)

komoot <- readRDS(here("data/komoot.rds"))

komoot_content <-
  komoot %>%
  pull(content)

coordinates <-
  komoot_content %>%
  map(list("features", 1, "geometry", "coordinates"))
```

It can occur that we end up with lists which are unnecessarily deep and we would like to make them flatter to make it easier to handle them.

We are starting in our setup with `komoot_content` and `coordinates` from section "Iterating and traversing".

An example for an list that seems a bit too deep is given here:

```
coordinates %>%
  pluck(1)
```

```
## [[1]]
## [1] 13.38886
##
## [[2]]
## [1] 52.51704
```

We can chop off a layer of a list and end up with a vector with one of the functions `purrr::flatten_*()`. In the `*` we need to specify what class the output will be:

```
coordinates %>%
  pluck(1) %>%
  flatten_dbl()
```

```
## [1] 13.38886 52.51704
```

Let's use `map()` to apply this to the entire list of our cities' coordinates:

```
coordinates %>%
  map(flatten_dbl)
```

```
## [[1]]
```

```
## [1] 13.38886 52.51704
##
## [[2]]
## [1] -79.38721  43.65396
##
## [[3]]
## [1] 34.78053 32.08048
##
## [[4]]
## [1]  8.542322 47.372396
```

6.5 Transposing

Click here to show setup code.

```
library(tidyverse)
library(here)

komoot <- readRDS(here("data/komoot.rds"))

komoot_content <-
  komoot %>%
  pull(content)

coordinates <-
  komoot_content %>%
  map(list("features", 1, "geometry", "coordinates"))
```

You might know the mathematical concept of transposition from your linear algebra courses. A similar concept is available in `R` when we are dealing with lists.

We are starting in our setup with `komoot_content` and `coordinates` from section "Iterating and traversing".

Let's apply `purrr::transpose()` to our list `coordinates`:

```
coordinates %>%
  transpose()
```

```
## [[1]]
## [[1]][[1]]
## [1] 13.38886
##
## [[1]][[2]]
## [1] -79.38721
```

```
##
## [[1]][[3]]
## [1] 34.78053
##
## [[1]][[4]]
## [1] 8.542322
##
##
## [[2]]
## [[2]][[1]]
## [1] 52.51704
##
## [[2]][[2]]
## [1] 43.65396
##
## [[2]][[3]]
## [1] 32.08048
##
## [[2]][[4]]
## [1] 47.3724
```

What was originally a list with 4 elements of which each one was a list of 2 elements has become a list of 2 elements of which each one is a list of 4 elements. With `flatten_dbl()` we can simplify the structure, so that we end up with a list of 2, where each element consists of a vector of 4. The first vector contains the longitude and the second the latitude of our cities:

```
coordinates_transposed <-
  coordinates %>%
  transpose() %>%
  map(~ flatten_dbl(.))
coordinates_transposed
```

```
## [[1]]
## [1]  13.388860 -79.387207  34.780527   8.542322
##
## [[2]]
## [1] 52.51704 43.65396 32.08048 47.37240
```

6.5.1 Exercises

1. Explain what happens if you transpose a tibble:

```
komoot %>%
  transpose()
```

6.6 Rectangling

Click here to show setup code.

```
library(tidyverse)
library(here)

komoot <- readRDS(here("data/komoot.rds"))

komoot_content <-
  komoot %>%
  pull(content)

coordinates_transposed <-
  komoot_content %>%
  map(list("features", 1, "geometry", "coordinates")) %>%
  transpose() %>%
  map(~ flatten_dbl(.))
```

Most of us R-users feel most at ease when dealing in R with data frames on which we can use a plethora of well-known (by us) functions with non-startling behaviour. What if we don't get our data in such a form?

We are starting in our setup with the list `coordinates_transposed` from section "Transposing".

A tibble is internally a list of named vectors of equal length. In two easy steps we can therefore make a tibble out of the unnamed list `coordinates_transposed`:

```
coordinates_transposed %>%
  rlang::set_names(c("lon", "lat"))
```

```
## $lon
## [1]  13.388860 -79.387207  34.780527   8.542322
##
## $lat
## [1] 52.51704 43.65396 32.08048 47.37240
```

```
coordinates_transposed %>%
  rlang::set_names(c("lon", "lat")) %>%
  as_tibble()
```

```
## # A tibble: 4 x 2
##      lon   lat
##    <dbl> <dbl>
## 1  13.4   52.5
## 2 -79.4   43.7
## 3  34.8   32.1
```

```
## 4   8.54  47.4
```

If you want to keep the names open for now, but still get a tibble, you can set `as_tibble()`'s argument `.name_repair = "universal"`:

```
coordinates_transposed %>%
  as_tibble(.name_repair = "universal")
```

```
## New names:
## * `` -> ...1
## * `` -> ...2
```

```
## # A tibble: 4 x 2
##     ...1  ...2
##    <dbl> <dbl>
## 1  13.4   52.5
## 2 -79.4   43.7
## 3  34.8   32.1
## 4   8.54  47.4
```

```
coordinates_transposed %>%
  as_tibble(.name_repair = "universal") %>%
  rename(lon = ...1, lat = ...2)
```

```
## New names:
## * `` -> ...1
## * `` -> ...2
```

```
## # A tibble: 4 x 2
##      lon   lat
##    <dbl> <dbl>
## 1  13.4   52.5
## 2 -79.4   43.7
## 3  34.8   32.1
## 4   8.54  47.4
```

6.7 Accessing APIs

Click here to show setup code.

```
library(tidyverse)
library(here)
```

When dealing with web-APIs, query results come frequently in the JSON (**J**ava**S**cript **O**bject **N**otation) format. How to deal with this in R? A way to "talk" with APIs is provided by the package {httr}. The "GET"-query is executed by using `httr::GET()` with the URL, containing the query specifics, as an argument:

```
req <- httr::GET("https://photon.komoot.de/api/?q=Paradeplatz&limit=3")
```

If you are using this command in a script, you need to wait until the query is finished processing:

```
httr::stop_for_status(req)
```

The result of the query can be accessed via `httr::content()`:

```
content <- httr::content(req)
content
```

```
## $features
## $features[[1]]
## $features[[1]]$geometry
## $features[[1]]$geometry$coordinates
## $features[[1]]$geometry$coordinates[[1]]
## [1] 8.538948
##
## $features[[1]]$geometry$coordinates[[2]]
## [1] 47.36981
##
##
## $features[[1]]$geometry$type
## [1] "Point"
##
##
## $features[[1]]$type
## [1] "Feature"
##
## $features[[1]]$properties
## $features[[1]]$properties$osm_id
## [1] 905841
##
## $features[[1]]$properties$osm_type
## [1] "R"
##
## $features[[1]]$properties$extent
## $features[[1]]$properties$extent[[1]]
## [1] 8.538163
##
## $features[[1]]$properties$extent[[2]]
## [1] 47.37027
##
## $features[[1]]$properties$extent[[3]]
## [1] 8.539516
##
```

```
## $features[[1]]$properties$extent[[4]]
## [1] 47.36935
##
##
## $features[[1]]$properties$country
## [1] "Switzerland"
##
## $features[[1]]$properties$osm_key
## [1] "highway"
##
## $features[[1]]$properties$city
## [1] "Zurich"
##
## $features[[1]]$properties$osm_value
## [1] "pedestrian"
##
## $features[[1]]$properties$postcode
## [1] "8001"
##
## $features[[1]]$properties$name
## [1] "Paradeplatz"
##
## $features[[1]]$properties$state
## [1] "Zurich"
##
##
##
## $features[[2]]
## $features[[2]]$geometry
## $features[[2]]$geometry$coordinates
## $features[[2]]$geometry$coordinates[[1]]
## [1] 7.108249
##
## $features[[2]]$geometry$coordinates[[2]]
## [1] 50.88602
##
##
## $features[[2]]$geometry$type
## [1] "Point"
##
##
## $features[[2]]$type
## [1] "Feature"
##
## $features[[2]]$properties
## $features[[2]]$properties$osm_id
```

```
## [1] 389550464
##
## $features[[2]]$properties$osm_type
## [1] "N"
##
## $features[[2]]$properties$country
## [1] "Germany"
##
## $features[[2]]$properties$osm_key
## [1] "place"
##
## $features[[2]]$properties$city
## [1] "Cologne"
##
## $features[[2]]$properties$osm_value
## [1] "locality"
##
## $features[[2]]$properties$postcode
## [1] "51147"
##
## $features[[2]]$properties$name
## [1] "Paradeplatz"
##
## $features[[2]]$properties$state
## [1] "North Rhine-Westphalia"
##
##
##
## $features[[3]]
## $features[[3]]$geometry
## $features[[3]]$geometry$coordinates
## $features[[3]]$geometry$coordinates[[1]]
## [1] 8.684258
##
## $features[[3]]$geometry$coordinates[[2]]
## [1] 49.38674
##
##
## $features[[3]]$geometry$type
## [1] "Point"
##
##
## $features[[3]]$type
## [1] "Feature"
##
## $features[[3]]$properties
```

```
## $features[[3]]$properties$osm_id
## [1] 391678888
##
## $features[[3]]$properties$osm_type
## [1] "W"
##
## $features[[3]]$properties$extent
## $features[[3]]$properties$extent[[1]]
## [1] 8.683635
##
## $features[[3]]$properties$extent[[2]]
## [1] 49.38719
##
## $features[[3]]$properties$extent[[3]]
## [1] 8.68488
##
## $features[[3]]$properties$extent[[4]]
## [1] 49.3863
##
##
## $features[[3]]$properties$country
## [1] "Germany"
##
## $features[[3]]$properties$osm_key
## [1] "place"
##
## $features[[3]]$properties$city
## [1] "Heidelberg"
##
## $features[[3]]$properties$osm_value
## [1] "locality"
##
## $features[[3]]$properties$postcode
## [1] "69120"
##
## $features[[3]]$properties$name
## [1] "Paradeplatz"
##
## $features[[3]]$properties$state
## [1] "Baden-Württemberg"
##
##
##
##
## $type
## [1] "FeatureCollection"
```

As you can see, the result, as it is displayed in R, is already a nested list at this point. And we know how to deal with these objects.

The object did originally come as a JSON object though, which you can see if you look at the literal result of the query:

```
text_content <- httr::content(req, as = "text")
cat(text_content)
```

```
## {"features":[{"geometry":{"coordinates":[8.538948327037028,47.369806999999994],"type
```

The package {jsonlite} has the function `jsonlite::prettify()` to offer, in order to display a one-line JSON-structure in a more clearly laid-out manner:

```
cat(jsonlite::prettify(text_content))
```

```
## {
##     "features": [
##         {
##             "geometry": {
##                 "coordinates": [
##                     8.538948327037028,
##                     47.369806999999994
##                 ],
##                 "type": "Point"
##             },
##             "type": "Feature",
##             "properties": {
##                 "osm_id": 905841,
##                 "osm_type": "R",
##                 "extent": [
##                     8.5381631,
##                     47.3702704,
##                     8.5395156,
##                     47.3693475
##                 ],
##                 "country": "Switzerland",
##                 "osm_key": "highway",
##                 "city": "Zurich",
##                 "osm_value": "pedestrian",
##                 "postcode": "8001",
##                 "name": "Paradeplatz",
##                 "state": "Zurich"
##             }
##         },
##         {
##             "geometry": {
##                 "coordinates": [
```

```
##                     7.1082488,
##                     50.8860177
##                 ],
##                 "type": "Point"
##             },
##             "type": "Feature",
##             "properties": {
##                 "osm_id": 389550464,
##                 "osm_type": "N",
##                 "country": "Germany",
##                 "osm_key": "place",
##                 "city": "Cologne",
##                 "osm_value": "locality",
##                 "postcode": "51147",
##                 "name": "Paradeplatz",
##                 "state": "North Rhine-Westphalia"
##             }
##         },
##         {
##             "geometry": {
##                 "coordinates": [
##                     8.684257597277371,
##                     49.3867415
##                 ],
##                 "type": "Point"
##             },
##             "type": "Feature",
##             "properties": {
##                 "osm_id": 391678888,
##                 "osm_type": "W",
##                 "extent": [
##                     8.6836355,
##                     49.3871856,
##                     8.6848797,
##                     49.3862973
##                 ],
##                 "country": "Germany",
##                 "osm_key": "place",
##                 "city": "Heidelberg",
##                 "osm_value": "locality",
##                 "postcode": "69120",
##                 "name": "Paradeplatz",
##                 "state": "Baden-Württemberg"
##             }
##         }
##     ],
```

```
##     "type": "FeatureCollection"
## }
```

Chapter 7

Tidy evaluation

writing functions that work with datasets of different shape

This chapter offers an introduction to tidy evaluation.

Knowledge of tidy evaluation can become necessary when creating own functions in the framework of the tidyverse. The code in this chapter requires the rlang package, which provides the functions required for tidy evaluation, in addition to the tidyverse.

```
library(tidyverse)
library(rlang)
```

```
##
## Attaching package: 'rlang'
```

```
## The following objects are masked from 'package:purrr':
##
##     %@%, as_function, flatten, flatten_chr, flatten_dbl,
##     flatten_int, flatten_lgl, flatten_raw, invoke, list_along,
##     modify, prepend, splice
```

7.1 A custom plotting function

Let's try to build a function that takes a data frame and an unquoted column name and produces a histogram from this column. A naive approach would be the following function definition:

```
tidy_histogram <- function(.data, x) {
  .data %>%
    ggplot(aes(x = x)) +
```

```
    geom_histogram()
}
```

Let's test this function in an easy setting:

```
data <- tibble(a = 1:10)

try(print(
  data %>%
    tidy_histogram(a)
))
```

```
## Error in FUN(X[[i]], ...) : object 'a' not found
```

```
try(print(
  data %>%
    tidy_histogram("a")
))
```

```
## Error : StatBin requires a continuous x variable: the x variable is discrete. Perha
```

Neither the first nor the second attempt worked. What went wrong?

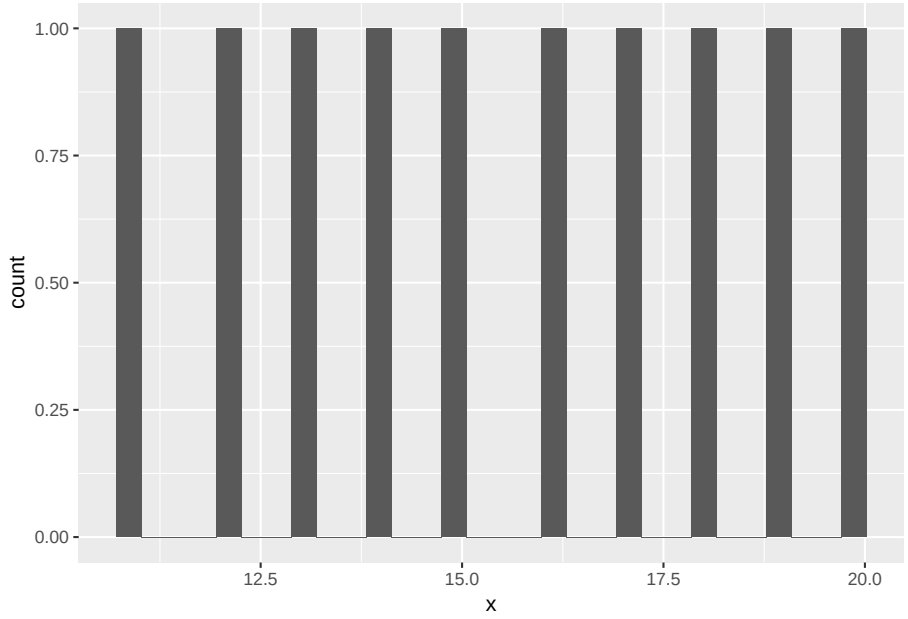
If we add another column called `x`, it suddenly works:

```
data <- tibble(a = 1:10, x = 11:20)

data %>%
```

```
  tidy_histogram(a)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The reason for it is, that our function is hard-coded to display a variable called `x`. The problem lies in the code-snippet `aes(x = x)`.

In the tidyverse the solution for avoiding these ambiguities is by using expressions to capture the meaning (here: user input) of an unquoted variable and subsequently the bang-bang-operator (written in the code as `!!`; also called "unquote") to access/pass on this meaning at the right place.

In our example we do the following:

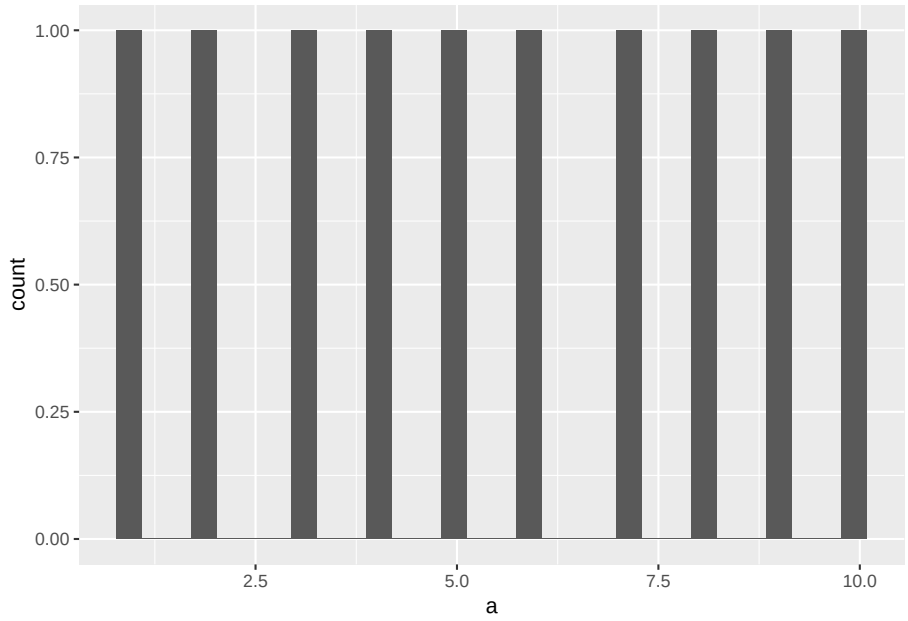
```
tidy_histogram <- function(.data, x) {
  # Treat the argument as a variable name
  expr <- enquo(x)

  .data %>%
    # Tell ggplot2 that expr *contains* the name of the variable,
    # instead of expecting a variable named `expr`
    ggplot(aes(x = !!expr)) +
    geom_histogram()
}

data %>%
```

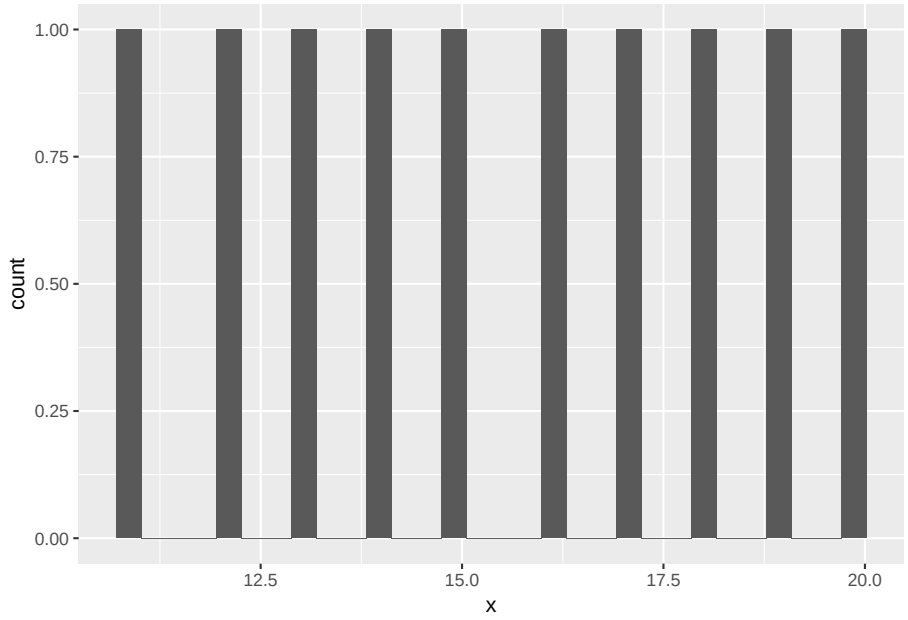
```
  tidy_histogram(a)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
data %>%
  tidy_histogram(x)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
try(print(
  data %>%
    tidy_histogram(y)
))
```

```
## Error in FUN(X[[i]], ...) : object 'y' not found
```

But this behavior is different from our usual base-R usage of variables in functions. How do some functions behave this way and others another way? Let's have a look at how the 'problematic' tidyverse function `aes()` is implemented:

```
aes
```

```
## function (x, y, ...)
## {
##     exprs <- rlang::enquos(x = x, y = y, ...)
##     is_missing <- vapply(exprs, rlang::quo_is_missing, logical(1))
##     aes <- new_aes(exprs[!is_missing], env = parent.frame())
##     rename_aes(aes)
## }
## <bytecode: 0x4d808c0>
## <environment: namespace:ggplot2>
```

The reason for the behavior seen above is that the function itself makes use of capturing user input as an expression. In this case it uses the function `enquos()`, which captures one or more expressions along with an unique identifier for the environment in which they are supposed to be evaluated eventually. The default is to evaluate an expression ("standard evaluation"). With `enquo()` and `enquos()` the expressions that correspond to user input are captured.

As an example of a newly-built function in the tidyverse, here is a function that combines the functionalities of `dplyr::mutate()` and `purrr::map_dbl()`. It takes as arguments a data frame, the column it is supposed to act upon and the function call it is supposed to use on each of the columns values:

```
mutate_map_dbl <- function(.data, col, expr) {
  quo <- enquo(col)

  .data %>%
    mutate(new_column = map_dbl(!!quo, expr))
}

iris_nested <-
  iris %>%
  nest(-Species)

iris_nested %>%
  mutate_map_dbl(data, ~ mean(.$Petal.Width))
```

```
## # A tibble: 3 x 3
##   Species    data              new_column
##   <fct>      <list>                 <dbl>
## 1 setosa     <tibble [50 x 4]>      0.246
## 2 versicolor <tibble [50 x 4]>      1.33
```

```
## 3 virginica  <tibble [50 x 4]>      2.03
```

7.2 Do you need tidy evaluation?

Click here to show setup code.

```
library(tidyverse)
```

So does everyone who wants to create any functions in the framework of the tidyverse need deep knowledge about tidy evaluation?

The answer is, it depends: often enough, things "just work". In the following example, which is a slight extension of `dplyr::summarize()`, you do not need to capture any expressions. The function takes a data frame and an ellipsis. And the ellipsis can be directly passed on to a tidyverse function (buzzphrase: "pass the dots").

```
summarize_ungroup <- function(.data, ...) {
  .data %>%
    summarize(...) %>%
    ungroup()
}
```

The function does what it promised to do:

```
mean_airtime_per_day <-
  nycflights13::flights %>%
  group_by(year, month, day) %>%
  summarize_ungroup(mean(air_time, na.rm = TRUE))

mean_airtime_per_day
```

```
## # A tibble: 365 x 4
##    year month   day `mean(air_time, na.rm = TRUE)`
##   <int> <int> <int>                            <dbl>
## 1  2013     1     1                             170.
## 2  2013     1     2                             162.
## 3  2013     1     3                             157.
## # ... with 362 more rows
```

```
mean_airtime_per_day %>%
  groups()
```

```
## NULL
```

7.3 Explicit quote-unquote of ellipsis

Click here to show setup code.

```
library(tidyverse)
library(rlang)
```

There are cases though, when you need knowledge about what the user added to the ellipsis. This is then handled by capturing the content in a list of quosures, which can be unquoted by the `!!!`-operator. You need the "triple-bang" operator here, because the ellipsis can hold more than one expression. `!!!` does two things: it unquotes the content and splices it into the current call.

We can practice this with our little new tidyverse function `summarize_ungroup()`:

```
summarize_ungroup <- function(.data, ...) {
  # Capture (quote) with enquos()
  quos <- enquos(...)

  # Use (unquote-splice) with !!!
  .data %>%
    summarize(!!!quos) %>%
    ungroup()
}
```

We didn't need to process the content, but it still works, as we can see here:

```
mean_airtime_per_day <-
  nycflights13::flights %>%
  group_by(year, month, day) %>%
  summarize_ungroup(mean(air_time, na.rm = TRUE))

mean_airtime_per_day
```

```
## # A tibble: 365 x 4
##    year month   day `mean(air_time, na.rm = TRUE)`
##   <int> <int> <int>                          <dbl>
## 1  2013     1     1                           170.
## 2  2013     1     2                           162.
## 3  2013     1     3                           157.
## # ... with 362 more rows
```

```
mean_airtime_per_day %>%
  groups()
```

```
## NULL
```

To come back to our original example: `aes()` uses exactly this "quote-unquote-splice"-pattern:

```
aes
```

```
## function (x, y, ...)
## {
##     exprs <- rlang::enquos(x = x, y = y, ...)
##     is_missing <- vapply(exprs, rlang::quo_is_missing, logical(1))
##     aes <- new_aes(exprs[!is_missing], env = parent.frame())
##     rename_aes(aes)
## }
## <bytecode: 0x4d808c0>
## <environment: namespace:ggplot2>
```

At the time of producing the material for this course, `summarize()` did not make use of this pattern.

7.4 Names

Click here to show setup code.

```
library(tidyverse)
library(rlang)
```

User input in an ellipsis can be named or unnamed. We can distinguish between those two kinds and make use of this distinction, in the following way creating a special interface for the function:

```
gsu <- function(.data, ...) {
  # Capture (quote) with enquos()
  quos <- enquos(...)

  is_named <- (names2(quos) != "")
  named_quos <- quos[is_named]
  unnamed_quos <- quos[!is_named]

  # Use (unquote-splice) with !!!
  .data %>%
    group_by(!!!unnamed_quos) %>%
    summarize(!!!named_quos) %>%
    ungroup()
}
```

The `named_quos` are our summary columns (name is name of the new column, value is the expression to be used on the input column(s)) and the `unnamed_quos` are now the grouping columns:

```
mean_airtime_per_day <-
  nycflights13::flights %>%
```

```
  gsu(year, month, day, mean_air_time = mean(air_time, na.rm = TRUE))

mean_airtime_per_day
```

```
## # A tibble: 365 x 4
##    year month   day mean_air_time
##   <int> <int> <int>         <dbl>
## 1  2013     1     1          170.
## 2  2013     1     2          162.
## 3  2013     1     3          157.
## # ... with 362 more rows
```

7.5 Debugging

Click here to show setup code.

```
library(rlang)
```

You can use the capturing functions (creating `quosures` or `expressions`) also outside of functions:

```
quos(x = a)
```

```
## $x
## <quosure>
## expr: ^a
## env:  0x36a96e8
```

```
a <- sym("b")
x_quos <- quos(x = !!a)
x_quos
```

```
## $x
## <quosure>
## expr: ^b
## env:  0x36a96e8
```

The `sym()` function here creates a so-called `symbol` from a `character` variable. Unquoting a symbol variable means that the symbol is interpreted as a variable in the dataset.

Capturing expressions in quosures can help you understand what is happening behind the scenes and for example give you clues as to why your code is not doing what it is supposed to do.

Quosures can also be nested:

```
quos(y = c, !!!x_quos)
```

```
## $y
## <quosure>
## expr: ^c
## env:  0x36a96e8
##
## $x
## <quosure>
## expr: ^b
## env:  0x36a96e8
```

7.6 Argument names

Click here to show setup code.

```
library(tidyverse)
library(rlang)
```

At the end of section A custom plotting function we defined a function `mutate_map_dbl()`. A downside of this function was, that it did create the desired new column, but you weren't able to specify the column name in the function call.

Let's try to improve this aspect of the function:

```
mutate_map_dbl <- function(.data, col, ...) {
  quos <- build_quos(!!enquo(col), ...)
  .data %>%
    mutate(!!!quos)
}

build_quos <- function(col, ...) {
  args <- list(...)
  stopifnot(length(args) == 1)

  expr <- args[[1]]

  map_quo <- build_map_quo(!!enquo(col), expr)

  set_names(list(map_quo), names(args))
}

build_map_quo <- function(col, expr) {
  quo <- enquo(col)
```

```
  quo(map_dbl(!!quo, expr))
}
```

Again, like in section Names we are able to make use of the fact that an expression in an ellipsis of sorts `x = y` is treated in a way, that `x` is the name and `y` is the value of an object (here we used this in the code snippet `names(args)`).

A lot of things happen here: 1. the main function `mutate_map_dbl()` calls a helper function `build_quos()` 2. `build_quos()` in turn calls a helper's helper function `build_map_quo()` 3. `mutate_map_dbl()` then uses the output of the nested function calls to (hopefully) produce the desired result.

What output do the two helper functions produce? Let's test it:

```
build_quos(data, mean_petal_width = ~ mean(.$Petal.Width))
```

```
## $mean_petal_width
## <quosure>
## expr: ^map_dbl(^data, expr)
## env:  0xb7b7ef8
```

```
build_map_quo(mean_petal_width, ~ mean(.$Petal.Width))
```

```
## <quosure>
## expr: ^map_dbl(^mean_petal_width, expr)
## env:  0xbf54580
```

And finally, let's see if our function lives up to our expectations:

```
iris %>%
  nest(-Species) %>%
  mutate_map_dbl(data, mean_petal_width = ~ mean(.$Petal.Width))
```

```
## # A tibble: 3 x 3
##   Species    data              mean_petal_width
##   <fct>      <list>                       <dbl>
## 1 setosa     <tibble [50 x 4]>            0.246
## 2 versicolor <tibble [50 x 4]>            1.33
## 3 virginica  <tibble [50 x 4]>            2.03
```

7.7 purrr-style mappers

Click here to show setup code.

```
library(tidyverse)
library(rlang)

mutate_map_dbl <- function(.data, col, ...) {
```

```
  quos <- build_quos(!!enquo(col), ...)
  .data %>%
    mutate(!!!quos)
}
```

Is there still potential to improve our function `mutate_map_dbl()`?

We start here with only the highest level function definition of `mutate_map_dbl()` from section Argument names, i.e. without the definition of its helper functions `build_quos()` and `build_map_quo()`.

What we are trying to achieve now, is to rid ourselves from the need to provide the tilde before the function. This is a slightly more tricky task, and here is how to go about it:

```
build_quos <- function(col, ...) {
  args <- enquos(...)
  stopifnot(length(args) == 1)

  expr <- args[[1]]

  map_quo <- build_map_quo(!!enquo(col), !!expr)

  set_names(list(map_quo), names(args))
}

build_map_quo <- function(col, expr) {
  quo <- enquo(col)
  mapper <- as_mapper_quosure(!!enquo(expr))
  quo(map_dbl(!!quo, !!mapper))
}

as_mapper_quosure <- function(expr) {
  quo <- enquo(expr)

  rlang::new_function(
    alist(... = , . = ..1, .x = ..1, .y = ..2),
    quo_get_expr(quo),
    quo_get_env(quo)
  )
}
```

We needed to add one more level in the hierarchy of function calling. The helper function `as_mapper_quosure()` creates a new function with the help of `rlang::new_function()`, which eventually makes it possible to leave out the tilde.

```
as_mapper(~ mean(.$Petal.Width))
```

```
## <lambda>
## function (..., .x = ..1, .y = ..2, . = ..1)
## mean(.$Petal.Width)
## <environment: 0x36a96e8>
## attr(,"class")
## [1] "rlang_lambda_function"
```

```
as_mapper_quosure(mean(.$Petal.Width))
```

```
## function (..., . = ..1, .x = ..1, .y = ..2)
## mean(.$Petal.Width)
## <environment: 0x36a96e8>
```

```
build_map_quo(mean_petal_width, mean(.$Petal.Width))
```

```
## <quosure>
## expr: ^map_dbl(^mean_petal_width, <function(..., . = ..1, .x = ..1,
##           .y = ..2) mean(.$Petal.Width)>)
## env:  0x9d7ee30
```

We see that our function `as_mapper_quosure()` is closely related to the function `purrr::as_mapper()`, but produces a quosure of a proper function and not a lambda function. Also, it does not require the tilde.

So much to the theory, but does our main function also still behave in the right way?

```
iris %>%
  nest(-Species) %>%
  mutate_map_dbl(data, mean_petal_width = mean(.$Petal.Width))
```

```
## # A tibble: 3 x 3
##   Species    data              mean_petal_width
##   <fct>      <list>                       <dbl>
## 1 setosa     <tibble [50 x 4]>            0.246
## 2 versicolor <tibble [50 x 4]>            1.33
## 3 virginica  <tibble [50 x 4]>            2.03
```

Chapter 8

Best practices

R code is often organized in packages that can be installed from centralized repositories such as CRAN or GitHub. If you are new to writing R packages, this course cannot give a complete introduction into packages. It is still useful to embrace some very few concepts of R packages to gain access to a vast toolbox and also organize your code in a standardized way familiar to other users. With the first steps in place, the road to your first R package may become less steep.

- Create a `DESCRIPTION` file to declare dependencies and allow easy reloading of the functions you define
- Store your functions in `.R` files in the `R/` directory in your project
  - Scripts that you execute live in `script/` or a similar directory
- Use roxygen2 to document your functions close to the source
- Write tests for your functions, e.g. with testthat

See R packages for a more comprehensive treatment.

8.1 DESCRIPTION

Create and open a new RStudio project. Then, create a `DESCRIPTION` file with `usethis::use_description()`:

```
# install.packages("usethis")
usethis::use_description()
```

Double-check success:

```
# install.packages("devtools")
devtools::load_all()
```

Declare that your project requires the tidyverse and the here package:

```
usethis::use_package("here")
# Currently doesn't work, add manually
# https://github.com/r-lib/usethis/issues/760
# usethis::use_package("tidyverse")
```

8.2 R

With a `DESCRIPTION` file defined, create a new `.R` file and save it in the `R/` directory. (Create this directory if it does not exist.) Create a function in this file, save the file:

```
hi <- function(text = "Hello, world!") {
  print(text)
  invisible(text)
}
```

Do not source the file.

Restart R (with Ctrl + Shift + F10 in RStudio).

Run `devtools::load_all()` again, you can use the shortcut Ctrl + Shift + L or Cmd + Shift + L in RStudio.

Check that you can run `hi()` in the console:

```
hi()
```

```
## [1] "Hello, world!"
```

```
hi("Wow!")
```

```
## [1] "Wow!"
```

Edit the function:

```
hi <- function(text = "Wow!") {
  print(text)
  invisible(text)
}
```

Save the file, but do not source it.

Run `devtools::load_all()` again, you can use the shortcut Ctrl + Shift + L or Cmd + Shift + L in RStudio.

Check that the new implementation of `hi()` is active:

```
hi()
```

```
## [1] "Wow!"
```

All functions that are required for your project are stored in this directory. Do not store executable scripts, use a `script/` directory.

8.3 roxygen2

The following intuitive annotation syntax is a standard way to create documentation for your functions:

```
#' Print a welcome message
#'
#' This function prints "Wow!", or a custom text, on the console.
#'
#' @param text The text to print, "Wow!" by default.
#'
#' @return The `text` argument, invisibly.
#'
#' @examples
#' hi()
#' hi("Hello!")
hi <- function(text = "Wow!") {
  print(text)
  invisible(text)
}
```

This annotation can be rendered to a nicely looking HTML page with the roxygen2 and pkgdown packages. All you need to do is provide (and maintain) it.

8.4 testthat

Automated tests make sure that the functions you write today continue working tomorrow. Create your first test with `usethis::use_test()`:

```
# install.packages("testthat")
usethis::use_test("hi")
```

The file `tests/testthat/test-hi.R` is created, with the following contents:

```
test_that("multiplication works", {
  expect_equal(2 * 2, 4)
})
```

Replace this predefined text with a test that makes more sense for us:

```
test_that("hi() works", {
  expect_output(hi(), "Wow")
  expect_output(hi("Hello"), "Hello")
})
```

Run the new test with `devtools::test()`, you can use the shortcut Ctrl + Shift + T or Cmd + Shift + T in RStudio.

Check that the test actually detects failures by modifying the implementation of `hi()` and rerunning the test:

```
hi <- function(text = "Oops!") {
  print(text)
  invisible(text)
}
```

Run the new test with `devtools::test()`, you can use the shortcut Ctrl + Shift + T or Cmd + Shift + T in RStudio. One test should be failing now.

Chapter 9

- R for data science: https://r4ds.had.co.nz/
- Row oriented workflows: https://github.com/jennybc/row-oriented-workflows#readme
- Advanced R: http://adv-r.had.co.nz/
- Tidy evaluation: https://tidyeval.tidyverse.org/
- R packages: http://r-pkgs.had.co.nz/
- roxygen2: Vignettes in https://cran.r-project.org/package=roxygen2, especially:
  - Introduction to roxygen2
  - Generating Rd files for an overview of available tags
  - Write R documentation in Markdown
- How R searches and finds stuff: http://blog.obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/
- What they forgot to teach you: https://whattheyforgot.org/
- Parallel processing with a purrr-like interface: https://davisvaughan.github.io/furrr/
- Tidyverse principles: https://principles.tidyverse.org/
- Recursive lists to use in teaching and examples: https://github.com/jennybc/repurrrsive